Engs 31 / CoSc 56

Final Project Report

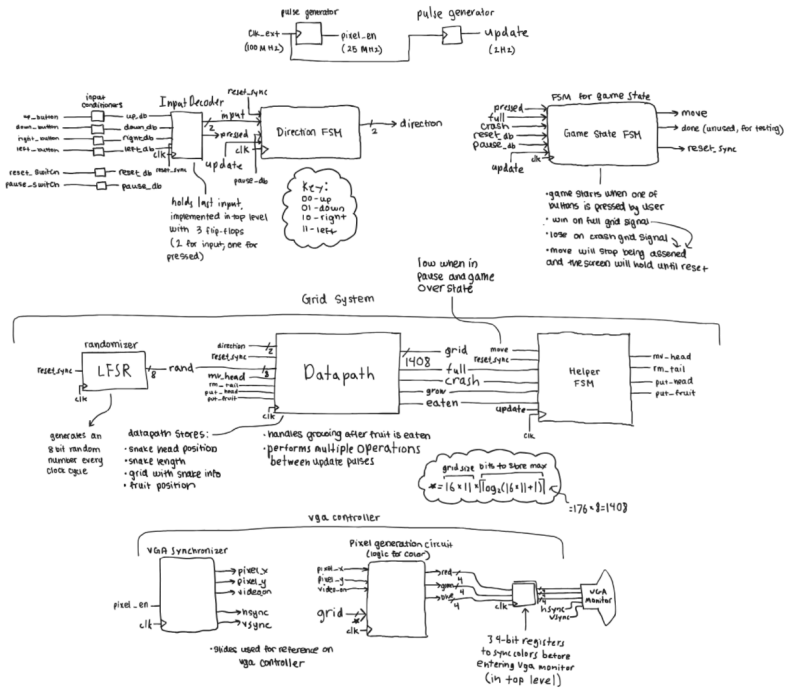
Owen Chen and Mary Haferd

Abstract: This report covers how we designed, implemented, and validated our final project for the course, which was to create the classic arcade game known as Snake. At a high level, Snake allows the player to control a snake, which can move across the screen. The objective of Snake is to collect as many fruits as possible, which causes the Snake to grow in length. Crashing the Snake is the loss condition, and the player must avoid both the border of the grid the game takes place on and the body of the snake itself. A more precise specification of Snake is given in the next section. The key components we created as part of our system are a VGA synchronizer, pixel generation circuit, LFSR randomizer, snake datapath, helper FSM, direction FSM, game state FSM, and an input conditioner. They are detailed extensively in the second section. We also tested these extensively, performing both behavioral simulation and hardware validation for each component except for the input conditioner, which was already tested beforehand. The testing we conducted is detailed in the third section. Ultimately, we succeeded in passing all of the tests we created, and our final implementation of Snake matched perfectly with what we expected.

1. System Overview

Our goal was to create the game Snake, using the buttons and switches on the FPGA as input and using a VGA as output to display the game. This means that the snake (player character) should be able to move around the grid (16x11 space on which the game occurs) according to the user's input. To be exact, movement typically entails removing the snake's tail and extending the snake in the direction the player indicates periodically (we chose 2Hz as the rate at which this occurs). Additionally, the snake is not able to move backwards; it can only continue to go in its current direction or switch to a direction perpendicular to it. On the grid, the snake can encounter one of four things as it moves: (1) empty space, which has no restrictions; (2) the border, which causes it to crash and the player to lose; (3) its own body, which also results in a crash; and (4) the fruit. When the snake eats the fruit by moving onto its square, it lengthens (this is implemented by not removing the tail but still extending the snake) and another fruit is randomly placed on an empty square of the grid. We also allow for the game to be paused and reset with switches. These are the specifications that our implementation must follow.

1.1. Top-level Block Diagram



1.2. Description of Ports

Our input ports are (1) clk_ext, the 100 MHz clock that goes to each of our components; (2-5) up_button, down_button, right_button, and left_button, which are connected to the buttons on the FPGA and allow the user to control direction; (6) pause_switch, indicating to several components when the game is paused; and (7) reset_switch, which instead of

resetting the system directly, tells the game state FSM to enter a reset state (the output of this state instructs all components with memory to return to their initial state). Our output ports are (1-2) vga_hsync and vga_vsync, which are sent to the VGA display and become low when the display should start a new line and a new frame respectively, in addition to (3-5) vga_red, vga_green, and vga_blue, which are 4-bit vectors indicating the color of the pixel that should be drawn by the VGA (updating at ~25 MHz).

1.3. Description of Components

The VGA synchronizer and pixel generation circuit work in conjunction to control the VGA. The VGA synchronizer tells the circuit which pixel on the VGA to generate the color for, in addition to telling the display when to draw a new line and a new frame via hsync and vsync. The pixel generation circuit maps values on the game grid to colors, and sends those colors to the VGA pixel by pixel. The circuit also deals with colors outside of the grid, such as the border and background colors.

The LFSR randomizer, datapath, and helper FSM work together to manage primarily the grid. The LFSR generates 8-bit random numbers which the datapath uses to replace the fruit after it is eaten by the snake. The helper FSM communicates with the datapath in order to determine which operation (move head, remove tail, place head, place fruit) it should execute. The datapath applies these operations to the data (snake head position, snake length, grid with snake information, fruit position), which it stores as registers. Both the datapath and the helper FSM also receive information such as snake direction and game state from the other FSMs as well.

The remaining components are the various input conditioners, direction FSM, and game state FSM. The input conditioners synchronize and debounce raw user input (buttons and switches) and send them to our components. The direction FSM determines the direction in which the snake faces using the debounced inputs, and ensures that the snake faces in a valid direction (for example, a snake cannot go from up to down directly, it must first go to a perpendicular direction such as left and right). The game state FSM determines the state of the game, such as whether it is idle (awaiting the first press), active (the game is running), over (lose or win), paused, or reset.

For glue logic, we decoded the 4 direction inputs (the up, down, right, and left buttons) into a 2-bit vector (“00”, “01”, “10”, and “11” respectively) and stored that in a register going into the direction FSM. This is done because the direction FSM only updates once every 2 Hz (game update rate), and without the registers inputs sent in between update pulses are ignored. There is also a similar 1-bit register that stores 1 once a button is pressed and holds that until reset. In addition, three 4-bit registers receive the RGB values from the pixel generation circuit and send them to the VGA display; this is done in order to synchronize the output, which fixes minor visual glitches on the VGA. Finally, two simple pulse generators are used to create the enable for the VGA synchronizer and the game update pulse.

2. Technical Description

We utilize the following components in our design: (1) VGA synchronizer, (2) pixel generation circuit, (3) LFSR randomizer, (4) snake datapath, (5) datapath helper FSM, (6) direction FSM, (7) game state FSM, and (8) input conditioner.

2.1. VGA Synchronizer

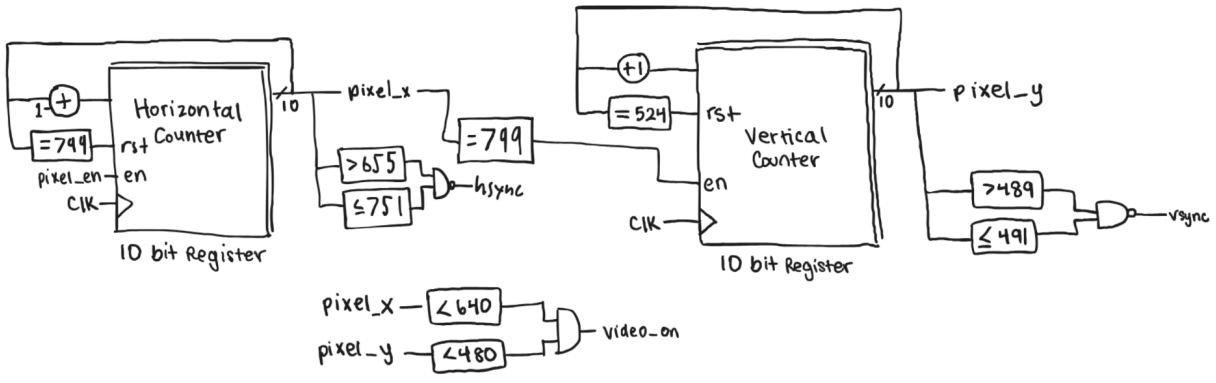
As mentioned earlier, the VGA synchronizer determines the exact pixel location for which the circuit must generate color, while also providing hsync and vsync.

2.1.1. Description of Ports

Our inputs for this component are (1) clk, the 100 MHz clock that we use for everything, and (2) pixel_en, a 25 MHz pulse that allows the horizontal counter increment when asserted. Our outputs are (1-2) pixel_x and pixel_y, which are 10-bit vectors indicating the pixel on the VGA display being drawn (these are values in the horizontal and vertical counters respectively); (3) video_on, which is low when the pixel position is off screen; (4) hsync, which tells the VGA display to start a new line when low (during the retrace period of the horizontal counter, between the front and back porch); and (5) vsync, which tells the VGA display to start a new frame when low (during the retrace period of the vertical counter).

2.1.2. Register Transfer Level Diagram

Both counters have their outputs plus 1 going into themselves, which means that they increment when enabled. They synchronously reset if the current value in the counter is the total counter. A comparator checking whether the horizontal counter has reached its total enables the vertical counter because we want pixel positions in row-major order. Comparators also check whether the value of the counters are within a range, which determines when hsync and vsync go low. The values of both counters are considered when determining whether the video is on or not.



2.1.3. Description of Memory

We have two 10-bit registers for each counter. The signal pixel_en is the enable for the horizontal counter. As mentioned previously, a comparator checking the total count of both registers produces their reset signals. In addition, a comparator checking the total count of the horizontal counter enables the vertical counter.

2.2. Pixel Generation Circuit

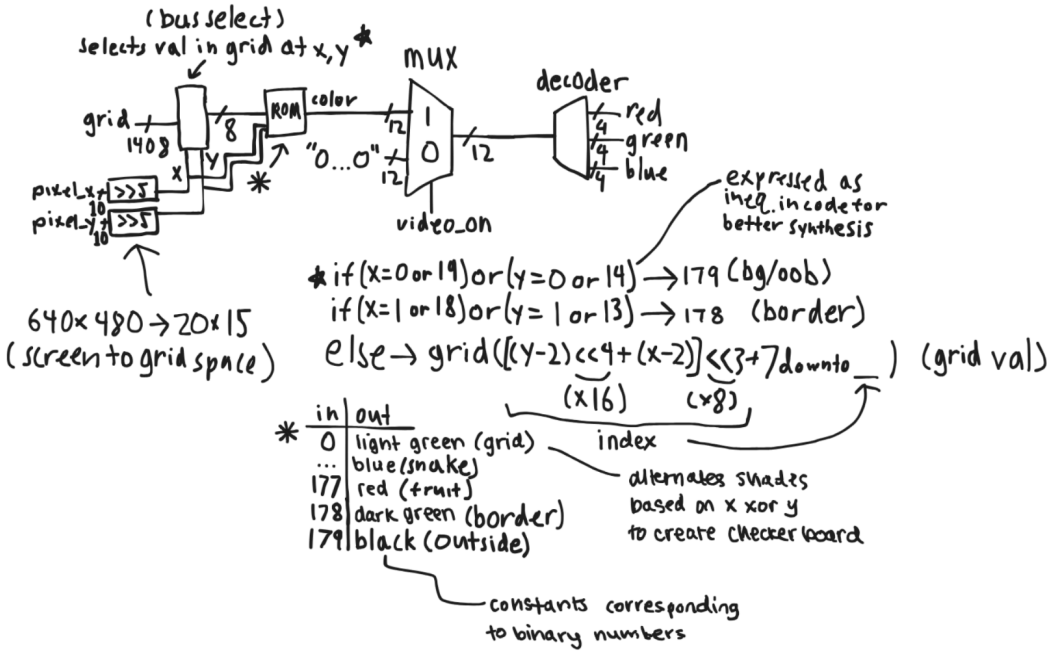
The pixel generation circuit is responsible for telling the VGA which color each pixel should be. It achieves this by sending three 4-bits values for red, green, and blue. Our pixel generation circuit does this by mapping pixel position to grid squares, determining the color based on the value at that square (or whether the pixel is even in the grid at all).

2.2.1. Description of Ports

Our inputs for this component are (1-2) pixel_x and pixel_y, a 10-bit vector that corresponds to the pixel we are generating color for, (3) video_on, which causes the circuit to output all zeroes, and (4) grid, which is the game grid that the circuit looks at when deciding what color to generate. Our pixel generation circuit does not use a clock because it has no registers. Our outputs are (1-3) red, green, and blue, which are 4-bit vectors containing the RGB color values that the current pixel should have.

2.2.2. Register Transfer Level Diagram

The pixel x and y inputs are converted into an index using math (notably, bitwise operators are used to perform multiplication), which is used to select an 8-bit value from the flattened grid. If the position being selected is not on the grid, then the selector outputs the value 178 if it is on the border and 179 if it is outside the border. The output of the selector is fed into a ROM, which maps the value to a 12-bit color constant. If its input is 0, then the output is light green (the shade of light green alternates according to x XOR y); if the input is 177, it is red (fruit); if the input is 178, it is dark green; if the input is 179, it is black; and if the input is any other value, then it is blue (snake). Before the color is decoded, we first check whether the video is on or not. If it is not, then we send “0…0” as the value to be decoded. Otherwise, we decode the color provided by the ROM into 4-bit red, green, and blue values by splitting the 12-bit color into thirds. The RTL diagram being described can be seen at the start of the next page.



2.2.3. Description of Memory

The only memory in the pixel generation circuit is a ROM that maps grid values to color constants. Its behavior is described in detail in the description of the RTL diagram, and is also shown as a table in the diagram. Because it is read only, there are no control signals.

2.3. LFSR Randomizer

The LFSR provides a random 8-bit number each clock cycle. This number is given to the datapath for use in randomly placing a fruit once it is eaten.

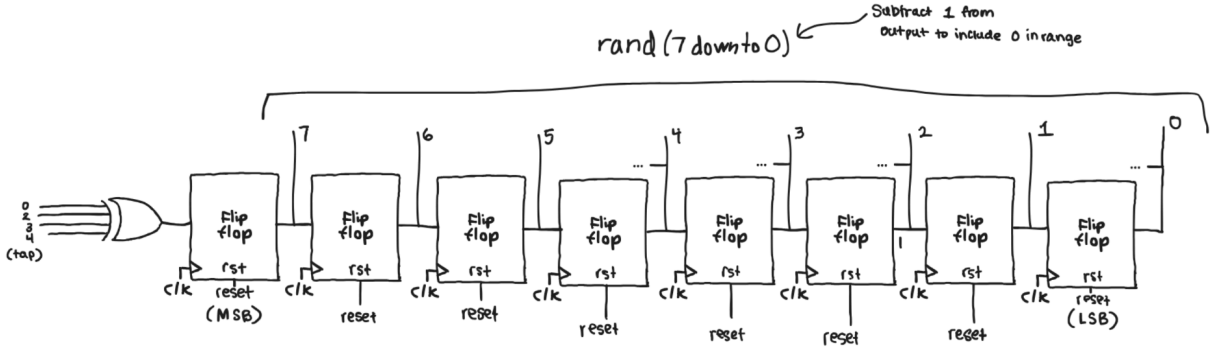
2.3.1. Description of Ports

Our input ports are simply (1) clk, a 100 MHz clock, and (2) reset, which resets the randomizer to its initial state when asserted. Our only output is rand, which is an 8-bit random number changing every clock cycle.

