

# **6.111 Final Report: Battleship**

Kade Bose and Edward Jin

{kadebose, ehjin}@mit.edu

# Contents

<b>1 Overview</b>	<b>3</b>
<b>2 Block Diagram and Project Structure</b>	<b>4</b>
2.1 Overview . . . . .	4
2.2 Implementation Details . . . . .	4
2.3 Utilization . . . . .	5
<b>3 Interfacing</b>	<b>6</b>
3.1 Button and Mouse Control - Edward . . . . .	6
3.2 UART RX and TX - Kade . . . . .	6
<b>4 Game Logic</b>	<b>8</b>
4.1 Multiplayer Mode - Edward . . . . .	9
4.2 Solo Mode - Edward . . . . .	9
4.2.1 CPU Randomness - Generation and Analysis . . . . .	10
<b>5 Graphics and Audio</b>	<b>11</b>
5.1 Graphics - Kade . . . . .	11
5.2 Audio - Edward . . . . .	14
<b>6 Challenges, Interesting Observations, and Lessons Learned</b>	<b>15</b>
<b>7 Code Appendix</b>	<b>16</b>
7.1 SystemVerilog . . . . .	16
7.1.1 audio_bgm.sv . . . . .	16
7.1.2 audio_explosion.sv . . . . .	18
7.1.3 audio_gen.sv . . . . .	20
7.1.4 audio_miss.sv . . . . .	22
7.1.5 base_board.sv . . . . .	24
7.1.6 blob.sv . . . . .	26
7.1.7 boat2.sv . . . . .	27
7.1.8 boat3a.sv . . . . .	28
7.1.9 boat3b.sv . . . . .	29
7.1.10 boat4.sv . . . . .	30
7.1.11 boat5.sv . . . . .	31
7.1.12 coverage.sv . . . . .	32
7.1.13 debouncer.sv . . . . .	33
7.1.14 explosion.sv . . . . .	34
7.1.15 fsm_state_pkg.sv . . . . .	35
7.1.16 game_state_controller.sv . . . . .	36
7.1.17 game_state_fsm.sv . . . . .	38
7.1.18 graphics_prep.sv . . . . .	47
7.1.19 hit_or_miss.sv . . . . .	50
7.1.20 hv_to_xy.sv . . . . .	51
7.1.21 picture_blob.sv . . . . .	53
7.1.22 placeable.sv . . . . .	54
7.1.23 rng.sv . . . . .	55
7.1.24 rock.sv . . . . .	57
7.1.25 rock_coords.sv . . . . .	58
7.1.26 rock_randomizer.sv . . . . .	59
7.1.27 rx.sv . . . . .	60
7.1.28 select_square.sv . . . . .	62
7.1.29 ship_coords.sv . . . . .	63

7.1.30	start_menu.sv	64
7.1.31	top_level.sv	66
7.1.32	tx.sv	69
7.1.33	user_input.sv	71
7.1.34	water.sv	74
7.1.35	xvga.sv	75
7.2	Python Scripts	77
7.2.1	Audio Generation Script	77
7.2.2	Random Walk Verification	78

## 1 Overview

For our final project in 6.111, we created a game of Battleship that can be played between two FPGAs in Multiplayer mode, or on a single FPGA in Solo mode. Players can modify the size of the board, the number of obstacles they must place their ships around, the presence and volume of background music and sound effects, and the difficulty of the gameplay in Solo mode. Our project allows both FPGAs to be programmed with the same bitstream, with a simple handshaking process at the start of each game determining which FPGA will be setting the game parameters.

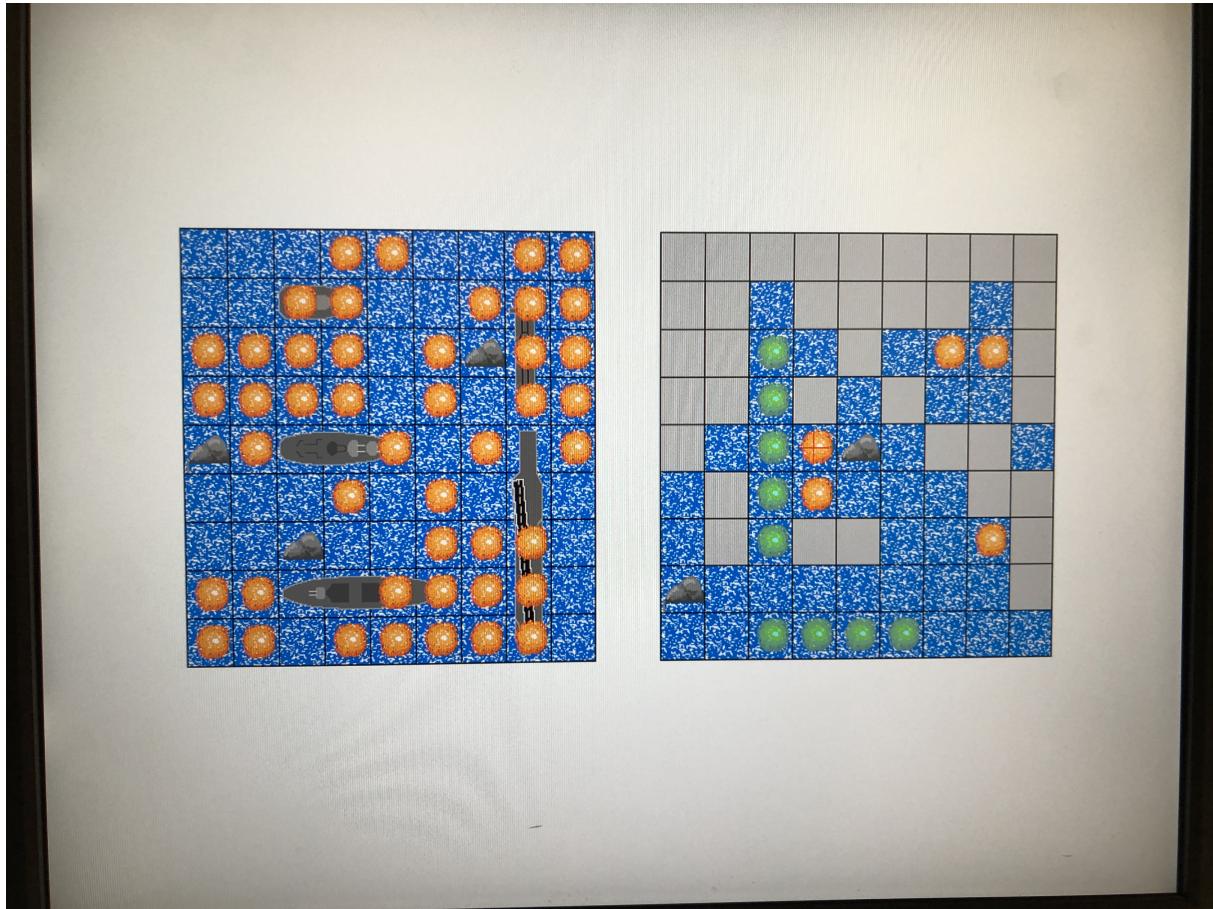


Figure 1: A game of battleship being played over two FPGAs

## 2 Block Diagram and Project Structure

### 2.1 Overview

Our project is broken down into 3 major subsystems: Interfacing, Game Logic, and Graphics/Audio. The block diagram below shows a high level overview of how these modules interact.