2.3.2. Register Transfer Level Diagram

We have a flip flop for each bit of the random number, with the most significant bit being the leftmost flip flop and the least significant being the rightmost. On the rising edge of the clock, the value of the 7th is shifted into the 6th, the value of the 6th into the 5th, and so on. The 0th value is dropped. The value going into the 7th flip flop is the XOR of all of the “taps” (bits 0, 2, 3, and 4), which represent what is known as the “feedback polynomial.” Some qualities of a feedback polynomial

are that the number of taps is even and that the GCD of the taps is 1. The purpose of a feedback polynomial is to ensure that the random numbers being generated are sufficiently random and cover all numbers in the range. Also, the seed (initial state) is "00000001" instead of "0…0" due to the fact that the XOR of zeroes is 0, meaning that the LFSR can never leave (and also never arrive at) a state of all zeroes. 1 is subtracted from the output to include 0 in its range because 0 is a valid fruit position on the grid. Not being able to generate "1…1" is not a concern due to the fact that the grid has less than 255 spaces.



2.3.3. Description of Memory

We use eight 1-bit flip-flops, one for each bit of the random number, all driven by the rising edge of the clock and capable of resetting synchronously. As mentioned previously, they reset to "00000001" instead of "0…0".

2.4. Snake Datapath

The datapath stores key information about the game, including the snake position, length, fruit position, and game grid. It also performs operations on this data based on the control signals it receives from the various FSMs. Note that because we have multiple FSMs whose outputs also affect much more than just the datapath, they are treated as standalone components and each have their own section rather than being included in this one.

2.4.1. Description of Ports

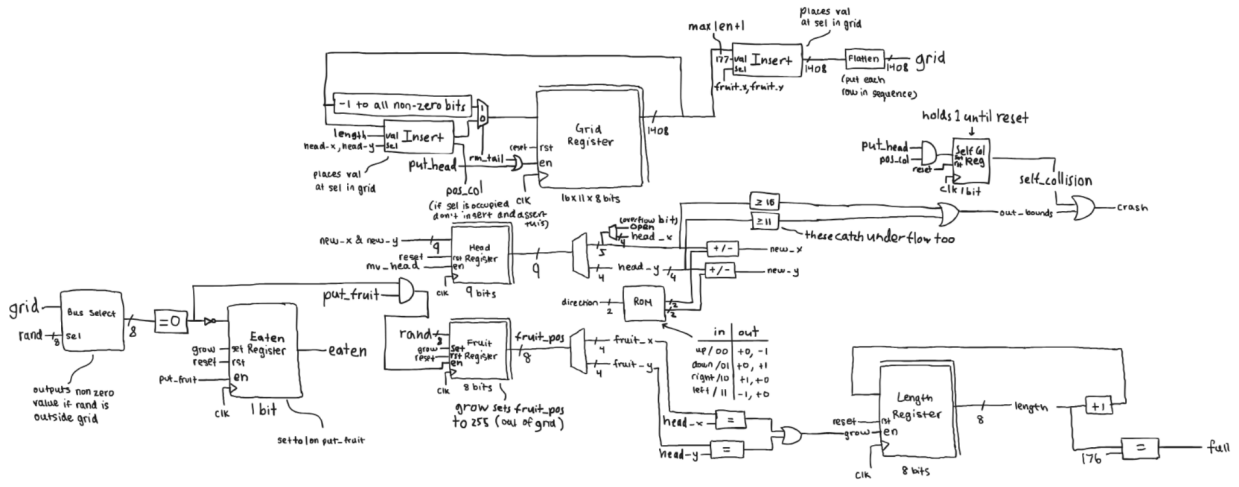
The datapath's input ports are (1) clk, the 100 MHz clock; (2) dir, a 2-bit vector encoding the direction (00 is up, 01 is down, 10 is right, and 11 is left); (3) rand, an 8-bit random number from the LFSR; (4) mv_head, a control signal telling the datapath to move the head position according to the direction; (5) rm_tail, another control signal telling the datapath to remove the tail of the snake; (6) put_head, which tells the datapath to actually place the head on the grid; (7) put_fruit, telling the datapath to attempt to find a new position for the fruit; and (8) reset, which

returns all memory in the datapath to an initial state. We also have the following outputs: (1) grid, a flattened 1408-bit representation of the game grid, with the fruit information added to it; (2) grow, a signal telling the FSM when the snake should grow so it can skip the remove tail state; (3) full, which tells the FSM that the game is over due to the grid being full; and (4) crash, which tells the FSM that the game is over due to the snake crashing into itself or the border.

2.4.2. Register Transfer Level Diagram

The 9-bit head register stores 5 bits for the x position of the snake head, and 4 bits for its y position. To move the head, a ROM decides based on the direction how to modify the x and y values (+1, +0, or -1), which are fed back into the register. The value of the register is checked using comparators to ensure that the head is within the bounds of the grid. If not, then crash is asserted (though this is not the only thing that drives the crash signal). Additionally, the value of the head register is checked against the value of the fruit register, and the grow signal is asserted for exactly one clock cycle when this happens, before the fruit is moved out of the grid temporarily. On the grow signal, the length counter increments, and the full signal is asserted if the snake length is equal to the max size of the grid. The grow signal also sets the value in the eaten register to 1, which is sent to the FSM. The FSM will determine whether a fruit needs to be placed or not based on the eaten signal. If it does, then it will tell the datapath to place a fruit. When this happens, if the random number from the LFSR happens to be within grid bounds and at an empty square (determined with a bus select), then the fruit register will store that position and the eaten register will store 0, telling the FSM to stop retrying.

The 16 x 11 x 8 grid register is also located in the datapath, and when it is told to remove the tail, will decrement each non-zero value in the grid by 1. When it is told to place the snake head, it will insert a value equal to the length into the current head position in the grid if that square is empty. If not, then the self collision register is set to 1 until reset, which causes crash to be asserted (this value OR whether the head is out of bounds drive crash). Finally, the grid is flattened before being outputted, meaning that each row is placed in sequence in a 1408-bit vector. The fruit position is also placed on the grid with the value 177 (one greater than the max length of the snake) before this. The RTL diagram of our Snake datapath is located at the start of the next page.



2.4.3. Description of Memory

This component contains a 1408-bit (16 x 11 x 8) register for the grid, a 9-bit register for the head position, an 8-bit register for the fruit position, an 8-bit register for the length of the snake, a 1-bit eaten register, and a 1-bit self collision register. These all update on the rising edge of the clock and reset synchronously. There is also a ROM that takes in the 2-bit direction vector whose output tells the adders whether to add, subtract, or leave the value of the head the same.

2.5. Datapath Helper FSM

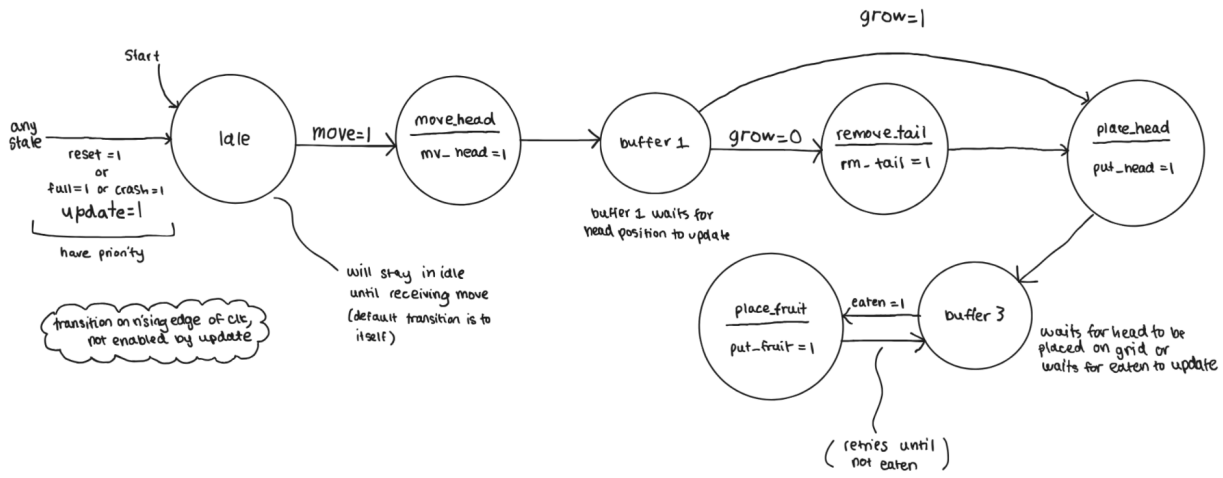
The datapath helper FSM communicates with the datapath to coordinate operations such as moving the snake. Among all the FSMs, it is the most specialized, but is still complex enough to warrant a distinct section in our report.

2.5.1. Description of Ports

The inputs to this component are: (1) clk, the 100 MHz clock; (2) update, the 2 Hz update pulse; (3) move, a control signal from another FSM indicating whether the snake should move this frame or not; (4) full, which indicates whether the grid is full or not; (5) crash, asserted when the snake has crashed; (6) grow, whether the snake should grow in length; (7) eaten, whether the snake has eaten the fruit and a new one needs to be placed; and (8) reset, whether to return the component to its original state or not. The outputs are: (1) mv_head, which tells the datapath to move the head position of the snake; (2) rm_tail, telling the datapath to remove the tail of the snake; (3) put_head, telling the datapath to place the snake's head on the grid; and (4) put_fruit, telling the datapath to attempt to place a fruit. The output signals should only be asserted for 1 clock cycle.

2.5.2. Finite State Machine Diagram

The helper FSM starts in idle. When it receives a move signal, it transitions to a state that tells the datapath to move the snake's head. There is a single buffer to ensure the head position and grow signal have been updated. If grow is high (a fruit was eaten), the remove tail state is skipped and the FSM transitions directly to the place head state. These states have outputs that instruct that datapath to perform the corresponding operation. Then, the FSM transitions to a buffer state where it waits for the eaten signal to update. If eaten is high, the put fruit signal is asserted in the next state before the FSM transitions back to the buffer (as the put fruit operation can require more than one attempt). If reset, full, crash, or update are asserted at any point, the FSM is sent back to idle. In the case of reset and update, the FSM will simply cycle through its states again, but for full and crash, the FSM will remain in the idle state until the next reset (this is not a result of any specific part of the FSM but rather conventions for the signals).



2.5.3. Description of Memory

Our state machine has 7 states, meaning that it needs two 3-bit registers to store the current and next states.

2.6. Direction FSM

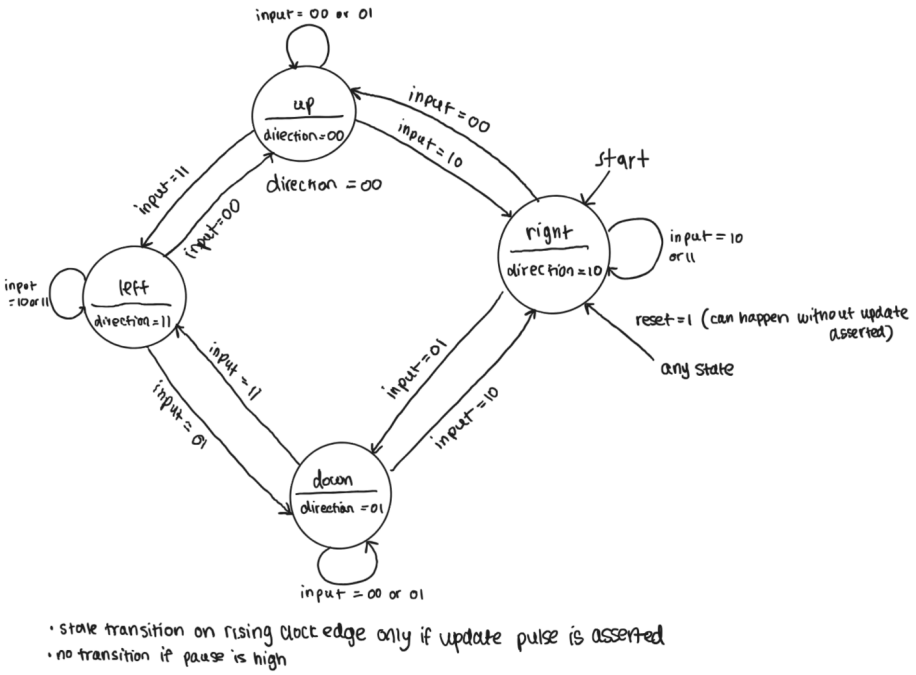
The direction FSM receives the debounced inputs for the direction buttons from the input conditioner and decides which direction the snake is facing. It ensures that invalid inputs are ignored and transitions based on valid inputs (as the game Snake has restrictions on how direction can be changed). This FSM is also important to the datapath, but is treated as a distinct component in our report.

2.6.1. Description of Ports

Our inputs are (1) clk (100 MHz); (2) update (2 Hz pulse); (3) input, a 2-bit vector representing the 2-bit direction input from the user (“00”, “01”, “10”, and “11” for up, down, left, and right respectively); (4) pause, whether the game is paused; and (5) reset, whether to reset the FSM. Our only output is a 2-bit vector for direction (same convention as input).

2.6.2. Finite State Machine Diagram

The direction FSM starts in the right state. Transitions only occur to states whose directions are perpendicular to the current direction. If the input is opposite to the current direction, the FSM remains in the same state (it also holds when the input is the same). For example, from the up state, the FSM can transition to left or right and will remain in the up state if up or down inputs are received. Each state outputs the corresponding 2-bit vector for direction. At any state, if reset is high, the FSM will transition to the right state. It can do this independent of the update pulse, but is still synchronous in that it happens on the rising edge of the clock. No transitions can occur if pause is asserted, except for reset.



2.6.3. Description of Memory

Our state machine has 4 states, meaning that it needs two 2-bit registers to store the current and next states.

2.7. Game State FSM

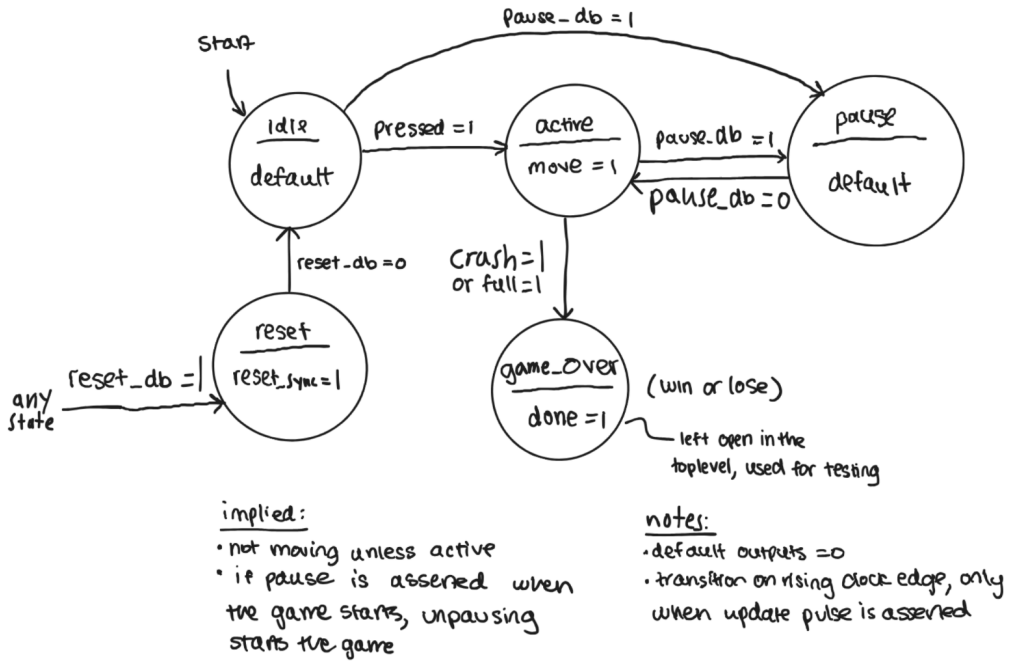
The game state FSM determines what state the game is in (e.g., active, paused) based on both user input and signals from the datapath.

2.7.1. Description of Ports

The inputs are (1) clk (100 MHz); (2) update (25 MHz pulse); (3) pressed, high once a button has been pressed; (4) full, indicating when the grid is full; (5) crash, indicating whether the snake has crashed; (and 6-7) pause_db and reset_db, the debounced pause and reset inputs. The outputs are (1) reset_sync, which resets all other components; (2) move, indicating that the snake should move; and (3) done, which signals that the game is over (only used for testing).

2.7.2. Finite State Machine Diagram

The game state FSM starts in an idle state and transitions to active when pressed is asserted. When active, it is correct for the snake to move, so the corresponding output is asserted by the FSM. If crash or full is asserted, the FSM transitions to a game over state which is held until reset. If reset is asserted from any state, the FSM transitions to a reset state and remains there until reset is unasserted (it then enters the idle state). Similarly, if pause is asserted (only from idle and active this time), the FSM transitions to the pause state until pause goes low (it then enters the active state).



2.7.3. Description of Memory

Our state machine has 5 states, meaning that it needs two 3-bit registers to store the current and next states.

2.8. Input Conditioner

This component was taken from lab 5 in the course, and was not designed specifically for this project. It synchronizes the raw input given through the button port using a 2 flip-flop synchronizer and sends debounced and monopulsed signals as outputs. Because we did not design the component for this project, we omit subsections for this component.

3. Design Validation

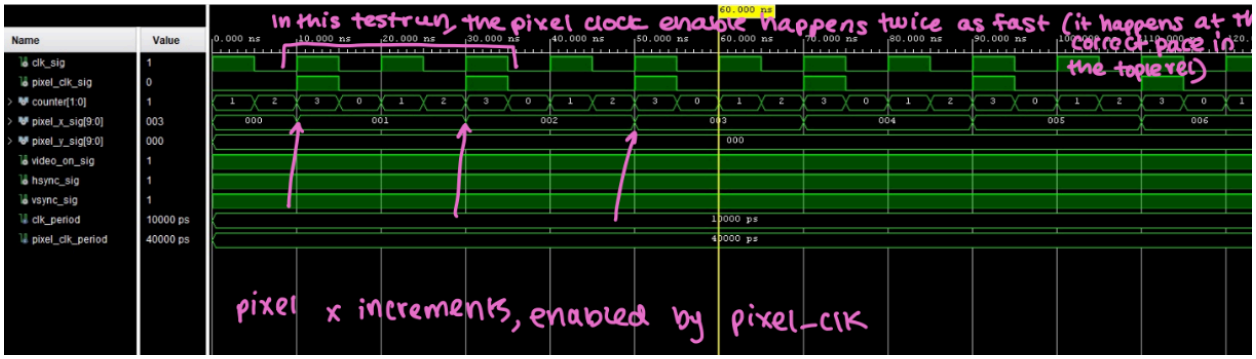
This section contains information about our tests for each component.

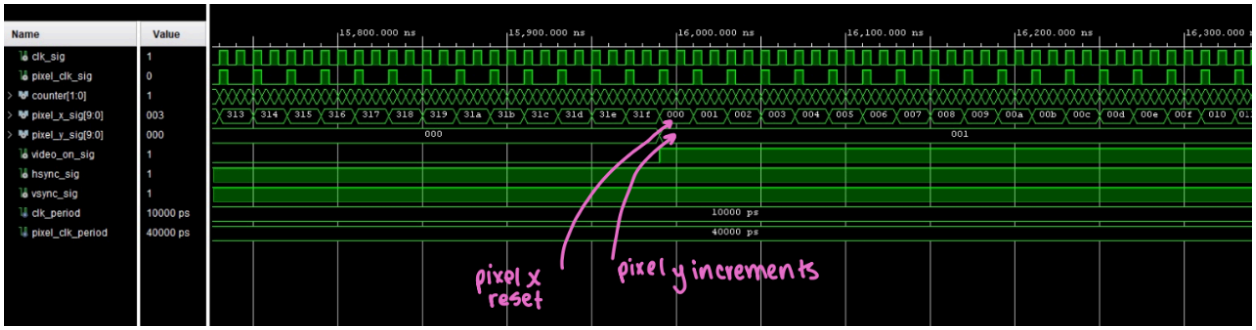
3.1. VGA Synchronizer

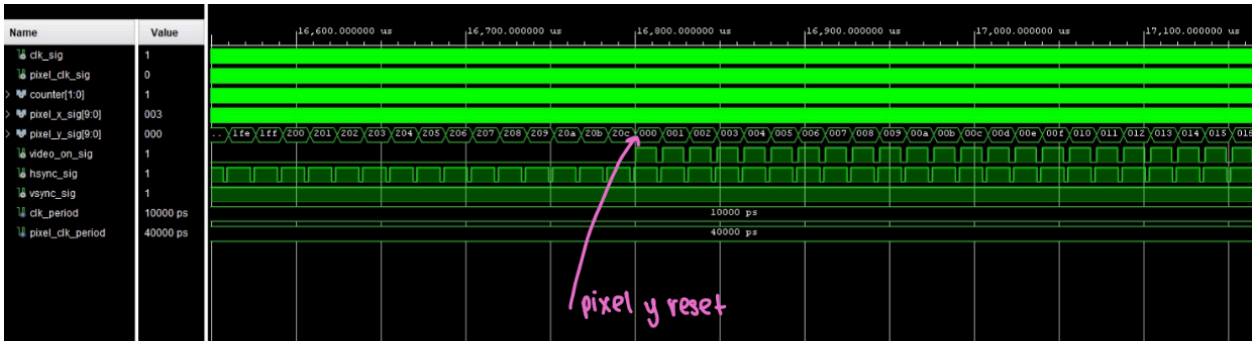
First, we generated a 100 MHz clock and a 25 MHz pulse (clock generation is common to all of our tests and will be omitted from now on). Then, we observed the outputs of the VGA synchronizer given these inputs. We expect pixel_x to increment every time the 25 MHz pulse (pixel_en) goes high and reset after 799, and we expect pixel_y to increment at the total count of pixel_x and reset after 524. Additionally, we expect video_on when pixel_x is less than 640 and pixel_y is less than 480. We also expect hsync and vsync to be low within certain ranges and high otherwise (according to the specification).

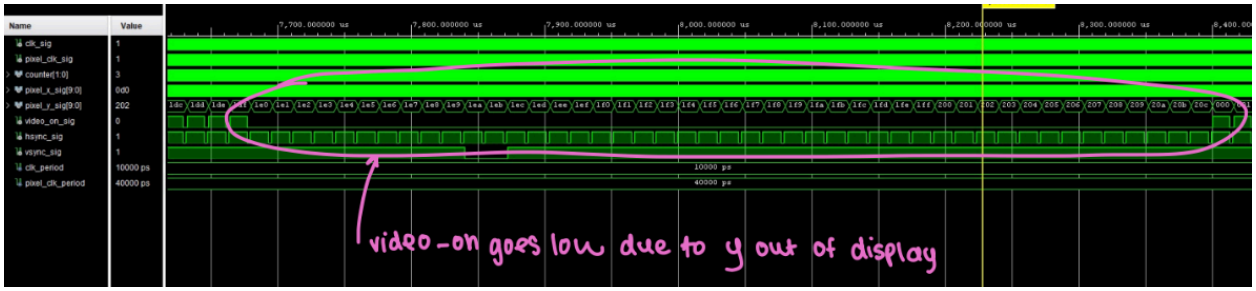
3.1.1. Behavioral Simulations

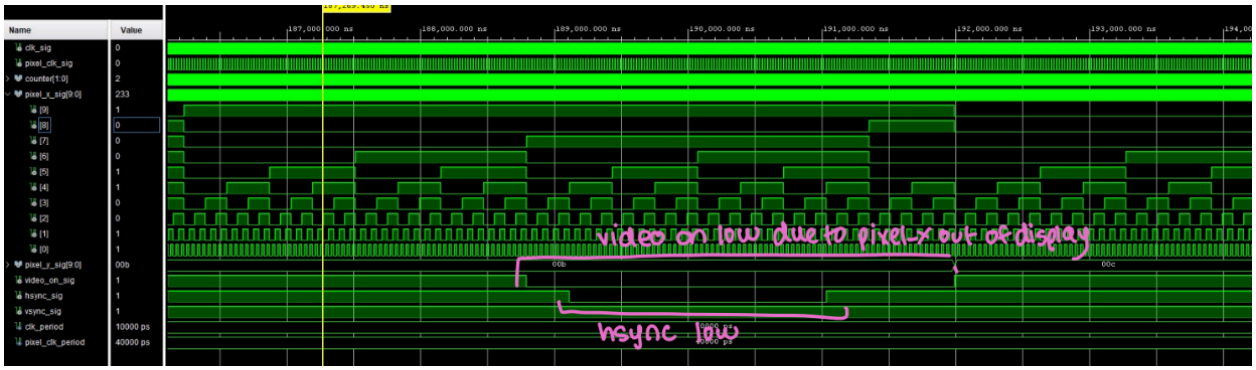
We needed a total of 6 screenshots to capture the test result.




pixel x reset
pixel y increments


pixel y reset


video_on goes low due to y out of display


video on low due to pixel_x out of display
hsync low


vsync low

3.1.2. Hardware Validation

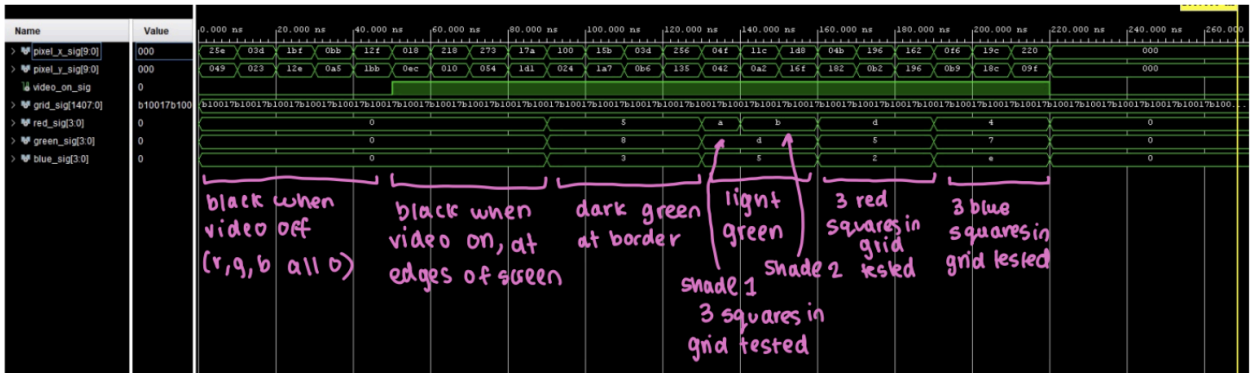
We used the provided test pattern along with our VGA synchronizer to test that it was working correctly.

3.2. Pixel Generation Circuit

First, we used a for loop to generate a grid of alternating colors. The pattern we chose is light green (empty square in grid), then red (fruit), then blue (snake). This repeats every three squares in the grid, except that the shade of light green alternates as well (this will create a checkerboard pattern in the actual game). Note that this is difficult to notice in the hardware validation due to how similar the shades are. We also expect that the border is dark green and outside the border is black, though this is not part of the grid. For the simulation, we sampled values of each color, using precomputed random numbers within calculated ranges to be thorough. We sampled 5 black squares when the video was off, 4 black squares outside of the grid, 4 dark green squares on each side of the border, and 3 of each other color at various points within the grid.

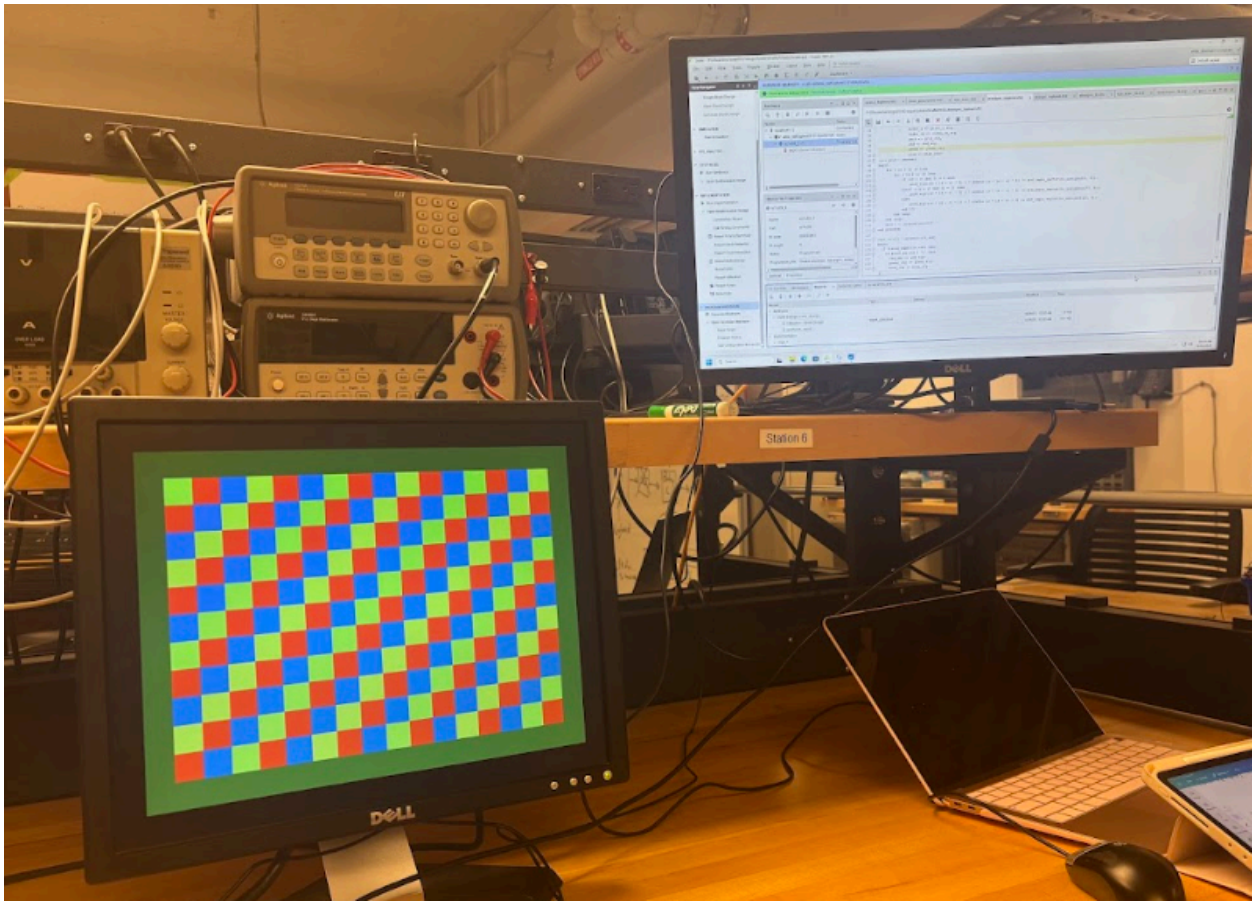
3.2.1. Behavioral Simulations

We only needed one screenshot for this component.