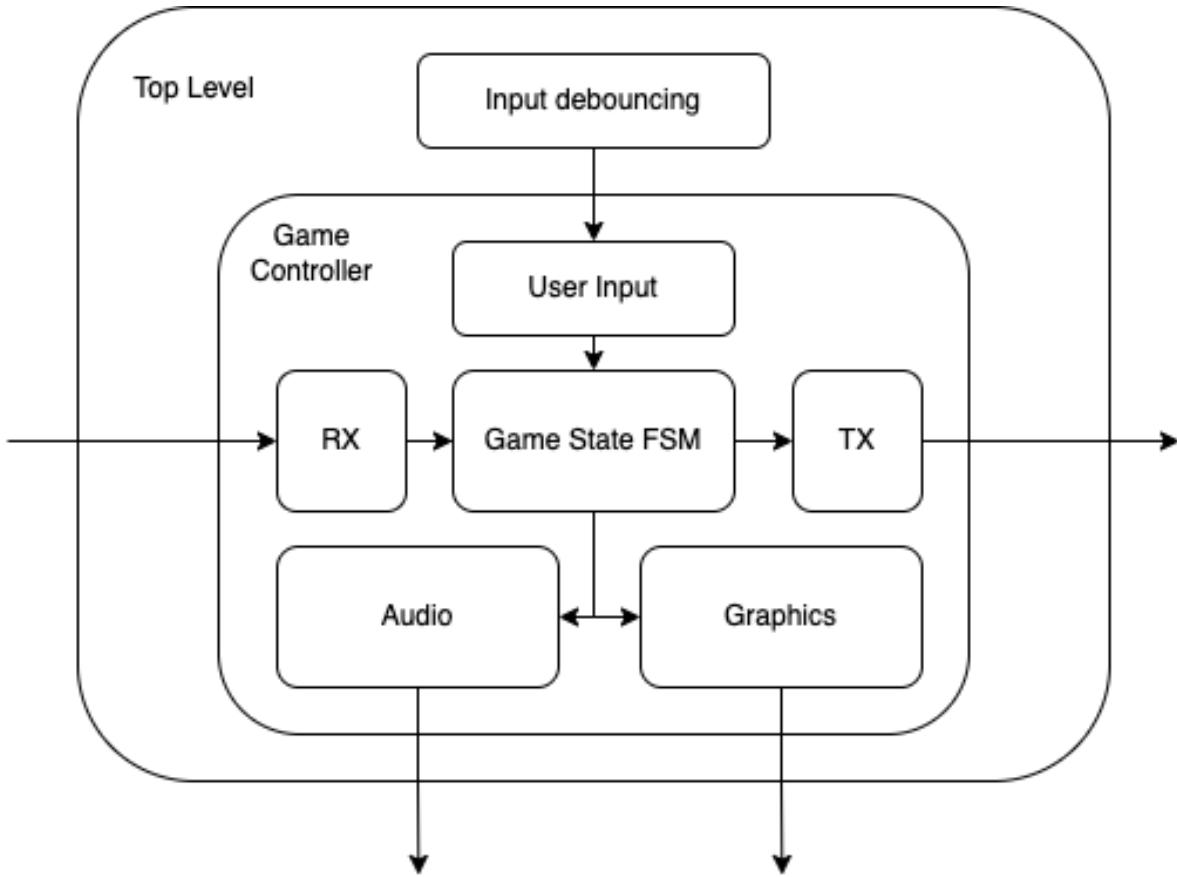


Figure 2: High level module diagram

### 2.2 Implementation Details

The Game State FSM is able to interface with the audio and graphics modules by respectively passing pulses (play\_explosion, play\_miss, play\_bgm) and two 12x12 arrays of the actual game board (opponent\_board and player\_board) to those modules. In order to avoid code repetition and to use some paradigms of object-oriented programming, we used a package (fsm\_state\_pkg.sv) that contains the possible game states as well as a Verilog struct representing the ships on the board. This allowed us to directly import the package in the various Verilog files to avoid repetition of the game state enums and also allowed us to use ship and ship\_array classes as module arguments, rather than passing the individual details one-by-one, improving code readability.

## 2.3 Utilization

As seen below, the utilization statistics show that we used roughly 16 percent of the available LUTs on the FPGA, meaning we had considerable room for expansion if needed. The 80% utilization of BRAMs comes primarily from our audio sources. The image sprites themselves only take up about 25 kB of BRAM storage. The audio storage space could be trimmed down by instead storing Fourier coefficients and using them to create synthetic audio, but we calculated that the entire audio file would fit inside the BRAMs with a 6 kHz sampling rate, meaning that we did not have to bother with this method.

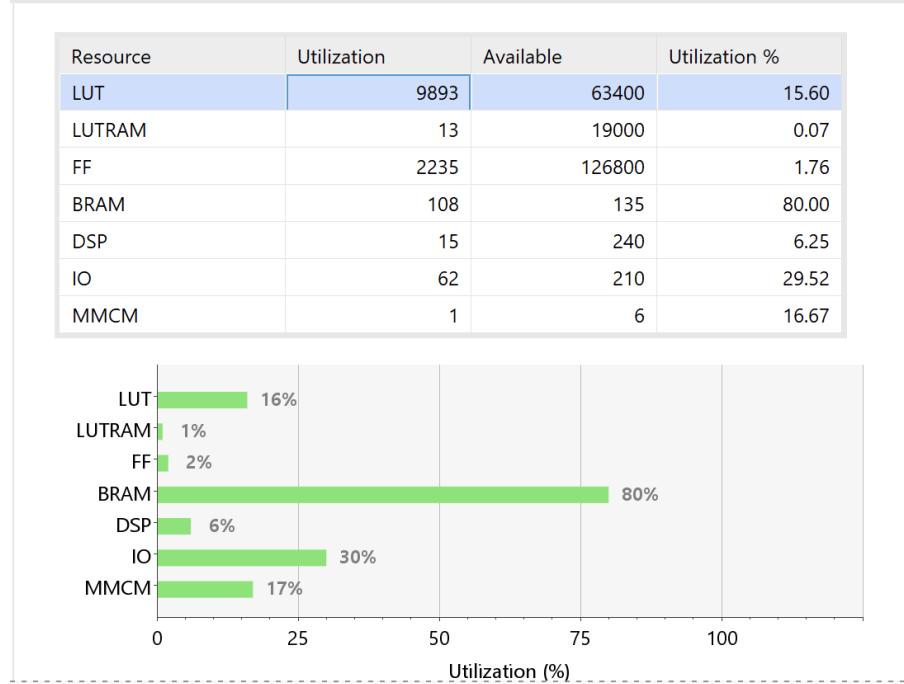


Figure 3: Utilization stats for battleship

### Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

<b>Total On-Chip Power:</b>	<b>0.313 W</b>
<b>Design Power Budget:</b>	<b>Not Specified</b>
<b>Power Budget Margin:</b>	<b>N/A</b>
<b>Junction Temperature:</b>	<b>26.4°C</b>
Thermal Margin:	58.6°C (12.7 W)
Effective θ <sub>JA</sub> :	4.6°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

[Launch Power Constraint Advisor](#) to find and fix invalid switching activity

### On-Chip Power

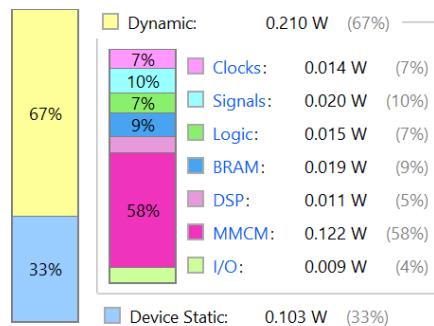


Figure 4: Power stats for battleship

### 3 Interfacing

There are two forms of interfacing implemented in the project. These are interfacing between the FPGA and the player, and interfacing between the two FPGAs in Multiplayer mode.

#### 3.1 Button and Mouse Control - Edward

The user\_input module interprets the external debounced directional input buttons (btnd, btnu, btml, btnr) and mouse position (mousex, mousey), and converts the inputs to  $(x, y)$  coordinates that correspond to map locations.<sup>1</sup> The mouse coordinates were obtained from a PS/2 mouse connected via USB and using the MouseCtl module.<sup>2</sup> Since the possible  $(x, y)$  positions depend on the game state, as well as the actual ship number, ship rotation, and ship lengths when placing ships, these are also inputs into the module.

For button input, the module detects when buttons are pressed by looking for rising edge transitions. For mouse input, the hv\_to\_xy.sv module is able to convert mouse position on the actual VGA monitor to  $(x, y)$  game coordinate squares, and can do so for our own board and the opponent's board.

The module checks the the actual game state to see what  $(x, y)$  are allowed:

- If we are in the start menu, then the only options available are the three options in the start menu. So, the coordinates are bounded from  $(0, 0)$  to  $(0, 2)$ , and only bbtnu and btnd inputs are processed. The mouse positions are tracked and updated when the coordinates enter a bounding box equal to that of the text (same as in start\_menu.sv).
- If we are placing ships, then the bounds for ships are a bit complicated. If we have a ship of length  $l$  and a grid of size  $n \times n$  (numbered from 0 to  $n - 1$ ), then the possible locations for the top left corner of the ship are 0 to  $n - 1$  in the short direction and 0 to  $n - l$  in the long direction. Finally, after each ship is placed, we do not want to leave the  $(x, y)$  position in a disallowed position, so we reset it to the maximum possible given the next ship length. Mouse inputs follow the same constraints, where we check if the mouse cursor position is valid, and if so, we update the  $(x, y)$  coordinates.
- If we are attacking, then we can simply limit the  $(x, y)$  coordinates from 0 to  $n - 1$  as those are the available squares for attacking. Mouse position is translated to coordinates on the opponent's board, leading to the same  $(x, y)$ .

On transition states between the start menu and the game starting, ship placement, and attacking, the module resets the output  $(x, y)$  to 0, as it doesn't make sense to keep the previous  $(x, y)$  coordinates.

#### 3.2 UART RX and TX - Kade

We had two options in terms of communicating between the two FPGAs. The first option involved sending the entire opponent board state between the two devices, at which point the game could make hit and miss decisions locally. We decided against this method in order to make the game more reliant on the communication modules between FPGAs, and to reduce the amount of data needed to be stored and transmitted at start up. The method we decided on was to send packets of information containing single move data between the FPGAs.

We determined that we would need 16 bits to communicate all of the information between the two FPGAs, and so we designed a module that uses an AXI-esque ready, valid handshaking to load a 16 bit buffer and send the data using the UART serial communication protocol. When the TX module is not actively sending across the TX line, it holds the ready signal high. The game logic fsm pulls the valid signal high for a single clock cycle with the 16 bit move data on the data\_in line as soon as it has a move to send. If the TX module is ready, it transmits the 16 bytes in two 10 bit packets. These packets are comprised of 8 data bits, a start

---

<sup>1</sup>The debounced btnc/mouse buttons are instead processed by the game state FSM, as they control gameflow rather than the actual  $(x, y)$  position.