3.2.2. Hardware Validation

We tested pixel generation using the VGA synchronizer.

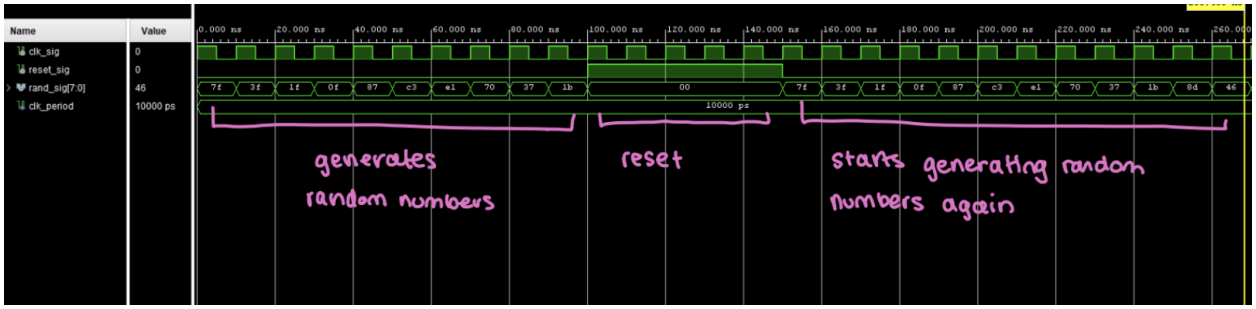


3.3. LFSR Randomizer

We ran the randomizer for 10 clock cycles, expecting that the numbers appear sufficiently random. Then, we tested reset, ensuring that the LFSR returned to its initial state. After, we expect that the randomization continues, cycling through the same random numbers because an LFSR is deterministic.

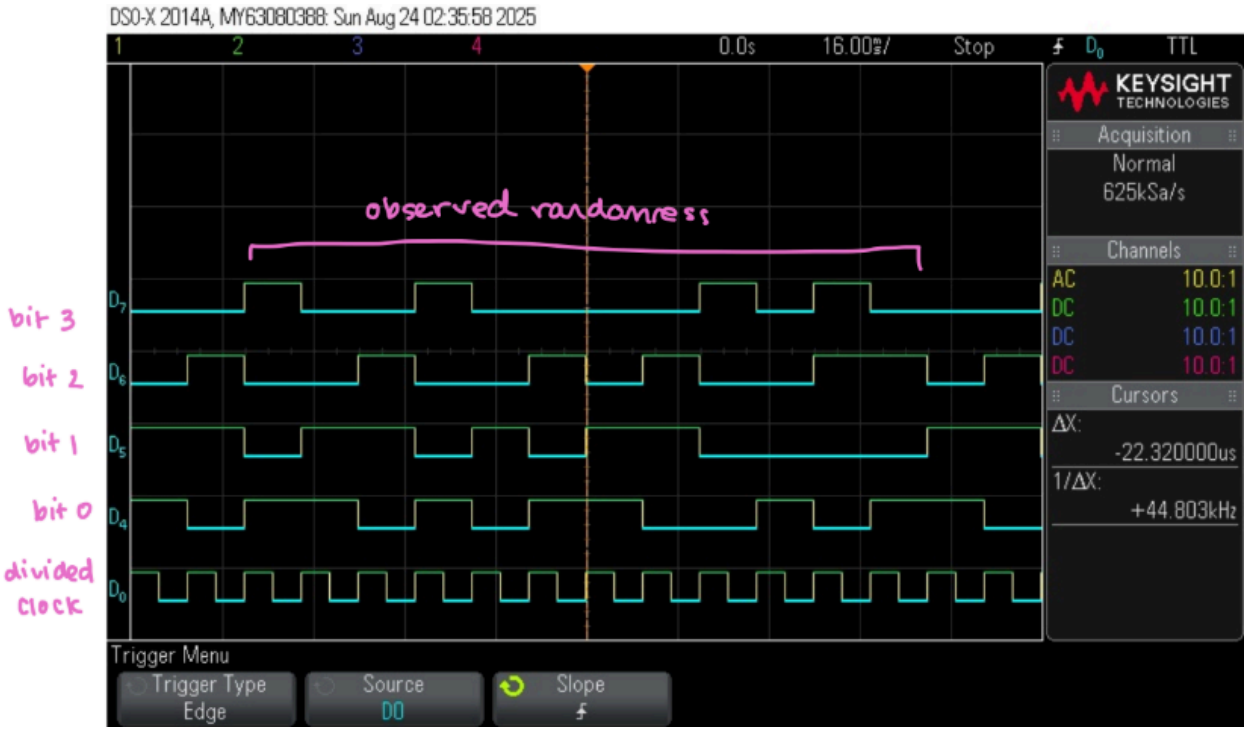
3.3.1. Behavioral Simulations

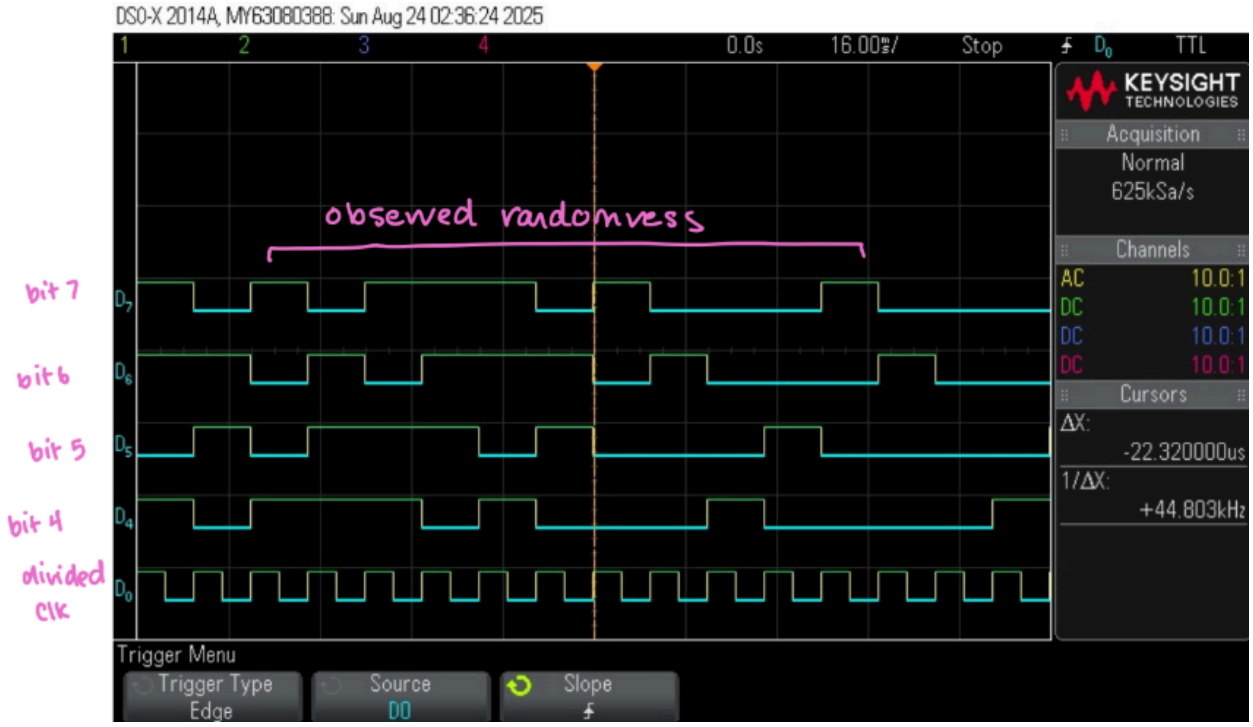
We only needed one screenshot for this component.

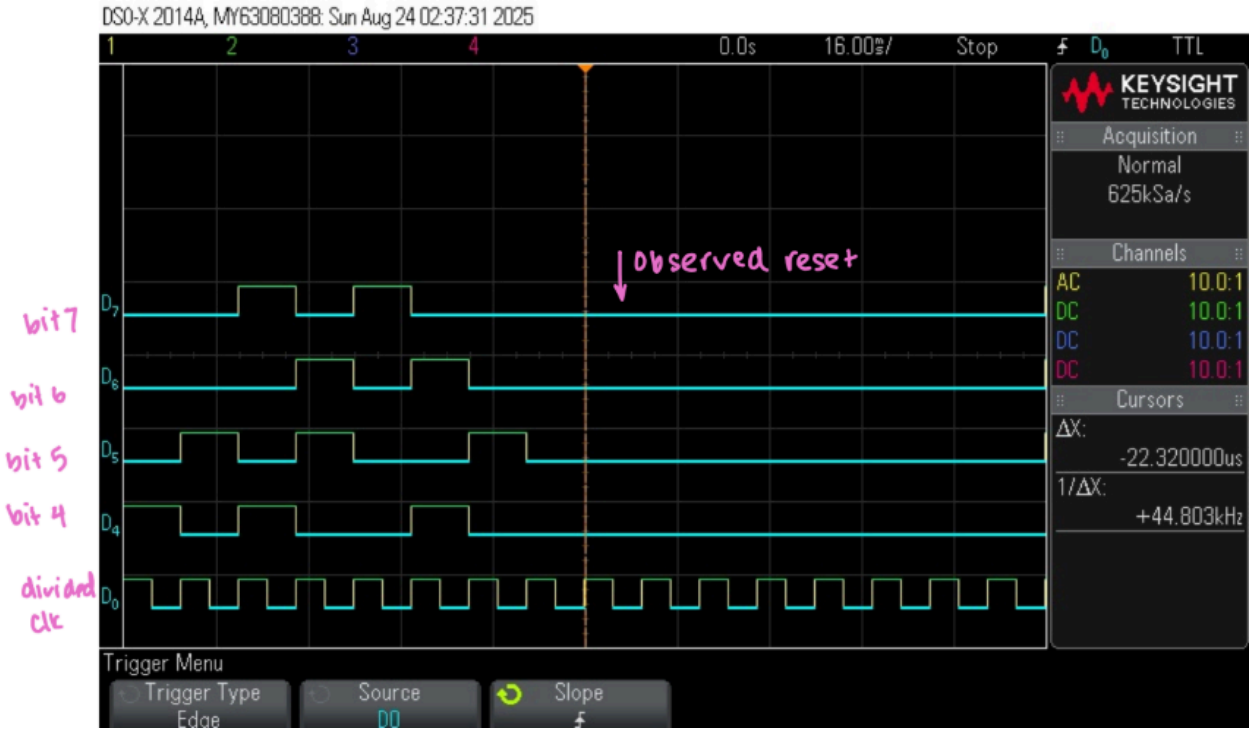


3.3.2. Hardware Validation

We had to take multiple pictures on the oscilloscope to capture everything.







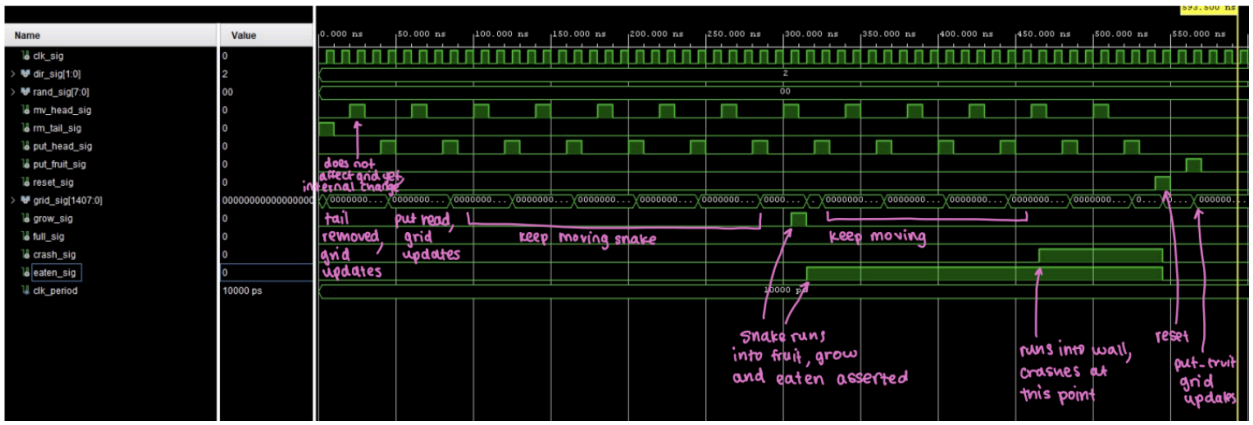
3.4. Snake Datapath

Because of the difficulty of interpreting a 1408-bit flattened grid signal, we conducted a simpler behavioral simulation where we asserted control signals and checked that the grid changed when expected (rather than looking at its actual value). Our hardware validation

is able to more thoroughly test behavior due to the fact that we can observe the grid on an actual display. For testing, we check that each datapath operation works as intended.