<sup>2</sup>found on the course website: [https://web.mit.edu/6.111/volume2/www/f2020/ps2\\_mouse/](https://web.mit.edu/6.111/volume2/www/f2020/ps2_mouse/)

bit and an stop bit.

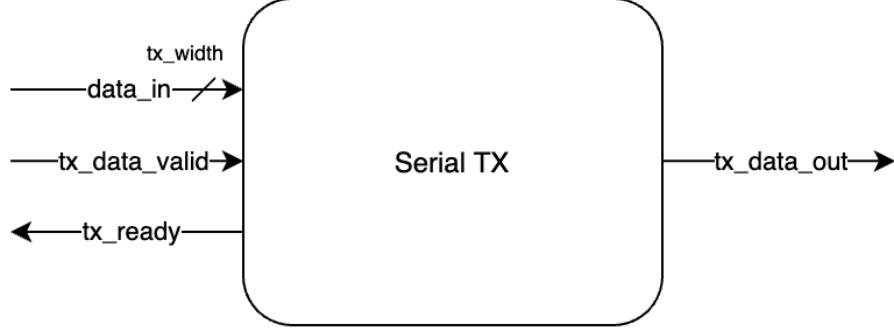


Figure 5: Serial TX module inputs and outputs

The RX module performs the reverse operation. It also has ready, valid and data\_in lines, and it holds valid low until it receives two full UART packets. When the RX line goes low, indicating a start bit, the RX module polls the value of the RX line at the pre-defined baud rate (9600) and stores the incoming data in the 16 bit register. When all 16 bits are received, it immediately raises the valid signal until one clock cycle after the ready signal is asserted. The RX module, in order to reduce metastability, waits half of the baud rate after receiving the start bit before sampling the RX wire, and also synchronizes the input through 2 registers to further protect the integrity of the message.



Figure 6: Serial RX module inputs and outputs

Both of these modules were designed to be parameterized to adjust the size of the buffer register, and in turn the messages sent. The UART protocol stays the same at 8 data bits per packet, however, so the buffer must be a multiple of 8.

## 4 Game Logic

The overall game logic is controlled by the Game State FSM, whose high-level structure is shown below:

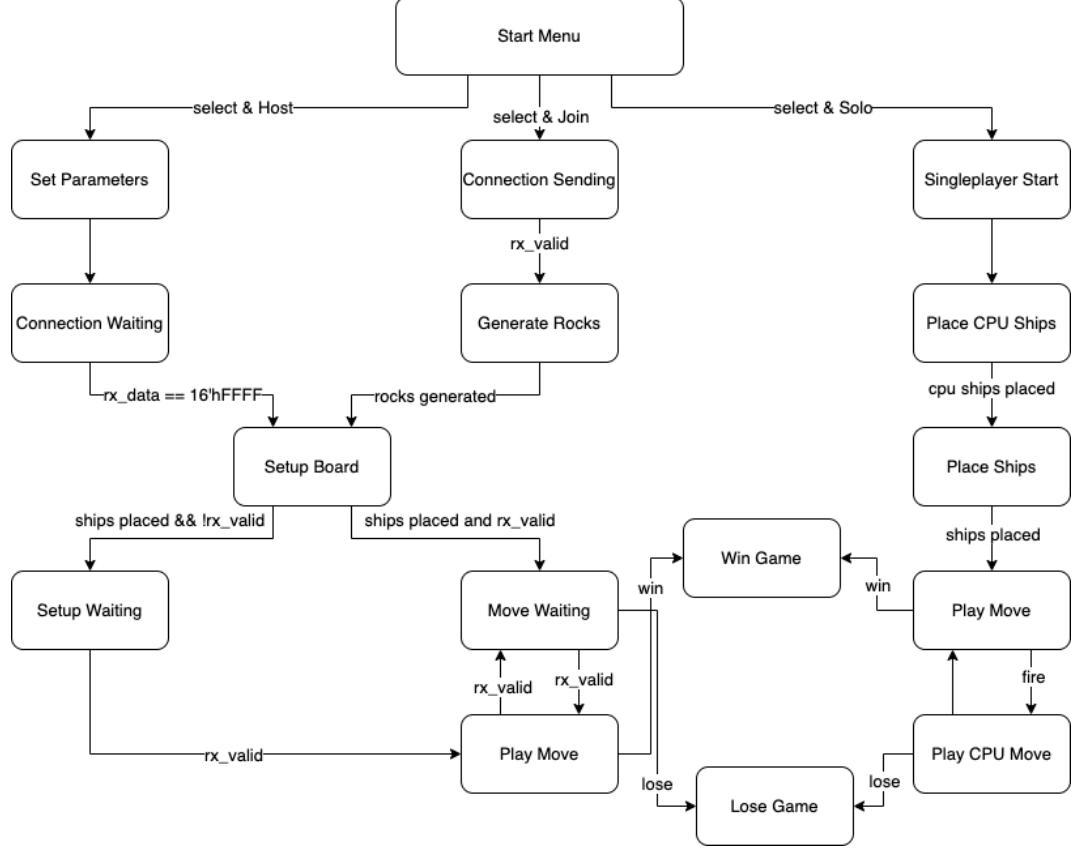


Figure 7: High level Game State FSM

From the start menu, a player can either choose to HOST, JOIN, or play singleplayer (SOLO). The HOST and JOIN options are used in multiplayer mode and respectively set one device as the host and the other as the guest, and the host's game options (number of rocks and game size) are passed to the guest device. The switches on the device were mapped to game options and functions as follows:

- sw[0]: ship rotation (when placing ships)
- sw[2:1]: game size - 9 (game sizes from 9 to 12 are possible)
- sw[3]: hard mode enable (in singleplayer)
- sw[6:4]: number of rocks
- sw[9]: mouse enable
- sw[10]: hit/miss sounds enable
- sw[11]: background music enable
- sw[14:12]: volume control
- sw[15]: reset

## 4.1 Multiplayer Mode - Edward

In multiplayer mode, the devices are connected through the ja[0] and jb[0] ports, which respectively act as RX and TX ports. The devices' ground ports are also connected in order to provide a common ground, lessening the error of incorrect transmission.

When the host device moves to the Set Parameters state, it locks in the game state options that were set with its switches, and waits for a device to join. The joining device joins by sending a 16'FFFF signal over UART, which the host device interprets by sending back a packet {game\_size, num\_rocks, 9'b0} that the guest device sets its game parameters with. Afterwards, both players use the RNG (see section 4.2.1) to randomly generate the requisite number of rocks. This causes both FPGAs to now be in the SETUP\_BOARD state.

In this state, the players can place down their battleships, which is dynamically displayed on the VGA monitor. By using the directional buttons or the mouse, as well as the sw[0] rotation switch or the right mouse button, players can place down ships wherever they are valid. Players confirm the ship placement with the center button or a left mouse click. Collision-detection modules make sure ships cannot be placed on rocks or other ships. In order to incentivize players not to stall at this step, as well as break the symmetry between the devices, we have the player that finishes setup first play the first move by going to the PLAY\_MOVE state while the player that finishes later goes to the MOVE\_WAITING state. This ordering is enforced by sending a 16'hFF00 signal, which indicates a player is done with their placement.

In the MOVE\_WAITING state, the defending player simply has to wait for the opponent to make a move. Once the opponent (who is in the PLAY\_MOVE state) does, then we receive the  $(x, y)$  coordinate of the hit through UART and update our own board to show the opponent's hit location. Then, we check to see if the location intersected any one of our ships or a rock, or if any one of our ships was sunk. Depending on this collision check, we send back the following data over UART:

- If the attack missed, we send back 16'b0.
- If the attack hit, but did not sink a ship, we send back 16'b1000000000000000.
- If the attack hit and sunk ship  $i$ , we send back {2'b11, ships[i].orientation, i[2:0], 2'b00, ships[i].x, ships[i].y}. The position and orientation, as well as the number of the ship are necessary so the attacking player's board can be updated to show a sunk ship in the correct position.
- If the attack hit a rock, then we send back 16'b0100000000000000.

The attacking player then receives this data back and updates their opponents' game board properly. Finally, after each attack, the number of hits that a player has received is counted, and if this number is equal to 17 (the total number of ship squares on the board), then we transition to respective winning and losing states. Otherwise, the player that was previously defending in the MOVE\_WAITING state transitions to the PLAY\_MOVE state, and similarly the attacking player transitions to the MOVE\_WAITING state. This continues until one player wins, which is guaranteed to happen as the game does not allow hitting the same square twice. On game ending states, pressing the center button resets the entire device, bringing users back to the start menu, and allows the users to play another game.

## 4.2 Solo Mode - Edward

The overall gameplay for solo mode is very similar for multiplayer mode, and much of the logic is the same. However, a RNG (see section 4.2.1) is used for generating all CPU actions, placement, and rock positions. The rock positions and CPU ship placement are all randomly generated in the the game is created, while the CPU actions are generated (with a small  $\frac{1}{3}$ -second delay) after each player move. The gameflow is the same, though all the communication from before is done completely with logic on one device instead of using UART communications. The necessity of checking all CPU ship intersections on the player move's added moderate complexity in hardware, using roughly two times as many LUTs as the multiplayer mode.

While we did not have sufficient time to implement a smarter CPU,<sup>3</sup> we compensated lack of intelligence with special powers. When hard mode is enabled, the CPU is able to fire three shots after every player shot, instead of one. This results in a win probability experimentally found to be around  $\frac{1}{10}$ , while the player nearly always wins in easy mode.

#### 4.2.1 CPU Randomness - Generation and Analysis

The randomness used to direct the CPU logic was generated with an *xorshift* linear feedback shift register, which involves taking bitwise ORs of shifted versions of the initial seed. We did not need long random numbers, and so we ended up using a simple 32-bit number xorshift register with a period of  $2^{32} - 1$  that updates on each clock cycle, making the RNG extremely difficult to exploit. The RNG is fast and cheap in hardware, and it is random enough for our purposes.

From this RNG that produces 32-bit output, we needed to create bounded random numbers, as many parameters (ex. ship placement, where to hit on the board, etc) are dependent on the game size as an upper bound. While the standard way to do this in software is to use modulo with the desired bound on the random number, it is harder to implement in hardware as divisions are expensive. To get around this limitation, we ran Markov-Chain Monte Carlo simulations of random walks, with randomness generated from that 32-bit xorshift RNG. The walks themselves are cheap (in hardware) to simulate, and it is easy to create a walk graph that has uniform stationary distribution, by allowing each number to transition to any adjacent number (or not transition at 0 or the maximum number), each with probability  $\frac{1}{2}$ . As each vertex has identical indegree and outdegree, this graph has a uniform stationary distribution; further, due to aperiodicity it is guaranteed to converge to the uniform distribution.

To ensure that the generated numbers are indeed random, we need to make sure we give enough time for the random walk to be truly random. Exponentiation of the walk matrix (with numpy, see code appendix) revealed that simulating 500 steps of the random walk would result in the uniform distribution starting from any distribution, within  $10^{-8}$  error. As a result, before the random numbers are sampled, we wait 500 clock cycles, to ensure that our bounded random numbers are truly random.

In terms of CPU randomness, our bounded RNG allows us to generate uniformly bounded random numbers that control things such as ship placement and where to attack. The CPU generates bounded random numbers as above, checks to see if they work, and if not, generates new ones. Even in the worst case, on the  $12 \times 12$  map, with all but one square filled up, there is still a  $\geq \frac{1}{12^2} = \frac{1}{144}$  chance of generating a valid square. This means that after 144 iterations we would have a  $\leq \frac{1}{e}$  chance of failing to generate a good random number, and after  $10 \cdot 12^2 < 2000$  iterations our probability of failure is  $\leq \frac{1}{e^{10}}$ . 2000 clock cycles is fast enough to be not noticeable by the player, so this method of ‘try it until it works’ for random generation is sufficient for our purposes.

One potential issue, which never popped up in our testing but is theoretically possible, is when there is no place to place the actual CPU ships. This is only possible on a  $9 \times 9$  board, as there are enough open spaces in a  $10 \times 10$  or bigger board to prevent this from ever happening.<sup>4</sup> However, this requires an extremely unlikely configuration (7 rocks in the middle row, another ship blocking the other two unfilled columns, and the last 5-long ship being set to a vertical orientation). While this configuration can theoretically occur, softlocking the game, it is extremely unlikely. One way to work around this issue would be to rerandomize CPU ship orientation, generate CPU ship positions first, and then place rocks afterwards, which guarantees a feasible board. However, we did not prioritize this change, as this would lead to no change in gameplay quality due to its extremely low probability of occurrence.

---

<sup>3</sup>This could have been done, for example, by having the CPU shoot squares close to previous hits instead of random squares

<sup>4</sup>Proof: there are at least 20 disjoint  $5 \times 1$  areas to place the last ship, and 19 total items placed before this (12 ship squares and 7 rocks), so one  $5 \times 1$  area must be completely empty

## 5 Graphics and Audio

### 5.1 Graphics - Kade

The first interaction the player has with the game is the main menu, which uses blobs and a font rom to look up c string characters and display them on the screen. A red select blob overlays the currently selected option. The position of the red blob changes as the user presses the button or moves the mouse within the area of the choice blobs.

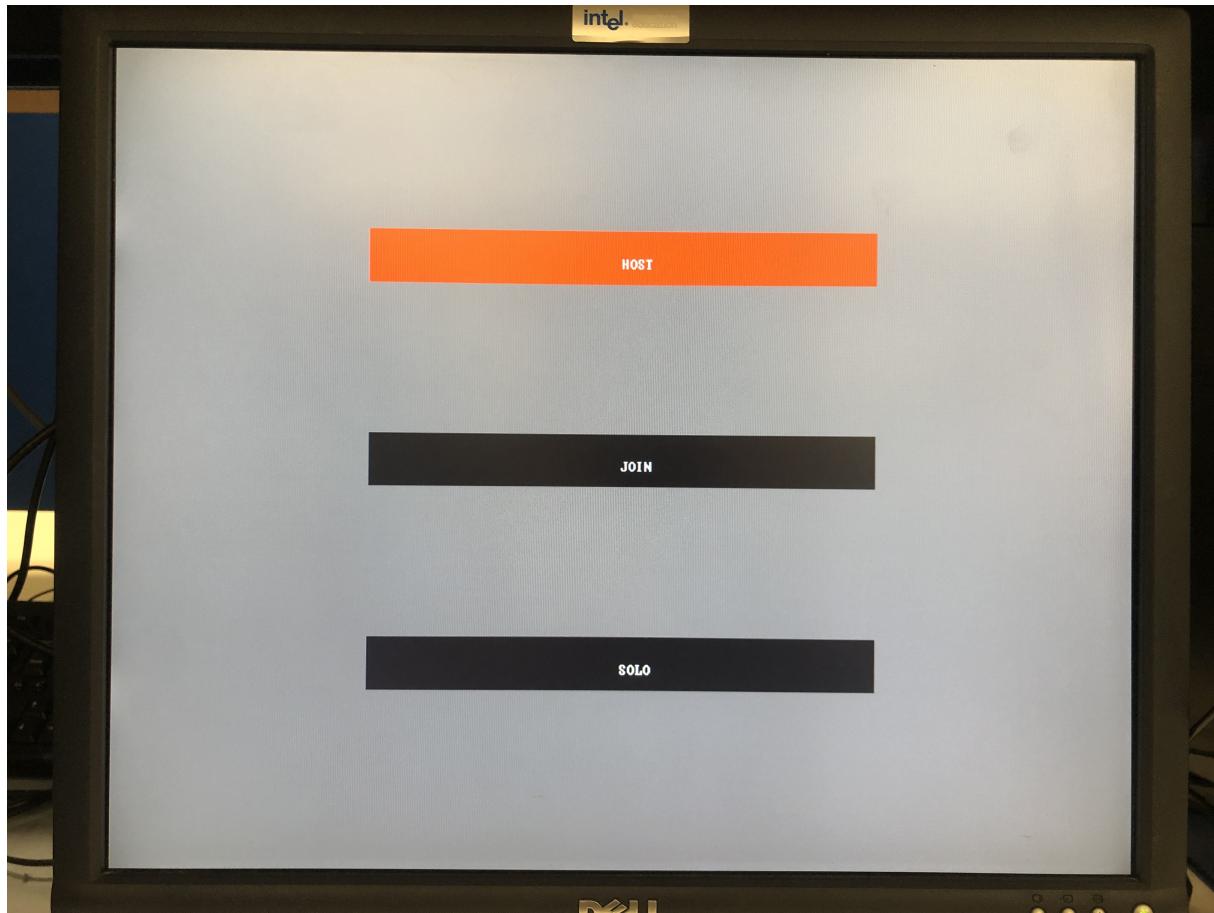


Figure 8: Main menu screen

Once the game begins, in order to display the state of the board, I created a graphics preparation module that muxes together a number of different sprite pixels to output the appropriate pixel on a display using VGA. To generate the signals for the hcount and vcount of the display, I used the XVGA module from Lab 3. These signals allowed me to select the appropriate pixel based on the hcount, vcount coordinate we were currently displaying. To generate these signals, the module needs a 65 MHz clock, and because our project only relied on human input and VGA display, we chose to have the system clock use this 65 MHz clock in order to eliminate the need for any clock domain crossing. There are a number of different sprites I designed to be displayed on the VGA display, and each module takes hcount and vcount as inputs along with some variety of signals such as game state or board data, and outputs a single 12 bit value indicating the color of the pixel (R,G,B). If a sprite's pixel value is 12'h000, meaning black, the pixel is not displayed. For sprites that needed to use black, such as the board lines, 12'h0F0 was chosen as the default value.