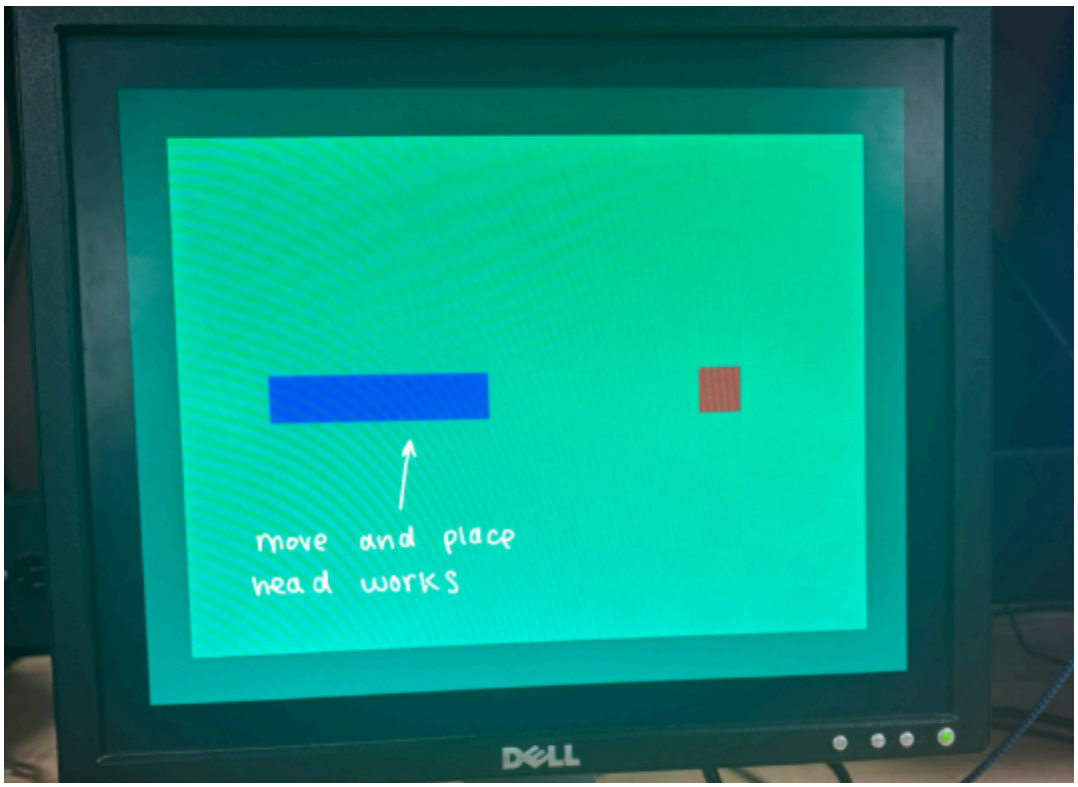
3.4.1. Behavioral Simulations

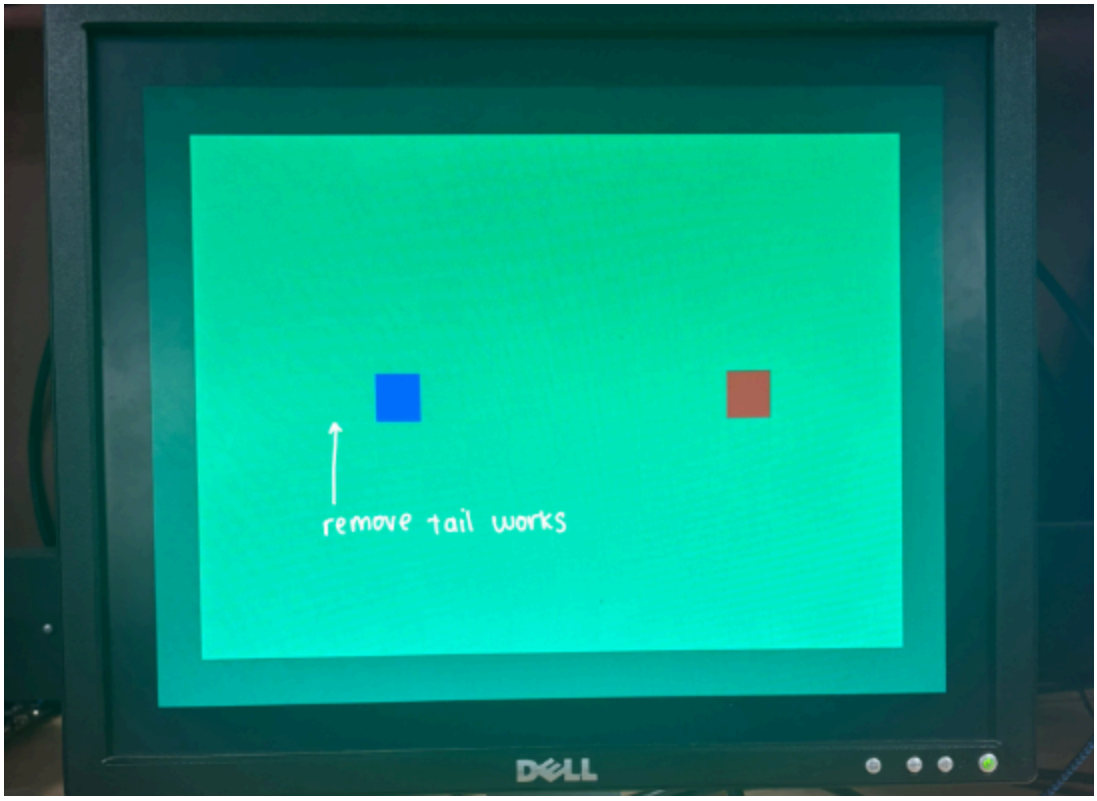
The results of the simulation are depicted below.



3.4.2. Hardware Validation

We mapped each input in the datapath to physical input devices and the output to the VGA using the previous components.




remove tail works
DELL

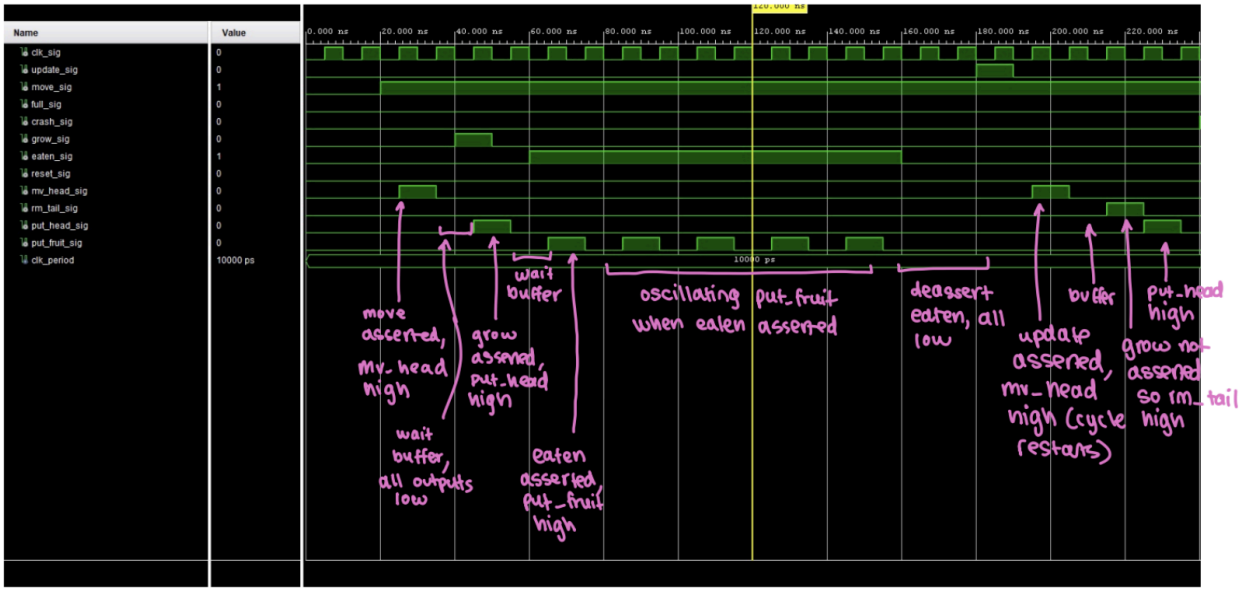

place fruit works
DELL

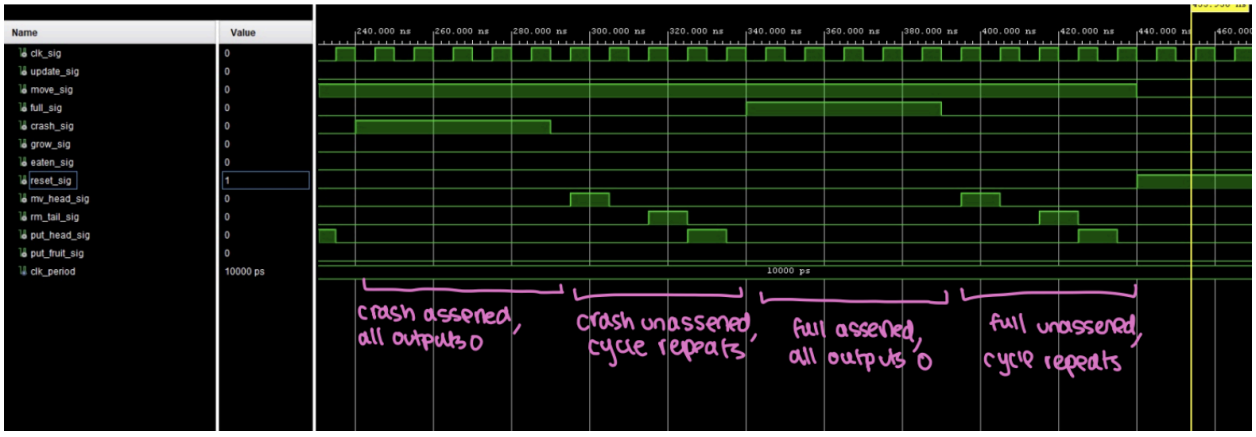
3.5. Datapath Helper FSM

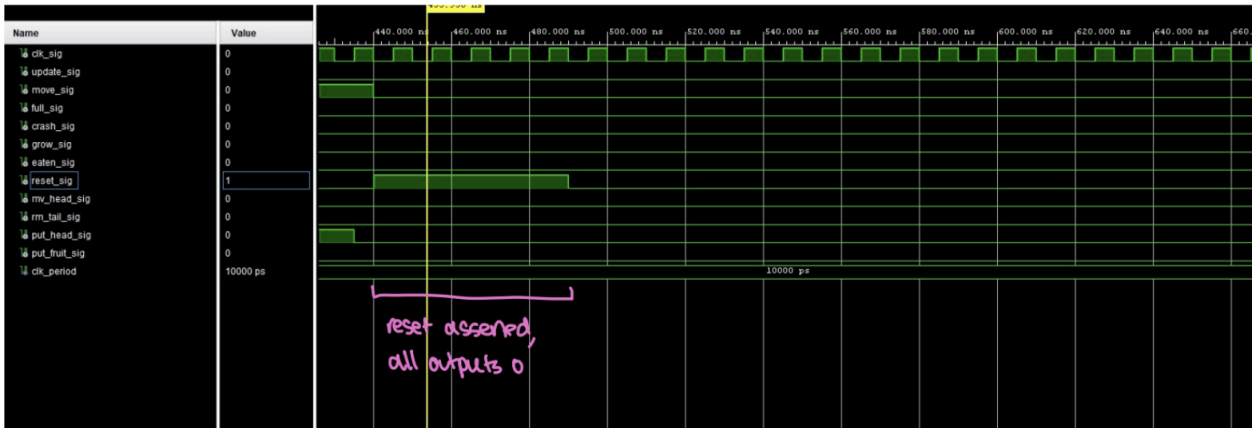
For our testing strategy, we assert move and expect mv_head to go high. We wait one clock cycle for the buffer state, expecting everything to be low. Then, we assert grow to skip the tail removal state, expecting put_head to go high. We wait for another buffer, expecting everything to be low again. Then, we assert the eaten signal for multiple clock cycles, expecting the put_fruit signal to oscillate. When we deassert eaten, we expect all outputs to be low. After asserting the update signal, we expect the cycle to restart again (this time, with rm_tail going high because we don't assert grow). We also test that all signals are low whenever the crash, full, and reset signals go high, and that the FSM will continue to cycle through its states when these signals go from high to low.

3.5.1. Behavioral Simulations

The three screenshots below capture our simulation.

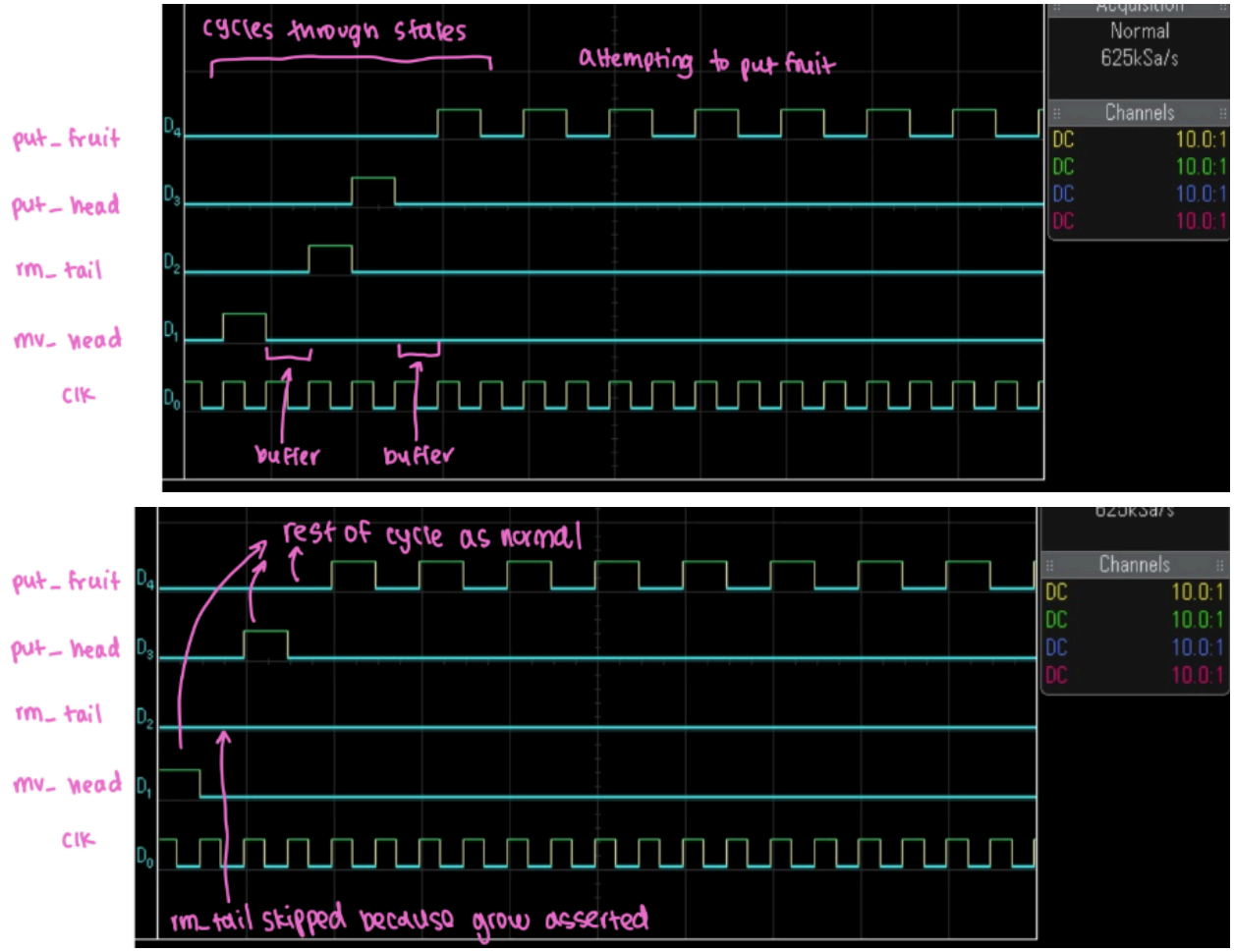






3.5.2. Hardware Validation

We took two pictures of the oscilloscope for hardware validation.