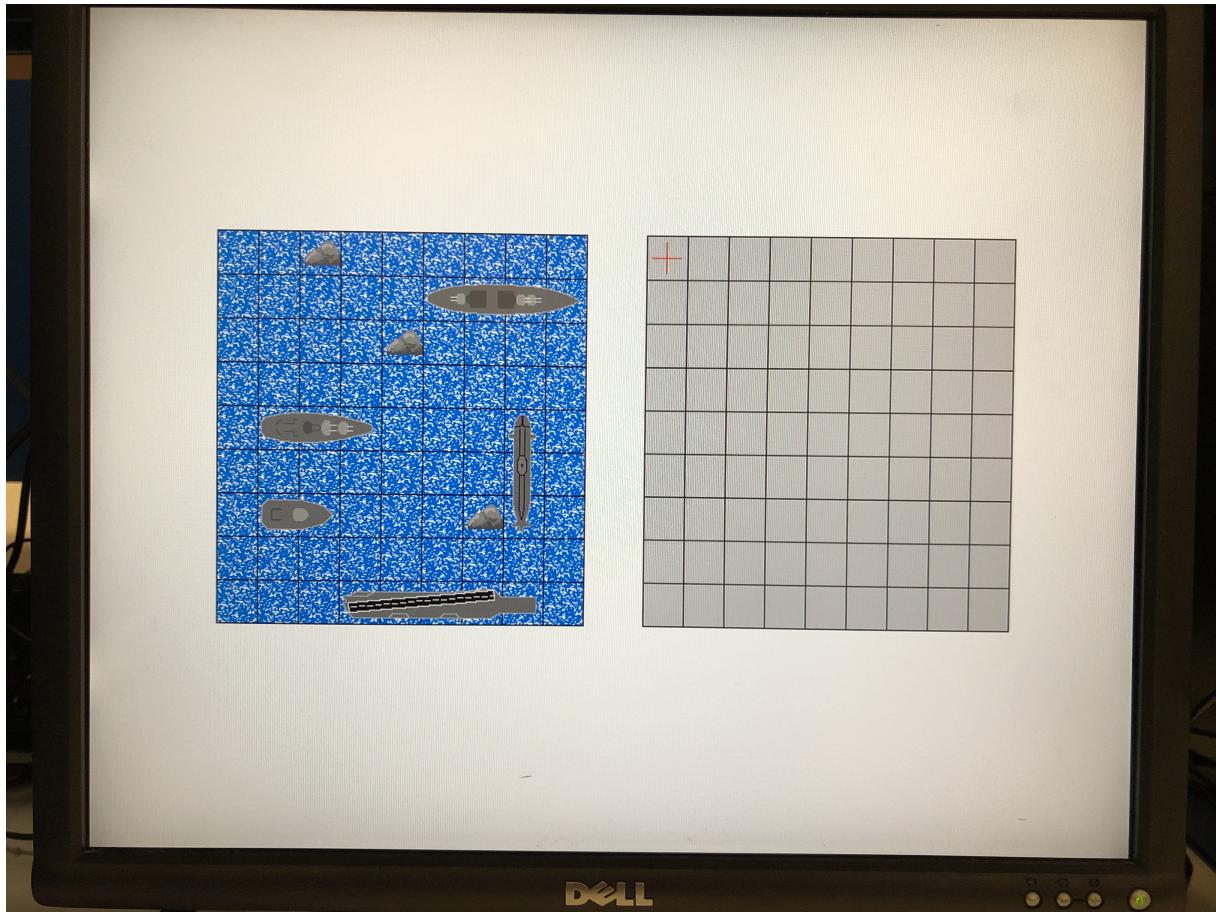


Figure 9: Game view after placing all ships and prepared to fire

There were several different sources of sprite data. Some sprites, such as the ships and the rock obstacles, were stored in BROMs and looked up based on the values of hcount and vcount, while others, such as the board and select cursor, were generated combinationally. The water sprites were generated using linear feedback shift registers as a means of generating pseudo-random patterns.

The ship, explosion, and rock sprites were created in Adobe Illustrator and ran through a Python script to generate four COE files per sprite: one that contains  $image\_height * image\_width$  8 bit values, and three  $8 * 256$  color BROMs containing the 256 most common shades of each color in the image. The image BROM values index into each of the color BROMs, and the output of each color BROM is truncated and concatenated into a 12 bit color. The challenge with using these is ensuring proper pipelining throughout the graphics module, especially when adding new sprites or registers to ensure that timings are met. To do this, I pipelined the XVGA signals using shift registers that could quickly add additional pipeline stages when needed. Each BROM was generated with one cycle of delay for lookup and one cycle for returning the output. With two connected BROMs, this results in 4 cycles of delay for lookup and another setting the pixel output. I therefore pipelined the XVGA signals to correctly match this offset. All sprites were generated using a modified picture blob module, with a top level wrapper module created to separate the BROMs from the picture blob, allowing for considerable reuse. The modified picture blob also allowed for rotation of sprites by modifying the calculated address based on a single bit. The calculation for the address of a rotated  $image\_height * image\_width$  sprite is  $image\_width - 1 - (vcount\_in - y\_in) + (hcount\_in - x\_in) * image\_width$  where  $x\_in$  and  $y\_in$  are values between the max width and height of the screen in pixels. The rotation allows users to flip the ship orientation when placing them on the board, just as you would be able to in the physical game. This helped save storage on the FPGA as only a single version of the sprite had to be stored in ROM.

In addition to the BROM sprites, there were also combinationally generated sprites displayed. Because these were calculated in a single clock cycle, they had to be pipelined in the same way as the XVGA signals in order to maintain synchronicity through the graphics pipeline. Examples of combinational sprites include the game board row and column lines, the grey coverage squares on the opponent board, the select cross-hairs for aiming shots, and the background.

The coverage module ties together two instances of hit or miss modules, which each take in two  $12 \times 12 \times 3$  arrays representing board states, and displays the appropriate sprite in each grid square, encoded as follows:

- 3'b000: Covered
- 3'b001: Hit
- 3'b010: Miss
- 3'b011: Rock
- 3'b100: Sunk

Covered was represented by a grey square, hit was represented by the explosion sprite, miss was represented by all black which was then not drawn (leaving water behind), rocks were represented by the rock sprite, and sunk was represented by alpha blending the explosion with  $12'h000$ , giving the explosion a green tint. The formula for alpha blending the explosion sprite and the white square is  $e\_pixel >> 1 + 12'h000 >> 1$ , essentially removing a large portion of the red shades in the image.

In order to display the appropriate coverage sprite at hcount and vcount, I had to translate our current location in hcount and vcount into x,y coordinates to index into each of the  $12 \times 12 \times 3$  arrays. I created a conversion module which would return the appropriate x and y indexes for the current square, which was then used to set the location and type of the square we were currently drawing. The conversion modules took in a start location in hcount and vcount on the screen and calculated the x and y value by checking if the value was between a certain threshold defined by the size of each grid square. One conversion module was required for each board, player and opponent.

The water pixels were generated using a linear-feedback shift register, similar to that of the CPU rng, to generate pseudo-random sequences. It works by starting with a seed and applying a linear function using xnor operations. Using Xilinx's documentation, I was able to choose the primitive polynomial for the seed length that I chose (10 bits), guaranteeing the maximum period before repetition of the generated sequence. I chose this value because it gave a sufficiently long period to cover one length of the board before repeating and was of the same size as vcount. This means that the maximum value generated could be 1023, and the minimum value could be 0. Using the vcount value to start the sequence, I applied the maximum linear function and checked whether the value generated was greater than 767, or three-quarters of the max value, giving a 75% chance of returning false. If the generated value was greater than 767 the module outputs a white pixel, otherwise it outputs a blue pixel. The value of 767 could be changed to vary the ratio between blue and white pixels in the water, or even to add additional shades of blue. Each time a new grid square was reached, the module reset the seed, giving a semi-repeating pattern while still maintaining a random appearing sequence. By resetting the seed, a smaller primitive polynomial could be used.

## 5.2 Audio - Edward

As mentioned previously, we were able to fit all our audio at a 6 kHz sampling rate onto the onboard BRAMs, so we did not need to use external memory. 6 kHz was chosen to balance the amount of memory used versus the quality of the audio. At a 6 kHz sampling rate, we can store approximately 80 seconds of audio with the 500 KB onboard BRAM memory. The actual audio we used was around 60 seconds total, so it ended up using about 360 KB of the BRAM, which is within device limits.

The audio was generated from 32-bit 44.1 kHz sampling rate wav files by first taking a discrete Fourier transform to the frequency domain, then applying a bandpass filter to remove any frequencies greater than the Nyquist frequency of 3 kHz. The inverse Fourier transform was then applied, and was sampled at 6 kHz. Finally, the audio coefficients were rounded to the nearest multiple of  $2^{-7}$  and converted to signed binary for usage in the COE files, which were then stored in Vivado Block ROM IPs.

The audio was played with similar code from lab 5, which involved reading the audio coefficients from the BROM and then setting them to be the output, making sure to transition to the next coefficient at the appropriate sampling rate. Each different audio file produced a different coefficient, and these coefficients were properly summed (depending on the input switches) in order to overlap sounds properly without overflow. These final summed coefficients were then scaled in order to control the volume, and finally passed to an audio PWM module that outputs the actual audio to the audio jack. The game state FSM controls when and what type of audio will be played, by passing in input pulses that start incrementing the address counters in the BROMs and cause them to play audio.

## 6 Challenges, Interesting Observations, and Lessons Learned

An initial challenge we ran into was creating a reliable communication channel between the two FPGAs. The first limitation in the implementation was that there was a misunderstanding in how the Serial TX module works. The module required a single clock cycle valid signal to send the data, but the game state FSM held valid high, causing tx to send two back to back packets. This caused issues because although the first transaction was cleared from the buffer, the state machine saw an additional transaction which was interpreted as unique. Additionally, the handshaking process that happens between the FSM and the Serial modules requires three clock cycles, one for the FSM or Serial to raise valid, one for the corresponding module to raise ready, and one for the initial module to lower valid. Our initial FSM did not take this into consideration, leading to multiple state changes happening during the handshaking. To combat this issue, additional delay states were added to the FSM to ensure that a transaction could be completed before the FSM attempted to continue. Although the modules were tested in simulation, this behavior was difficult to diagnose because of the vast difference between the baud rate and the system clock. This led to the parameterization of the module to allow for simplified simulation and ILA analysis. Parameterizing the RX and TX modules from the start would have made the debugging process much easier.

Another interesting challenge we ran into was during the creation of the singleplayer game mode. We attempted to have the singleplayer game mode behave similarly to how the multiplayer version works, but without the rx/tx data transfers. Instead, the data was replaced with randomly generated numbers, that would tell the CPU where to attack on our board. Adding the check for if this randomly-generated square hit one of our ships led to extremely long synthesis times (on the order of 20 minutes) and an implementation that would not finish within 30 minutes. According to the synthesis report log file, our code after adding singleplayer took around 70000 multiplexers, much more than the FPGA itself can handle. We hypothesized that the reason for this was that doing such a collision check in a single clock cycle is extremely costly, as we need to check all locations of all ships against all possible board squares of the CPU's attack in hardware. We did not run into this problem in multiplayer as one device generated the coordinates to attack, while the other actually checked if this position intersected the ships, allowing each device to do part of the work and only have one for loop instead of nested for loops. This asynchronous generation/checking was later replicated in the singleplayer mode, by having the random attack be generated on one clock cycle and having it be checked against the board on the next clock cycle, reducing singleplayer mode to use around 12000 multiplexers total and reasonable (about 5-10 minutes) of synthesis and implementation times. This issue overall taught us to not nest for loops and break down complicated logic over multiple cycles whenever possible, as well as the limits of Vivado and our FPGA.

In terms of lessons learned, we were made aware of the value of having clearly communicated interfacing constraints. Making sure that each module had clearly defined requirements would have saved us several days worth of debugging that could have been devoted to further feature integration. Although we feel we utilized our time effectively and reached the goals we set out to achieve at the beginning of the project, we would have benefited from starting the process slightly sooner. Another skill we both acquired was using testbenches to save us countless hours of debugging. Creating new projects to perform development and testing before integrating the module into the project was something that we found was worth the overhead of setting up a new project. This process flow also forced us to design with reusability in mind, making the project much more flexible if needed.

## 7 Code Appendix

### 7.1 SystemVerilog

#### 7.1.1 audio\_bgm.sv

```
1  `default_nettype none
2
3  module audio_bgm #(BROM_SIZE = 329286,
4                      SAMPLE_RATE_PARAMETER = 10833) (
5      input wire clk_in,           // 65MHz system clock
6      input wire rst_in,          // 1 to reset to initial state
7      input wire play_audio,      // 1 when we want to start playing audio
8      output logic signed [7:0] data_out,    // 8-bit PCM data to headphone
9      output logic ready //1 when done with audio
10 );
11     logic sample_clk;
12
13     logic[$clog2(SAMPLE_RATE_PARAMETER)+1:0] sample_counter;
14     assign sample_clk = (sample_counter == 0);
15     logic prev_play_audio;
16
17     always_ff @(posedge clk_in) begin
18         prev_play_audio <= play_audio;
19         if (rst_in || sample_counter == SAMPLE_RATE_PARAMETER - 1) begin
20             sample_counter <= 0;
21         end else begin
22             sample_counter <= sample_counter + 1;
23         end
24     end
25
26     logic [$clog2(BROM_SIZE) - 1 : 0] addr;
27     logic [$clog2(BROM_SIZE) - 1 : 0] max_addr;
28     logic [7:0] data_from_bram;
29     bgm bgm (.addr(addr), .clka(clk_in), .douta(data_from_bram));
30
31     enum {IDLE, PLAYING} state;
32     always_ff @(posedge clk_in) begin
33         if(rst_in) begin
34             addr <= 0;
35             data_out <= 0;
36             state <= IDLE;
37             ready <= 1;
38         end
39         else begin
40             if(state == IDLE) begin
41                 if(~prev_play_audio & play_audio) begin
42                     state <= PLAYING;
43                     addr <= 0;
44                     ready <= 0;
45                 end
46             end
47             else if(state == PLAYING) begin
48                 if(sample_clk) begin
49                     if(addr < BROM_SIZE) begin
50                         addr <= addr + 1;
51                         data_out <= {~data_from_bram[7], data_from_bram[6:0]};
52                         //convert from signed binary bram
53                     end
54                 end
55             end
56         end
57     end
58
59     assign data_out = (state == PLAYING) ? data_from_bram : 0;
60
61     assign ready = (state == IDLE) ? 1 : 0;
62
63     assign sample_clk = (sample_counter == 0) ? 0 : 1;
64
65     assign max_addr = BROM_SIZE - 1;
66
67     assign data_from_bram = (addr <= max_addr) ? data_from_bram : 0;
68
69     assign data_out = (addr <= max_addr) ? data_from_bram : 0;
70
71     assign ready = (addr > max_addr) ? 1 : 0;
72
73     assign sample_counter = (sample_counter == SAMPLE_RATE_PARAMETER - 1) ? 0 : sample_counter + 1;
74
75     assign prev_play_audio = (sample_counter == 0) ? 0 : play_audio;
76
77     assign play_audio = (state == PLAYING) ? 1 : 0;
78
79     assign clk_in = (rst_in) ? 0 : clk_in;
80
81     assign rst_in = (rst_in) ? 1 : 0;
82
83     assign play_audio = (play_audio) ? 1 : 0;
84
85     assign data_out = (data_out) ? 1 : 0;
86
87     assign ready = (ready) ? 1 : 0;
88
89     assign sample_clk = (sample_clk) ? 1 : 0;
90
91     assign sample_counter = (sample_counter) ? 1 : 0;
92
93     assign max_addr = (max_addr) ? 1 : 0;
94
95     assign data_from_bram = (data_from_bram) ? 1 : 0;
96
97     assign bgm = (bgm) ? 1 : 0;
98
99     assign addr = (addr) ? 1 : 0;
100    assign clka = (clka) ? 1 : 0;
101
102    assign douta = (douta) ? 1 : 0;
```

```
54         else begin //looping bgm
55             addr <= 0;
56         //             ready <= 1;
57         //             state <= IDLE;
58         end
59     end
60 end
61 end
62 endmodule
63
64 `default_nettype wire
```

### 7.1.2 audio\_explosion.sv

```
1 module audio_explosion #(BROM_SIZE = 19511,
2                         SAMPLE_RATE_PARAMETER = 10833) (
3     input logic clk_in,                      // 65MHz system clock
4     input logic rst_in,                      // 1 to reset to initial state
5     input logic play_audio,                 // 1 when we want to start playing audio
6     output logic signed [7:0] data_out,      // 8-bit PCM data to headphone
7     output logic ready //1 when done with audio
8 );
9     logic[$clog2(SAMPLE_RATE_PARAMETER)+1:0] sample_counter;
10    assign sample_clk = (sample_counter == 0);
11    logic prev_play_audio;
12
13    always_ff @(posedge clk_in) begin
14        prev_play_audio <= play_audio;
15        if (rst_in || sample_counter == SAMPLE_RATE_PARAMETER - 1) begin
16            sample_counter <= 0;
17        end else begin
18            sample_counter <= sample_counter + 1;
19        end
20    end
21
22    logic [$clog2(BROM_SIZE) - 1 : 0] addr;
23    logic [$clog2(BROM_SIZE) - 1 : 0] max_addr;
24    logic [7:0] data_from_bram;
25    explosion_audio explosion_audio (.addr(a(addr)), .clk_a(clk_in),
26                                     .dout_a(data_from_bram));
27
28    enum {IDLE, PLAYING} state;
29    always_ff @(posedge clk_in) begin
30        if(rst_in) begin
31            addr <= 0;
32            data_out <= 0;
33            state <= IDLE;
34            ready <= 1;
35        end
36        else begin
37            if(state == IDLE) begin
38                if(~prev_play_audio & play_audio) begin
39                    state <= PLAYING;
40                    addr <= 0;
41                    ready <= 0;
42                end
43            end
44            else if(state == PLAYING) begin
45                if(sample_clk) begin
46                    if(addr < BROM_SIZE) begin
47                        addr <= addr + 1;
48                        data_out <= {~data_from_bram[7], data_from_bram[6:0]};
49                        //convert from signed binary bram
50                    end
51                    else begin
52                        ready <= 1;
53                        state <= IDLE;
54                    end
55                end
56            end
57        end
58    end
59
```

```
57      end
58  endmodule
```