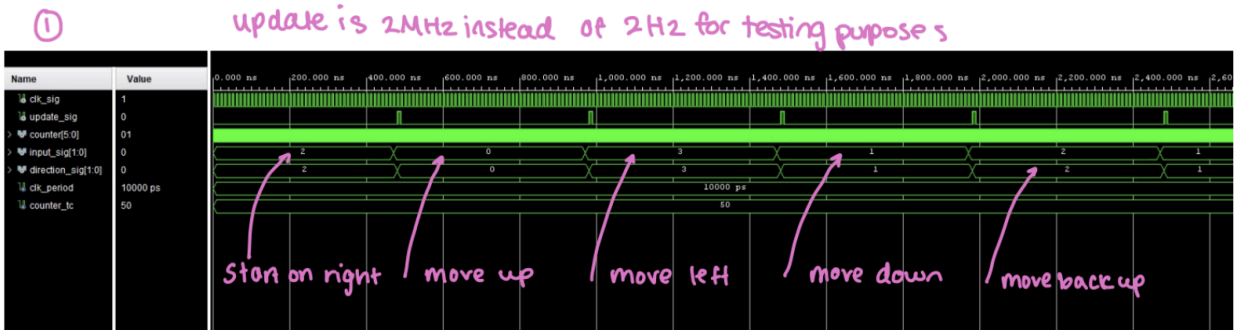


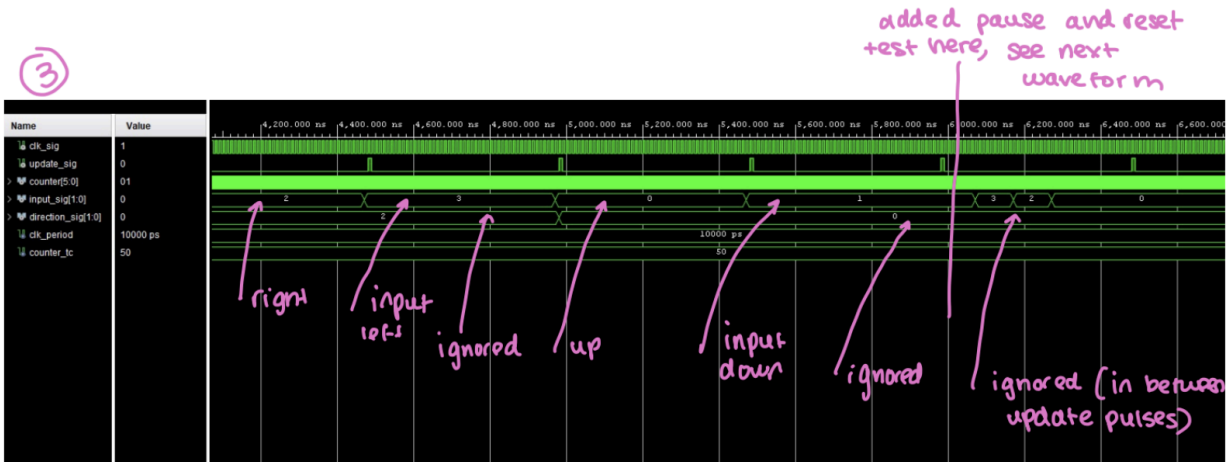
3.6. Direction FSM

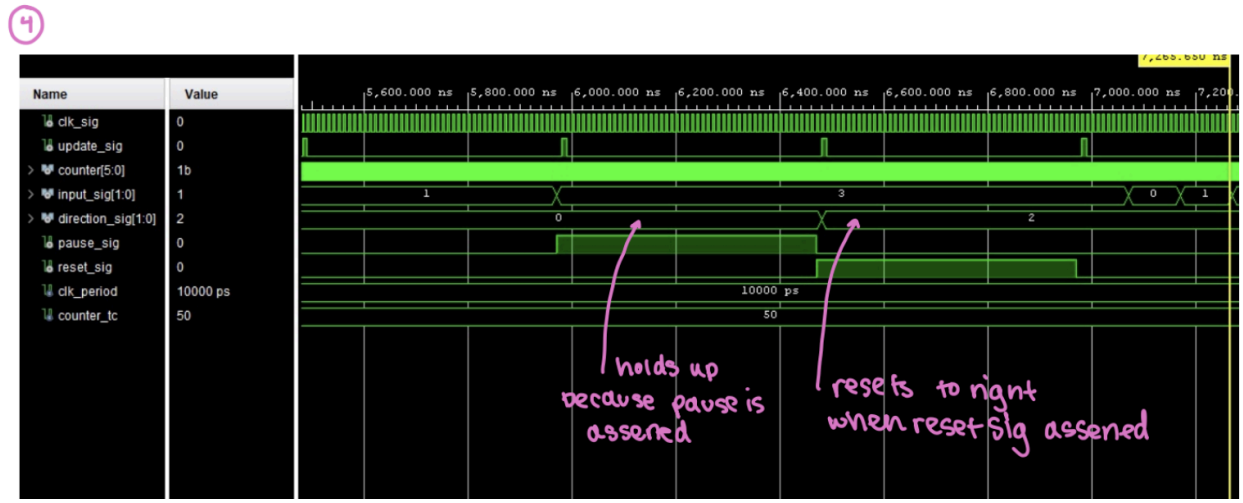
We tested that our FSM is capable of making transitions on valid inputs, and does not respond to invalid inputs. For example, we expect that the FSM starts at right and can go down, left, and then up (this cycle should also be valid in reverse). We expect the FSM to ignore inputs opposite and identical to the current direction. We also expect that the FSM does not change unless update is asserted, and that reset returns the direction to right and that pause holds the last direction.

3.6.1. Behavioral Simulations

We took four screenshots to capture our behavioral simulation.

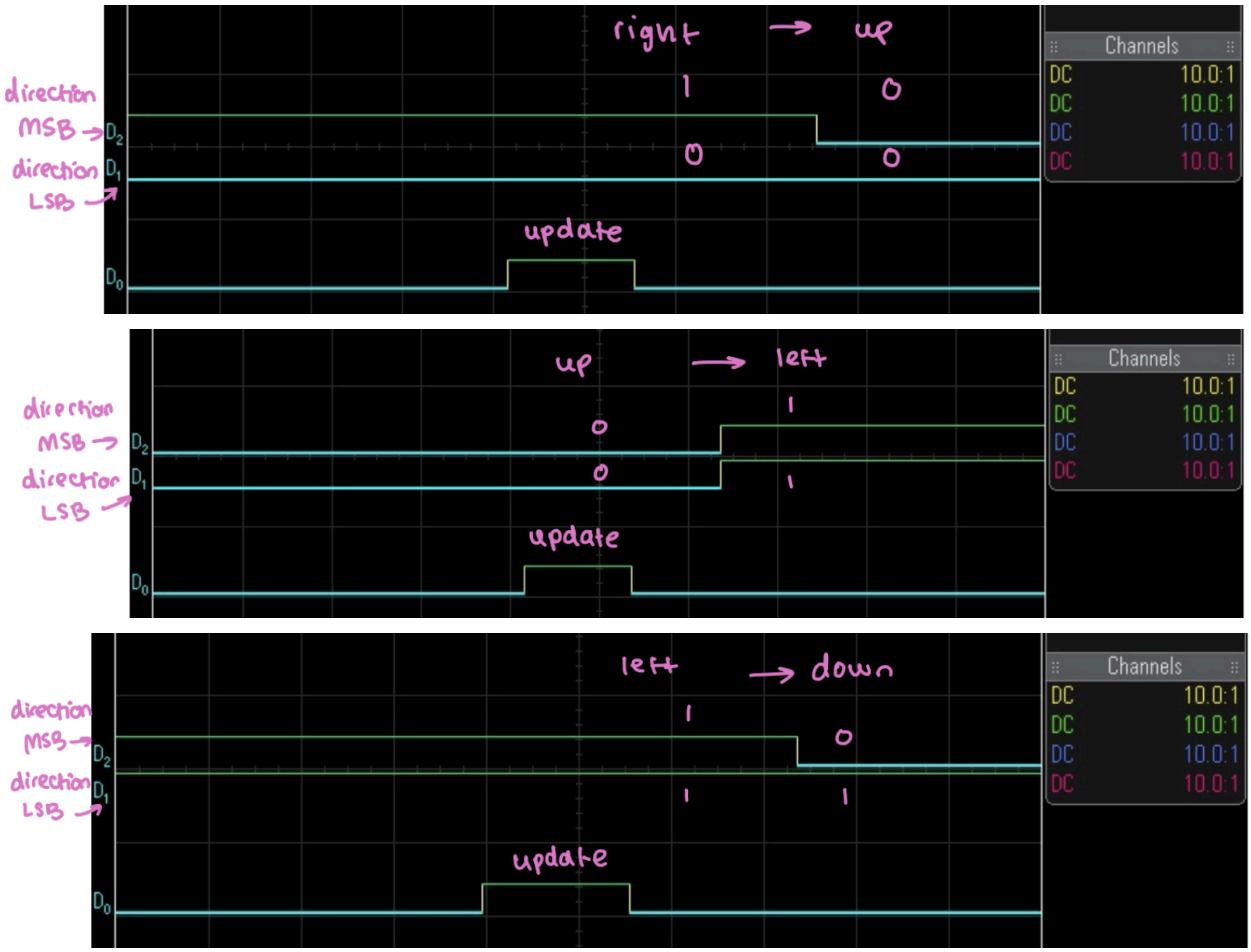


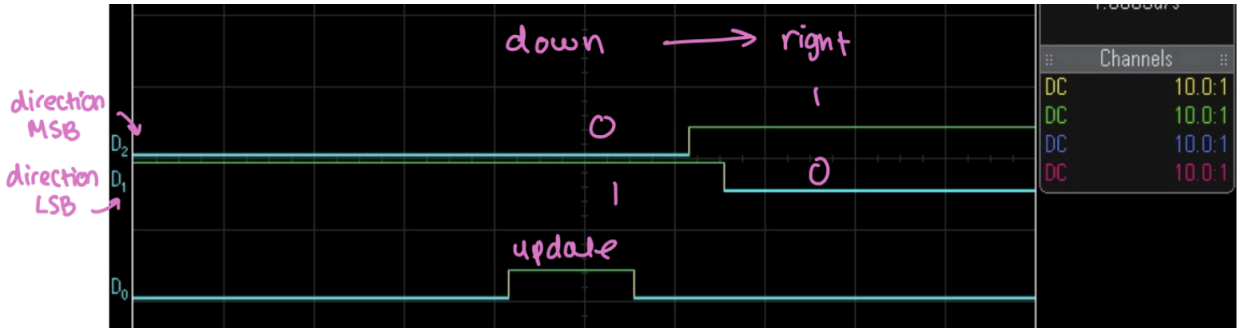




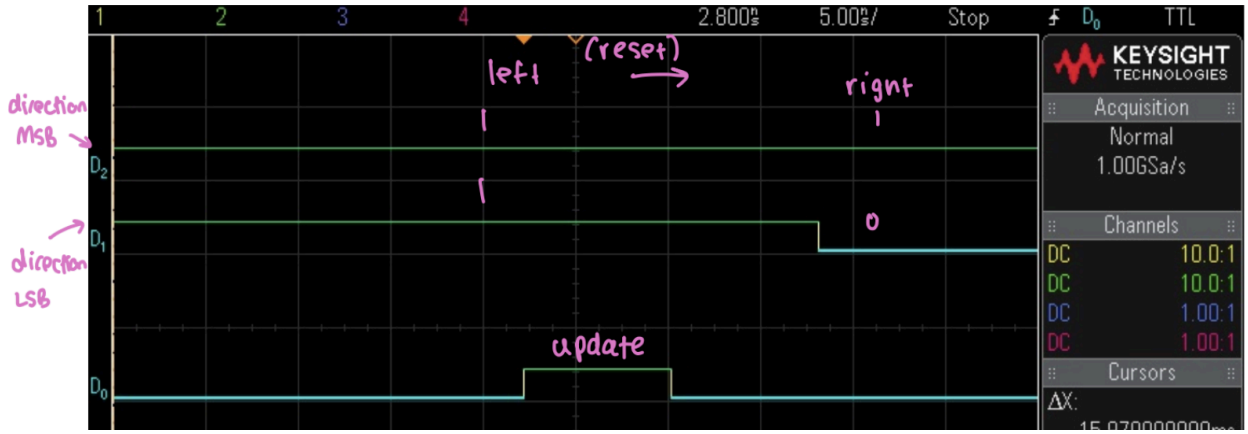
3.6.2. Hardware Validation

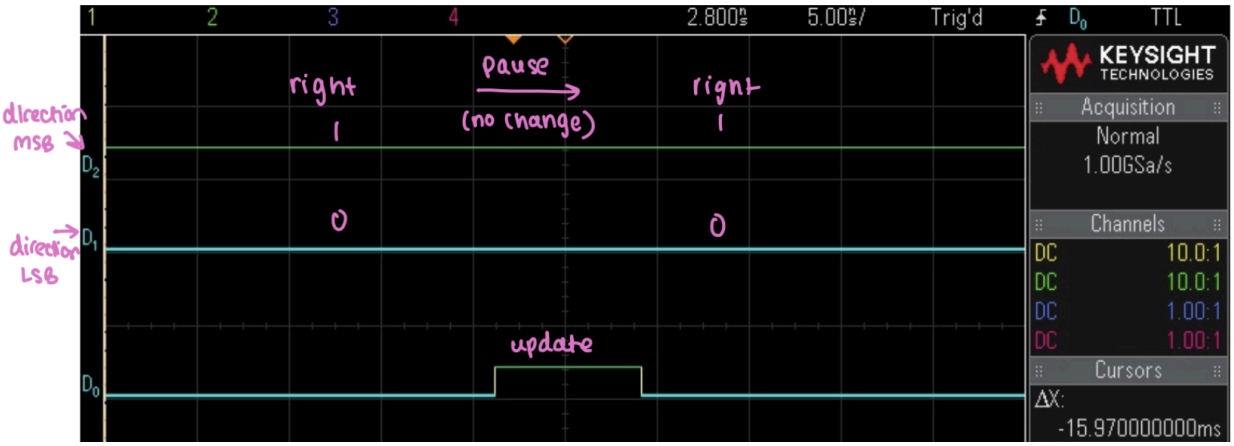
We needed seven screenshots to capture our testing on the oscilloscope.




down
right
1
0
1
0
direction MSB
direction LSB
update
D_2
D_1
D_0
Channels
DC 10.0:1
DC 10.0:1
DC 10.0:1
DC 10.0:1


does not respond to left
right
right
1
1
0
0
direction MSB
direction LSB
update
D_2
D_1
D_0
Averaging: 2
1.00GSa/s
Channels
DC 10.0:1
DC 10.0:1
DC 10.0:1
DC 10.0:1


1
2
3
4
2.800ns
5.00ns/
Stop
D_0 TTL
left
(reset)
right
1
1
1
0
direction MSB
direction LSB
update
D_2
D_1
D_0
KEYSIGHT TECHNOLOGIES
Acquisition
Normal
1.00GSa/s
Channels
DC 10.0:1
DC 10.0:1
DC 1.00:1
DC 1.00:1
Cursors
ΔX:

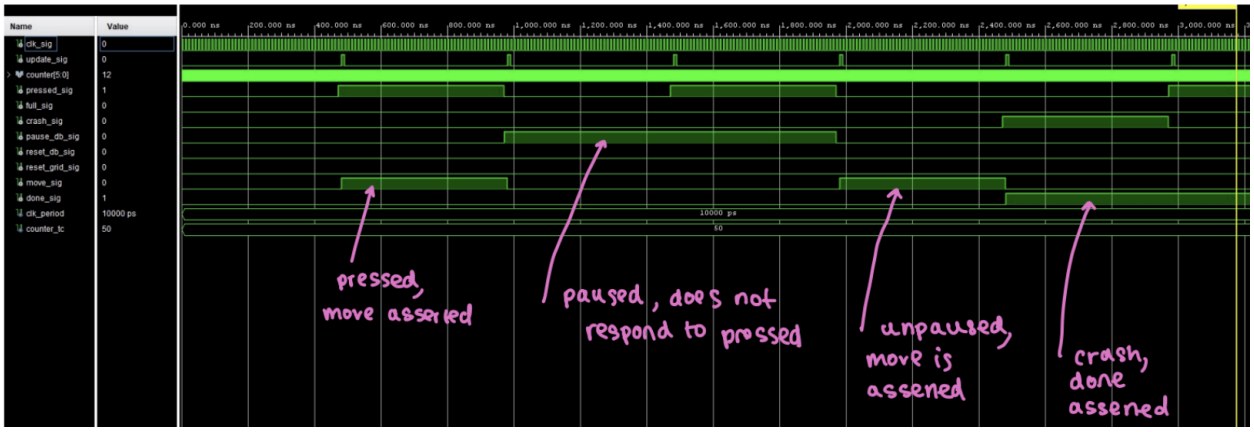

1
2
3
4
2.800ns
5.00ns/
Trig'd
D_0 TTL
right
pause
(no change)
right
1
1
0
0
direction MSB
direction LSB
update
D_2
D_1
D_0
KEYSIGHT TECHNOLOGIES
Acquisition
Normal
1.00GSa/s
Channels
DC 10.0:1
DC 10.0:1
DC 1.00:1
DC 1.00:1
Cursors
ΔX:
-15.970000000ms

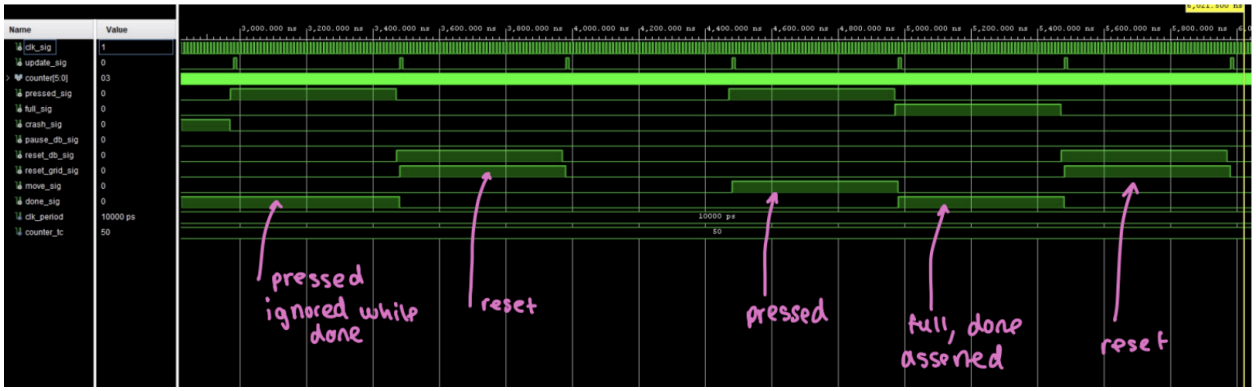
3.7. Game State FSM

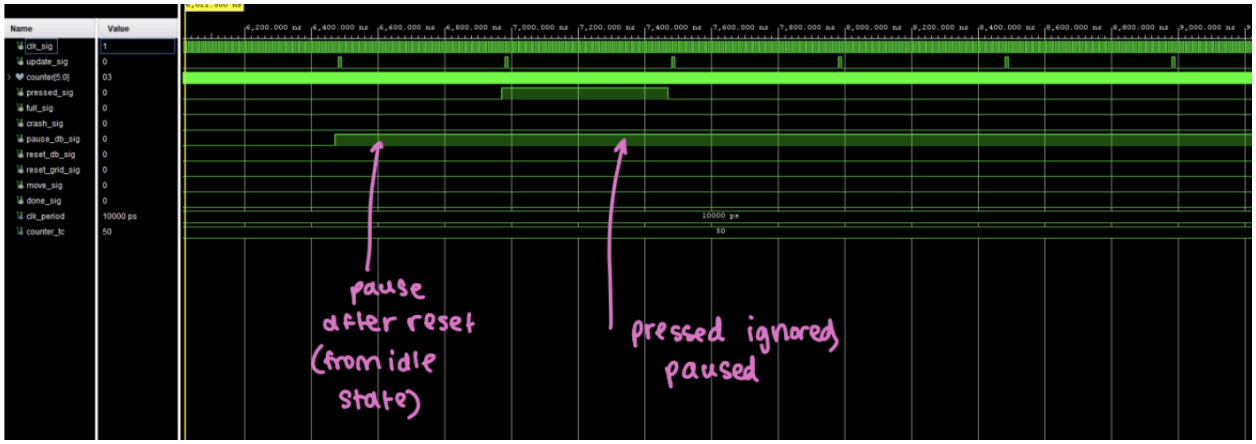
We test that when pressed is asserted, we expect the move signal to go high (indicating active state). When pause_db is asserted, even after pressed is asserted, we expect move to stay low (indicating pause state). When pause_db is low, we expect move to go high (returning to active). Whenever crash is asserted, we expect done to go high (indicating game over), and stay that way even when pressed is asserted. When reset_db is asserted, then reset_sync should go high. After, pressed should send the FSM back to active. Full, similar to crash, should also cause done to go high. Additionally, it is possible to go to the active state by asserting and deasserting pause_db from idle.

3.7.1. Behavioral Simulations

Our simulation is captured in the following three screenshots.

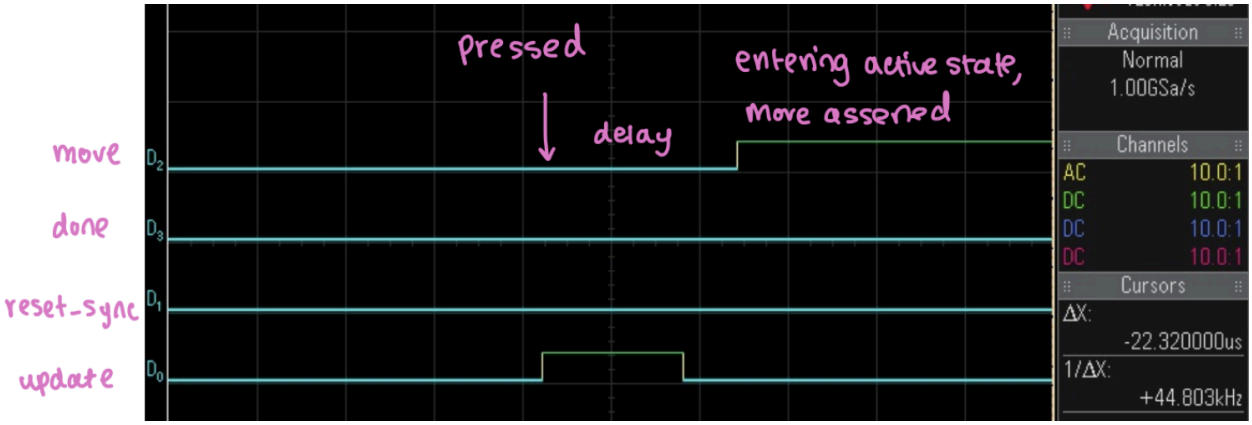


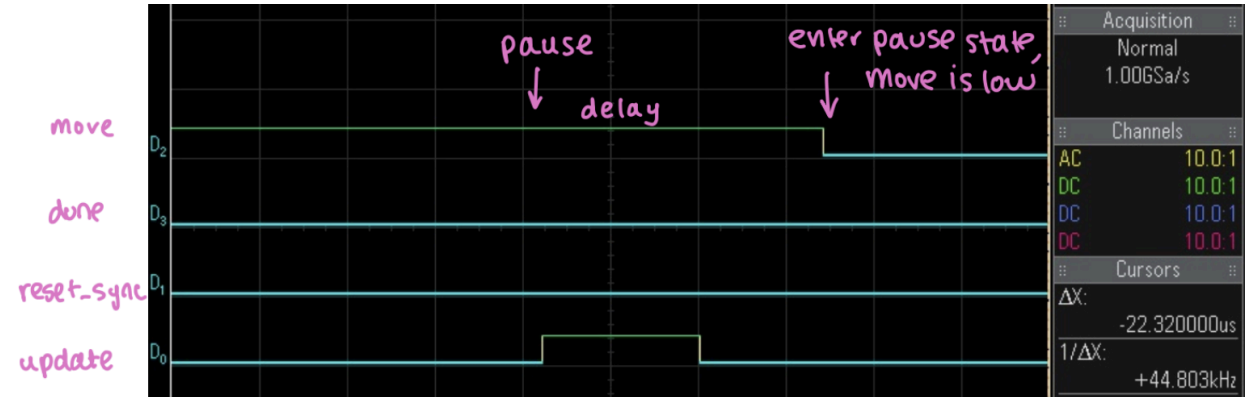


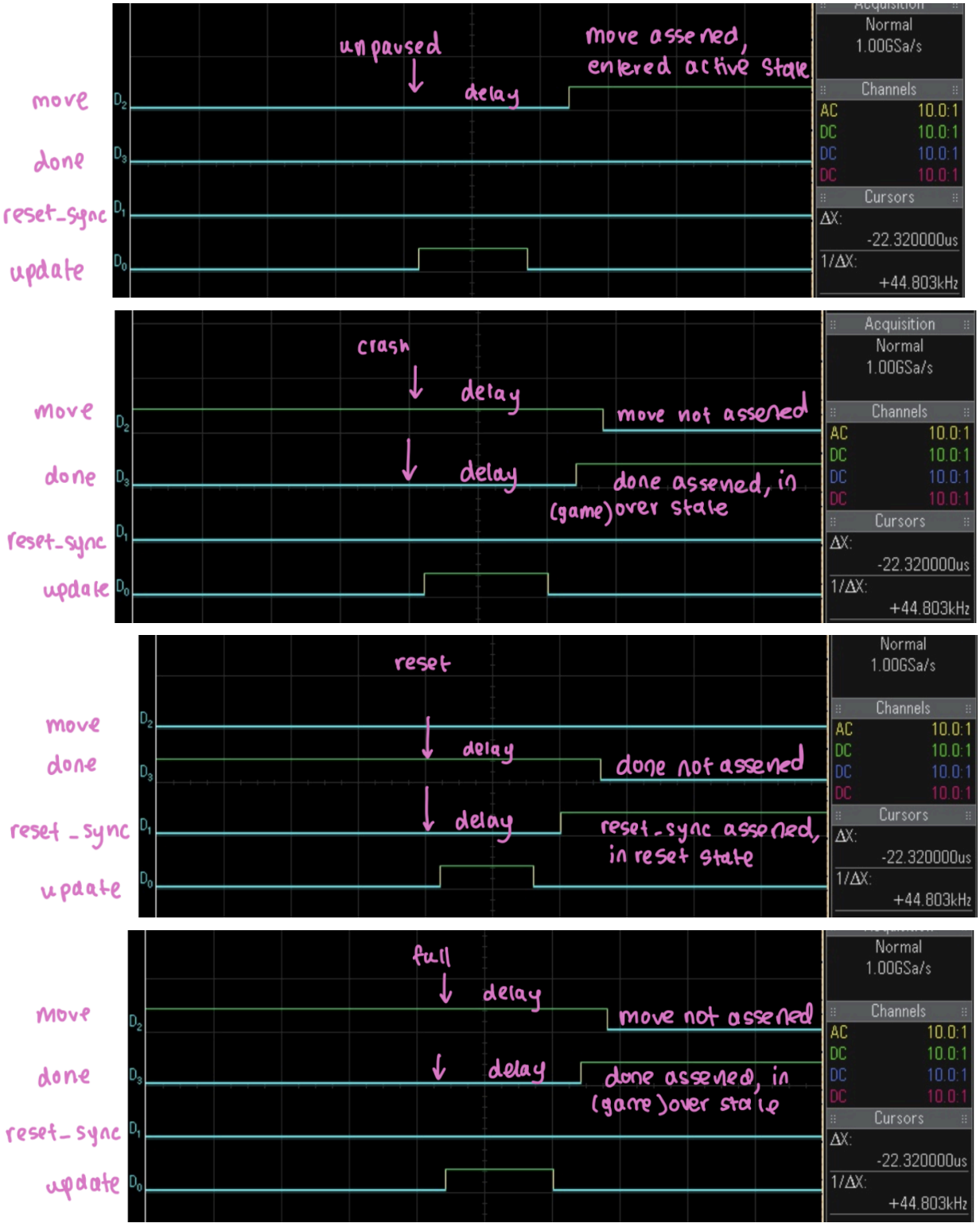


3.7.2. Hardware Validation

We needed six screenshots to capture our hardware validation.






move
done
reset_sync
update
unpaused
delay
move assered, entered active state
crash
delay
move not assered
delay
done assened, in (game)over state
reset
delay
done not assened
delay
reset_sync assened, in reset state
full
delay
move not assened
delay
done assened, in (game)over state
Normal
1.00GSa/s
Channels
AC 10.0:1
DC 10.0:1
DC 10.0:1
DC 10.0:1
Cursors
ΔX:
-22.320000us
1/ΔX:
+44.803kHz

3.8. Input Conditioner

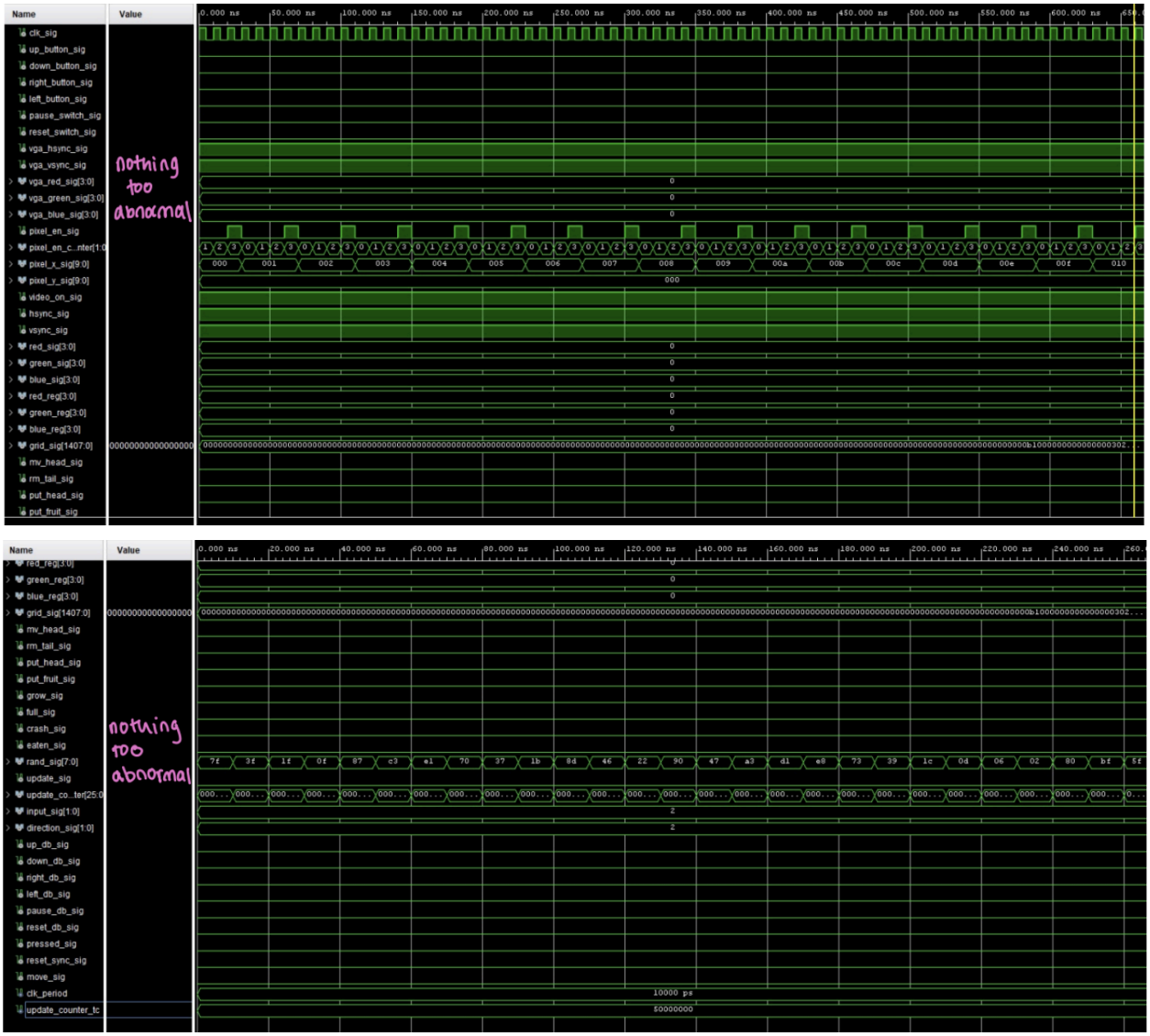
We omit testing details for the input conditioner because it was already tested extensively during the lab, by us and other students.