### 7.1.3 audio\_gen.sv

```

1 module audio_gen(input wire clk,
2                   input wire reset,
3                   input wire bgm_on,
4                   input wire sounds_on,
5                   input wire play_explosion,
6                   input wire play_miss,
7                   input wire play_bgm,
8                   input wire[2:0] volume_control,
9                   output logic aud_pwm,
10                  output logic aud_sd);
11
12 assign aud_sd = 1;
13 logic pwm_val;
14
15 logic [7:0] vol_out;
16 logic signed [7:0] bgm_out;
17 logic signed [7:0] exp_out;
18 logic signed [7:0] miss_out;
19
20 audio_bgm bgm(.clk_in(clk), .rst_in(reset), .play_audio(play_bgm),
21                 ↪ .data_out(bgm_out), .ready());
22 audio_explosion explosion(.clk_in(clk), .rst_in(reset),
23                           ↪ .play_audio(play_explosion), .data_out(exp_out), .ready());
24 audio_miss miss(.clk_in(clk), .rst_in(reset), .play_audio(play_miss),
25                  ↪ .data_out(miss_out), .ready());
26
27 logic signed [7:0] bgm_quiet;
28 assign bgm_quiet = bgm_out >>> 1;
29 logic signed [7:0] audio_signal;
30 always_comb begin
31   if(bgm_on && sounds_on)
32     audio_signal = bgm_quiet + ((miss_out + exp_out) >>> 1);
33   else if(!bgm_on && sounds_on)
34     audio_signal = miss_out + exp_out;
35   else if(bgm_on && !sounds_on)
36     audio_signal = bgm_out;
37   else
38     audio_signal = 0;
39 end
40
41 volume_control vc (.vol_in(volume_control), .signal_in(audio_signal),
42                     ↪ .signal_out(vol_out));
43 pwm pwm (.clk_in(clk), .rst_in(reset), .level_in({~vol_out[7], vol_out[6:0]}),
44                     ↪ .pwm_out(pwm_val));
45
46 assign aud_pwm = pwm_val?1'bZ:1'b0;
47 endmodule
48
49
50
51 'default_nettype wire
52 //Volume Control
53 module volume_control (input [2:0] vol_in, input signed [7:0] signal_in, output
54                         ↪ logic signed[7:0] signal_out);
55   logic [2:0] shift;
56   assign shift = 3'd7 - vol_in;
57   assign signal_out = signal_in>>>shift;
58 endmodule

```

```
52
53 //PWM generator for audio generation!
54 module pwm (input clk_in, input rst_in, input [7:0] level_in, output logic
55   ↪ pwm_out);
56   logic [7:0] count;
57   assign pwm_out = count<level_in;
58   always_ff @(posedge clk_in)begin
59     if (rst_in)begin
60       count <= 8'b0;
61     end else begin
62       count <= count+8'b1;
63     end
64   end
65 endmodule
```

#### 7.1.4 audio\_miss.sv

```

1 module audio_miss #(BROM_SIZE = 10943,
2                     SAMPLE_RATE_PARAMETER = 10833) (
3     input logic clk_in,           // 65MHz system clock
4     input logic rst_in,          // 1 to reset to initial state
5     input logic play_audio,      // 1 when we want to start playing audio
6     output logic signed [7:0] data_out,    // 8-bit PCM data to headphone
7     output logic ready //1 when done with audio
8 );
9     logic[$clog2(SAMPLE_RATE_PARAMETER)+1:0] sample_counter;
10    assign sample_clk = (sample_counter == 0);
11    logic prev_play_audio;
12
13    always_ff @(posedge clk_in) begin
14        prev_play_audio <= play_audio;
15        if (rst_in || sample_counter == SAMPLE_RATE_PARAMETER - 1) begin
16            sample_counter <= 0;
17        end else begin
18            sample_counter <= sample_counter + 1;
19        end
20    end
21
22    logic [$clog2(BROM_SIZE) - 1 : 0] addr;
23    logic [$clog2(BROM_SIZE) - 1 : 0] max_addr;
24    logic [7:0] data_from_bram;
25    miss miss (.addr(a), .clka(clk_in), .douta(data_from_bram));
26
27    enum {IDLE, PLAYING} state;
28    always_ff @(posedge clk_in) begin
29        if(rst_in) begin
30            addr <= 0;
31            data_out <= 0;
32            state <= IDLE;
33            ready <= 1;
34        end
35        else begin
36            if(state == IDLE) begin
37                if(~prev_play_audio & play_audio) begin
38                    state <= PLAYING;
39                    addr <= 0;
40                    ready <= 0;
41                end
42            end
43            else if(state == PLAYING) begin
44                if(sample_clk) begin
45                    if(addr < BROM_SIZE) begin
46                        addr <= addr + 1;
47                        data_out <= {~data_from_bram[7], data_from_bram[6:0]};
48                        //convert from signed binary bram
49                    end
50                    else begin
51                        ready <= 1;
52                        state <= IDLE;
53                    end
54                end
55            end
56        end
57    end

```

58   **endmodule**

### 7.1.5 base\_board.sv

```

39      ||
40      (hcount_in == 512 + spacing/2 + SQ_SIZE*0 || hcount_in == 512 +
41       ↪ spacing/2 + SQ_SIZE*1 || hcount_in == 512 + spacing/2 + SQ_SIZE*2
42       ↪ ||
43       hcount_in == 512 + spacing/2 + SQ_SIZE*3 || hcount_in == 512 +
44       ↪ spacing/2 + SQ_SIZE*4 || hcount_in == 512 + spacing/2 + SQ_SIZE*5
45       ↪ ||
46       hcount_in == 512 + spacing/2 + SQ_SIZE*6 || hcount_in == 512 +
47       ↪ spacing/2 + SQ_SIZE*7 || hcount_in == 512 + spacing/2 + SQ_SIZE*8
48       ↪ ||
49       hcount_in == 512 + spacing/2 + SQ_SIZE*9 || hcount_in == 512 +
50       ↪ spacing/2 + SQ_SIZE*10 || hcount_in == 512 + spacing/2 +
51       ↪ SQ_SIZE*11 ||
52       hcount_in == 512 + spacing/2 + SQ_SIZE*12)))
53   )

54
55   pixel_out = 12'h000;
56
57   else if(((vcount_in >= y_offset && vcount_in <= SQ_SIZE*size + y_offset) &&
58            ↪ ((hcount_in >= x_offset && hcount_in <= SQ_SIZE*size + x_offset) ||
59             ↪ (hcount_in
60              ↪ >=
61              ↪ 512
62              ↪ +
63              ↪ spacing
64              ↪ &&
65              ↪ hcount_in
66              ↪ <=
67              ↪ 512
68              ↪ +
69              ↪ spacing
70              ↪ +
71              ↪ SQ_SIZE
72              ↪ begin
73                  pixel_out = water_pixel;
74                  //Not the board
75                  end else pixel_out = CLEAR_PIXEL;
76              end
77
78  endmodule
79
80
81  `default_nettype wire

```

### 7.1.6 blob.sv

```
1  `default_nettype none
2
3  module blob
4      #(parameter WIDTH = 64,           // default width: 64 pixels
5          HEIGHT = 64,             // default height: 64 pixels
6          COLOR = 12'hFFF) // default color: white
7      (input wire [10:0] x_in,hcount_in,
8       input wire [9:0] y_in,vcount_in,
9       output logic [11:0] pixel_out);
10
11     always_comb begin
12         if ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
13             (vcount_in >= y_in && vcount_in < (y_in+HEIGHT)))
14             pixel_out = COLOR;
15         else
16             pixel_out = 0;
17     end
18   endmodule
19
20 `default_nettype wire
```

### 7.1.7 boat2.sv

```
1 `timescale 1ns / 1ps
2 /////////////////////////////////
3 `default_nettype none
4
5 module boat2( input wire clk_in,
6                 input wire [10:0] hcount_in, x_in,
7                 input wire [9:0] vcount_in, y_in,
8                 input wire flipped,
9
10                output wire [11:0] pixel_out);
11
12    localparam w = 81;
13    localparam h = 40;
14
15    logic [7:0] image_bits, red_mapped;
16    logic [$clog2(w*h)-1:0] image_addr;
17
18    picture_blob #( .WIDTH(w), .HEIGHT(h)) b2 (.pixel_clk_in(clk_in),
19          ↪ .hcount_in(hcount_in), .vcount_in(vcount_in), .x_in(x_in), .y_in(y_in),
20          ↪ .flip(flipped),
21          ↪ .red_mapped(red_mapped),
22          ↪ .image_addr(image_addr),
23          ↪ .pixel_out(pixel_out));
24
25
26 endmodule
27
28 `default_nettype wire
```

### 7.1.8 boat3a.sv

```
1  `timescale 1ns / 1ps
2  ///////////////////////////////////////////////////////////////////
3  `default_nettype none
4
5  module boat3a(input wire clk_in, rst_in,
6                  input wire [10:0] hcount_in, x_in,
7                  input wire [9:0] vcount_in, y_in,
8                  input wire flipped,
9
10                 output wire [11:0] pixel_out);
11
12    localparam w = 120;
13    localparam h = 40;
14
15    logic [7:0] image_bits, red_mapped;
16    logic [$clog2(w*h)-1:0] image_addr;
17
18    picture_blob #(WIDTH(w), HEIGHT(h)) b3a (.pixel_clk_in(clk_in),
19                                              .hcount_in(hcount_in), .vcount_in(vcount_in), .x_in(x_in), .y_in(y_in),
20                                              .flip(flipped),
21                                              .red_mapped(red_mapped),
22                                              .image_addr(image_addr),
23                                              .pixel_out(pixel_out));
24
25
26  endmodule
27
28  `default_nettype wire
```