3.9. Overall Design

After testing each of our components, we tested our overall system.

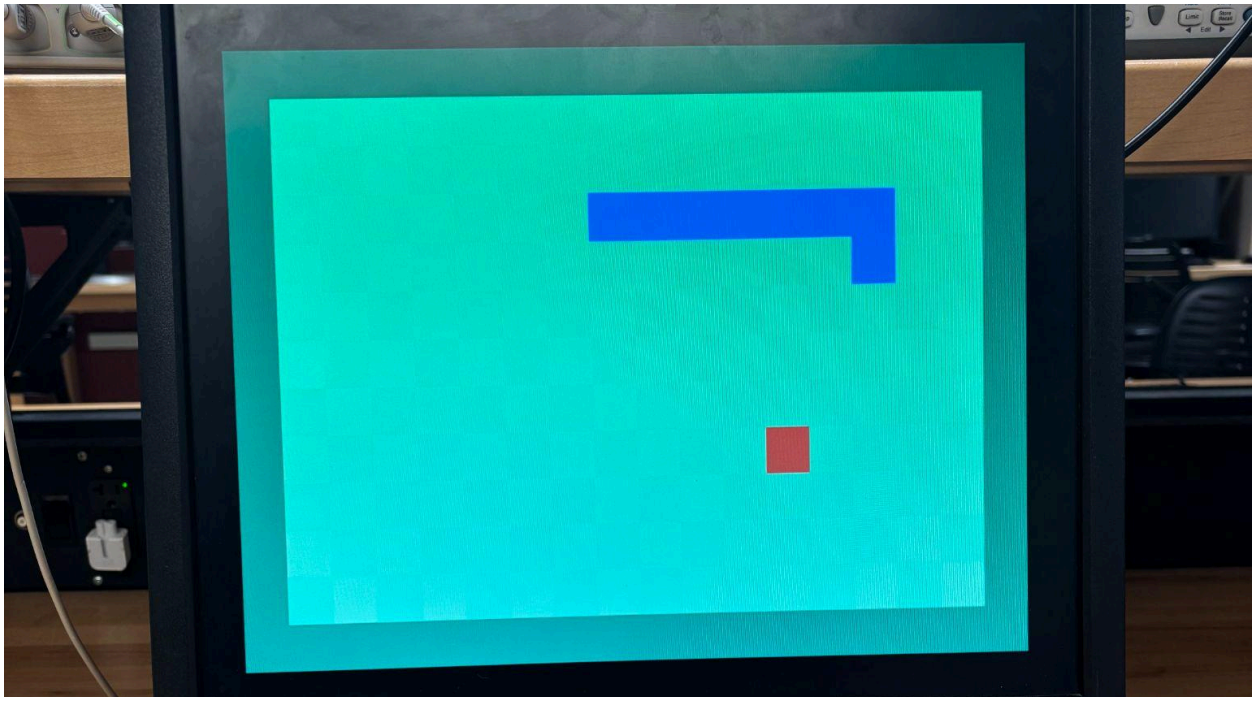
3.9.1. Behavioral Simulations

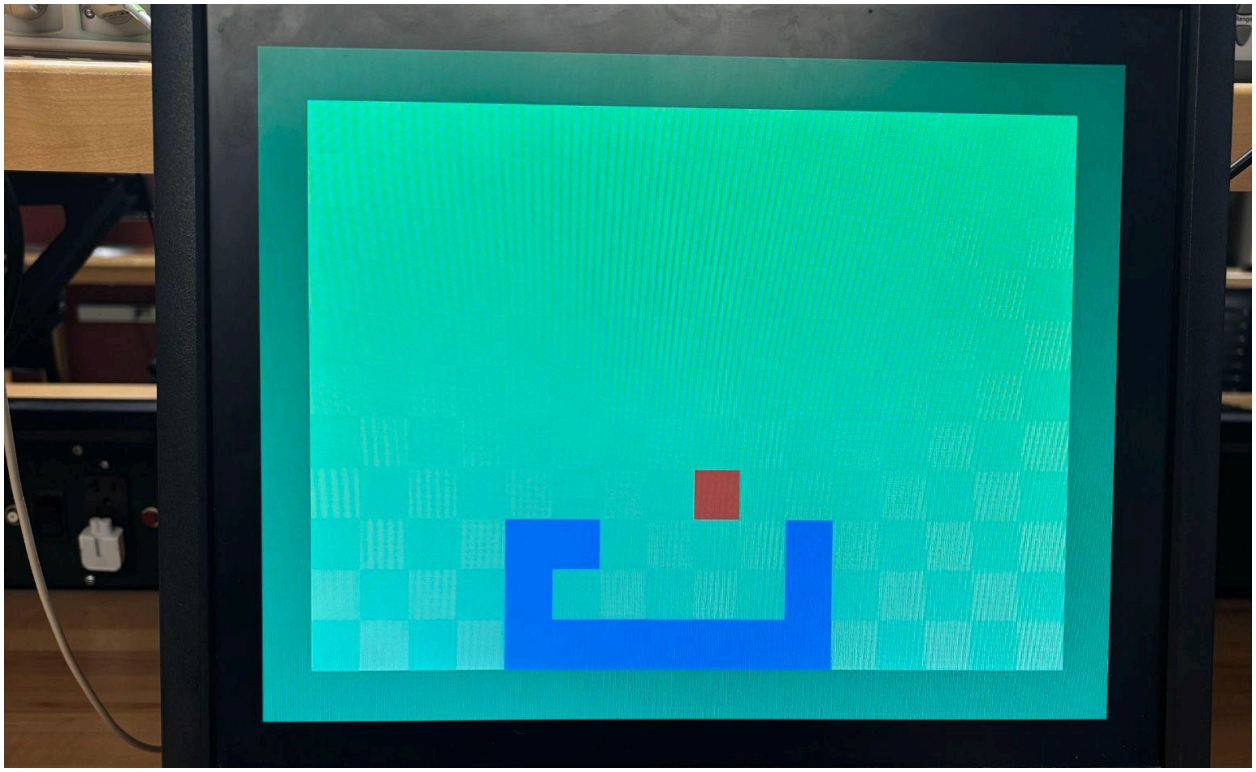
Because it is not practical to check our output, which is intended for a VGA, using waveforms, we simply ensured that there were no glaring issues in our simulation (e.g., no yellow or orange waves and no inexplicable assertions).



3.9.2. Hardware Validation

A video of our project working on hardware is available at this link. Additionally, we have attached several images of the game working below.





4. Analysis of the Design

We believe that our project was successful, as we managed to create a polished implementation of Snake with no observable bugs. Things did not go entirely according to plan during our design and implementation process, however. Completing the VGA controller happened much faster than anticipated, and the game logic took much longer than expected, especially the datapath. We had many unexpected bugs in the datapath having to do with integration. Although we managed to get each of our individual components to pass our tests, there were issues (especially with timing) in getting them to work together. A particularly difficult bug to solve was one due to a timing issue in our helper FSM, which resulted in the snake only growing once the tail passed the fruit position (the connection between these two things is complicated and not immediately obvious). A particularly interesting bug was that at one point, a fruit would be placed every update due to an issue with how our datapath and helper FSM worked together. In the end, we managed to get everything working as expected.

4.1. Resource Utilization

We used 4509 slice LUTs out of 20,800 (21.68%), all of which were for logic rather than memory. We used 1597 slice registers out of 41,600 (3.84%), all of which were flip flops rather than latches. We used 544 F7 muxes out of 16,300 (3.34%) and 78 F8 muxes out of 8,150 (0.96%). We used no block RAM, no DSP, and no specific features. We used 21 out of 106 (19.81%) bonded IOBs (input-output blocks) and nothing else. We used only one

BUFGCTRL clock buffer out of 32 (3.13%). Overall, we used a notable but not excessive amount of resources in the FPGA. We are well under the maximum amount of resources available on the Basys 3 board.

4.2. Residual Warnings

The following warnings are about features we do not intend to use: (1) design is not suitable for incremental synthesis, (2) parallel synthesis criteria is not met, (3) netlist is not ideal for floorplanning, and (4) switching activity may result in inaccurate power analysis. Because we do not need incremental synthesis, parallel synthesis, floorplanning, or power analysis, these warnings can be safely ignored. The following warnings exist as a result of intentional choices made for code readability: (1) 3D RAM is not supported, this will be implemented in registers, and (2) pixel_x[4] to pixel_x[0] and pixel_y[4] to pixel_y[0] are unconnected. For the first warning, although our grid is written as if it were 3D (16 x 11 x 8), this is just to make the code more readable. Our design works fine if this is not actually 3D and implemented as registers instead. For the second warning, these ports are unconnected because we shift out the 5 least significant bits of the pixel x and y when mapping from world space to grid space. This results in them having no effect on the output, which is actually intended for our design. The reason they still exist in the port map is because we might want to later utilize these bits (for example, if we wanted to draw sprites instead of squares we would need these bits). Additionally, it may confuse readers of the code if these bits are missing.

4.3. Division of Labor

Nearly all of the code was done through pair programming. For design, we created an initial version together, though many of the iterations made to our designs were thought of outside of meetings. This strategy was effective as it reduced the number of bugs we encountered and prevented us from having to do difficult code merges, as well as wasting time doing code under outdated assumptions.

4.4. Future Work

Here are some features that we would include if we had more time: (1) a text system for writing words to the VGA to display things like win and lose, (2) multiple different color palettes that the users could select between, (3) optional game modifiers (such as power ups and obstacles), and (4) drawing sprites instead of colored squares.

5. Acknowledgements

The following resources were helpful: (1) Nandland's article on variables, (2) EngineersGarage's article on linear feedback shift registers, (3) Digilent's Basys 3 reference manual, (4) University of Oklahoma's constraints file template, (5) the input conditioners from lab 5, and (6) ChatGPT, occasionally (and exclusively) for understanding error messages, formatting code snippets, and

automating tasks such as filling out port maps and creating signals for each port in a component. We were helped by the following teaching assistants: Will Vasquez (assigned to our group), Muneeb Chaudhary, Caroline Moore, Josh Meise, and Joe Quaratiello. Special thanks to Tad for guidance on our design!

6. Conclusions

In conclusion, we have provided a technical description of each component necessary to create a fully functional Snake game using VHDL. Additionally, we have provided a detailed description of our testing process and results, alongside an analysis of our design. If given the chance, we are not sure whether we would do the same project again. We were happy with the end result, and certain parts were very rewarding, but some parts were incredibly stressful. A major issue with choosing Snake as a project is that it is difficult to tell whether things actually work until the system is essentially done (unit testing can only reveal so much). And although some operations are suited to the concurrent nature of the FPGA (e.g., removing the tail of the snake), others (such as randomly placing a square on a grid) would perhaps be more suited to a processor.

Appendix A: VHDL Source Code

- direction_fsm.vhd
    - This is the direction FSM component (2.6).
- dp_helper_fsm.vhd
    - This is the datapath helper FSM component (2.5).
- game_state_fsm.vhd
    - This is the game state FSM component (2.7).
- input_conditioning.vhd
    - This is the input conditioner component (2.8).
- pixel_generation.vhd
    - This is the pixel generation circuit component (2.2).
- randomizer_lfsr.vhd
    - This is the LFSR randomizer component (2.3).
- snake_datapath.vhd
    - This is the snake datapath component (2.4).
- vga_sync.vhd
    - This is the VGA synchronizer component (2.1).

Appendix B: VHDL Testbenches

- clock_generation.vhd
    - This was not used in the final design, so it has no section in the report. It was used to slow down the clock for the oscilloscope, and was provided in lab 5.
- datapath_toplevel.vhd
    - This is the hardware validation code for the datapath (3.4.2).

- direction_fsm_tb.vhd
    - This is the testbench for the direction FSM (3.6.1).
- dirfsm_toplevel.vhd
    - This is the hardware validation code for the direction FSM (3.6.2).
- dp_helper_fsm_tb.vhd
    - This is the testbench for the datapath helper FSM (3.5.1).
- game_fsm_tb.vhd
    - This is the testbench for the game state FSM (3.7.1).
- game_fsm_toplevel.vhd
    - This is the hardware validation code for the game state FSM (3.7.2).
- helper_fsm_toplevel.vhd
    - This is the hardware validation code for the datapath helper FSM (3.5.2).
- pixelgen_tb.vhd
    - This is the testbench for the pixel generation circuit (3.2.1).
- pixelgen_toplevel.vhd
    - This is the hardware validation code for the pixel generation circuit (3.2.2).
- randomizer_tb.vhd
    - This is the testbench for the LFSR randomizer (3.3.1).
- randomizer_toplevel.vhd
    - This is the hardware validation code for the LFSR randomizer (3.3.2).
- snake_datapath_tb.vhd
    - This is the testbench for the datapath (3.4.1).
- snake_shell_tb.vhd
    - This is the testbench for our overall design (3.9.1).
- snake_toplevel.vhd
    - This is the toplevel shell for our overall design (3.9.2).
- testpat_toplevel.vhd
    - This is the hardware validation code for the VGA synchronizer (3.1.2).
- vga_sync_tb.vhd
    - This is the testbench for the VGA synchronizer (3.1.1).
- vga_test_pattern.vhd
    - This is the test pattern code used for validating the VGA synchronizer (3.1.2).

Appendix C: XDC Constraints

- datapath_constraints.xdc
    - This is the constraints file for hardware validation of the datapath (3.4.2).
- fsm1_constraints.xdc
    - This is the constraints file for hardware validation of the direction FSM (3.6.2).
- fsm2_constraints.xdc
    - This is the constraints file for hardware validation of the game state FSM (3.7.2).

- fsm3_constraints.xdc
    - This is the constraints file for hardware validation of the helper FSM (3.5.2).
- rand_constraints.xdc
    - This is the constraints file for hardware validation of the randomizer (3.3.2).
- snake_constraints.xdc
    - This is the constraints file for the overall design (3.9.2).
- vga_constraints.xdc
    - This is the constraints file for hardware validation of both the VGA synchronizer and the pixel generation circuit (3.1.2 and 3.2.2).