### 7.1.9 boat3b.sv

```
1  `timescale 1ns / 1ps
2  ///////////////////////////////////////////////////////////////////
3  `default_nettype none
4
5  module boat3b(input wire clk_in, rst_in,
6                  input wire [10:0] hcount_in, x_in,
7                  input wire [9:0] vcount_in, y_in,
8                  input wire flipped,
9
10                 output wire [11:0] pixel_out);
11
12    localparam w = 120;
13    localparam h = 40;
14
15    logic [7:0] image_bits, red_mapped;
16    logic [$clog2(w*h)-1:0] image_addr;
17
18    picture_blob #(WIDTH(w), HEIGHT(h)) b3b (.pixel_clk_in(clk_in),
19                                              .hcount_in(hcount_in), .vcount_in(vcount_in), .x_in(x_in), .y_in(y_in),
20                                              .flip(flipped),
21                                              .red_mapped(red_mapped),
22                                              .image_addr(image_addr),
23                                              .pixel_out(pixel_out));
24
25
26  endmodule
27
28  `default_nettype wire
```

### 7.1.10 boat4.sv

```
1  `timescale 1ns / 1ps
2  ///////////////////////////////////////////////////////////////////
3  `default_nettype none
4
5  module boat4(input wire clk_in, rst_in,
6                  input wire [10:0] hcount_in, x_in,
7                  input wire [9:0] vcount_in, y_in,
8                  input wire flipped,
9
10                 output wire [11:0] pixel_out);
11
12    localparam w = 161;
13    localparam h = 40;
14
15    logic [7:0] image_bits, red_mapped;
16    logic [$clog2(w*h)-1:0] image_addr;
17
18    picture_blob #(WIDTH(w), HEIGHT(h)) b4 (.pixel_clk_in(clk_in),
19                                              .hcount_in(hcount_in), .vcount_in(vcount_in), .x_in(x_in), .y_in(y_in),
20                                              .flip(flipped),
21                                              .red_mapped(red_mapped),
22                                              .image_addr(image_addr),
23                                              .pixel_out(pixel_out));
24
25
26  endmodule
27
28  `default_nettype wire
```

### 7.1.11 boat5.sv

```
1  `timescale 1ns / 1ps
2  ///////////////////////////////////////////////////////////////////
3  `default_nettype none
4
5  module boat5(input wire clk_in, rst_in,
6                  input wire [10:0] hcount_in, x_in,
7                  input wire [9:0] vcount_in, y_in,
8                  input wire flipped,
9
10                 output wire [11:0] pixel_out);
11
12    localparam w = 201;
13    localparam h = 41;
14
15    logic [7:0] image_bits, red_mapped;
16    logic [$clog2(w*h)-1:0] image_addr;
17
18    picture_blob #(WIDTH(w), HEIGHT(h)) b5 (.pixel_clk_in(clk_in),
19                                              .hcount_in(hcount_in), .vcount_in(vcount_in), .x_in(x_in), .y_in(y_in),
20                                              .flip(flipped),
21                                              .red_mapped(red_mapped),
22                                              .image_addr(image_addr),
23                                              .pixel_out(pixel_out));
24
25
26  `default_nettype wire
```

### 7.1.12 coverage.sv

```

1  `default_nettype none
2
3  module coverage #(parameter SQ_SIZE = 40)
4      (input wire clk_in, rst_in,
5       input wire [10:0] hcount_in, x_offset,
6       input wire [9:0] vcount_in, y_offset,
7       input wire [2:0] player_board [11:0][11:0],
8       input wire [2:0] opponent_board [11:0][11:0],
9       input wire [3:0] game_size,
10      output logic [11:0] pixel_out
11      );
12
13      logic [11:0] p_pixel;
14      logic [11:0] o_pixel;
15
16      logic [10:0] px;
17      logic [9:0] py;
18
19      logic [10:0] ox;
20      logic [9:0] oy;
21
22      localparam spacing = 62;
23
24      hv_to_xy #(.SQ_SIZE(SQ_SIZE)) player (.hcount_in(hcount_in),
25          ↪ .start_hcount(x_offset), .vcount_in(vcount_in),
26          ↪ .start_vcount(y_offset), .x_out(px), .y_out(py));
27      hv_to_xy #(.SQ_SIZE(SQ_SIZE)) opponent (.hcount_in(hcount_in),
28          ↪ .start_hcount(512+spacing/2), .vcount_in(vcount_in),
29          ↪ .start_vcount(y_offset), .x_out(ox), .y_out(oy));
30
31      hit_or_miss hmp (.clk_in(clk_in), .rst_in(rst_in), .x_in(px*SQ_SIZE +
32          ↪ x_offset), .y_in(py*SQ_SIZE + y_offset), .hcount_in(hcount_in),
33          ↪ .vcount_in(vcount_in), .hit(player_board[px][py]), .pixel_out(p_pixel));
34      hit_or_miss hmo (.clk_in(clk_in), .rst_in(rst_in),
35          ↪ .x_in(ox*SQ_SIZE+512+spacing/2), .y_in(oy*SQ_SIZE + y_offset),
36          ↪ .hcount_in(hcount_in), .vcount_in(vcount_in),
37          ↪ .hit(opponent_board[ox][oy]), .pixel_out(o_pixel));
38
39      always_ff @(posedge clk_in) begin
40          if (rst_in) begin
41              pixel_out <= 12'hFFF;
42          end else begin
43              pixel_out <= (px < game_size && py < game_size && p_pixel) ? p_pixel :
44                  (ox < game_size && oy < game_size && o_pixel) ? o_pixel :
45                      12'h000;
46          end
47      end
48  end
49
50  endmodule
51
52  `default_nettype wire

```

### 7.1.13 debouncer.sv

```
1 module debouncer (input wire reset, clk_in, noisy_in,
2                     output logic clean_out);
3
4     parameter DEBOUNCE_COUNT = 650000;
5     logic [19:0] count;
6     logic new_input;
7
8     always_ff @(posedge clk_in)
9         if (reset) begin
10             new_input <= noisy_in;
11             clean_out <= 0;
12             count <= 0; end
13         else if (noisy_in != new_input) begin new_input<=noisy_in; count <= 0; end
14         else if (count == DEBOUNCE_COUNT) clean_out <= new_input;
15         else count <= count+1;
16     endmodule
```

### 7.1.14 explosion.sv

```
1  `timescale 1ns / 1ps
2  ///////////////////////////////////////////////////////////////////
3  // Company:
4  // Engineer:
5  //
6  // Create Date: 11/23/2021 10:22:29 PM
7  // Design Name:
8  // Module Name: explosion
9  // Project Name:
10 // Target Devices:
11 // Tool Versions:
12 // Description:
13 //
14 // Dependencies:
15 //
16 // Revision:
17 // Revision 0.01 - File Created
18 // Additional Comments:
19 //
20 ///////////////////////////////////////////////////////////////////
21 `default_nettype none
22
23 module explosion( input wire clk_in, rst_in,
24                     input wire [10:0] hcount_in, x_in,
25                     input wire [9:0] vcount_in, y_in,
26
27                     output wire [11:0] pixel_out);
28
29     localparam w = 40;
30     localparam h = 40;
31
32     logic [7:0] red_mapped, green_mapped, blue_mapped, image_bits;
33     logic [$clog2(w*h)-1:0] image_addr;
34
35     picture_blob #(.WIDTH(w), .HEIGHT(h), .GREYSCALE(1'b0)) e1
36         ↪ (.pixel_clk_in(clk_in), .hcount_in(hcount_in), .vcount_in(vcount_in),
37             .x_in(x_in),
38             ↪ .y_in(y_in),
39             ↪ .flip(1'b0),
40             ↪ .red_mapped(red_mapped),
41             .green_mapped(green_mapped),
42             ↪ .blue_mapped(blue_mapped),
43             ↪ .image_addr(image_addr),
44             .pixel_out(pixel_out));
45
46     explosion1_image e1_image(.clka(clk_in), .addr(a),
47                             ↪ .dout(d));
48     explosion1_red e1_r(.clka(clk_in), .addr(a), .dout(red_mapped));
49     explosion1_blue e1_b(.clka(clk_in), .addr(a), .dout(blue_mapped));
50     explosion1_green e1_g(.clka(clk_in), .addr(a),
51                           ↪ .dout(green_mapped));
52
53 endmodule
54
55 `default_nettype wire
```

### 7.1.15 fsm\_state\_pkg.sv

```
1 package fsm_state; // package name
2     typedef enum {START_MENU, SET_GAME_PARAMETERS,
3                     CONNECTION_WAITING, CONNECTION_SENDING,
4                     SETUP_BOARD, SETUP_WAITING, PLAY_MOVE,
5                     MOVE_WAITING, END_GAME_WIN, END_GAME_LOSE,
6                     PLAY_STALL, PLAY_STALL_2, WAIT_MOVE_WAITING,
7                     WAIT_STALL, WAIT_STALL_2, RESET_GAME, GEN_ROCKS_RECEIVER,
8                     SINGLEPLAYER_START, SINGLEPLAYER_RANDOM_STALL,
9                     SINGLEPLAYER_PLACE_CPU_SHIPS, SINGLEPLAYER_PLACE_SHIPS,
10                    SINGLEPLAYER_PLAY_MOVE, SINGLEPLAYER_UPDATE_SUNK_SHIP,
11                    SINGLEPLAYER_PLAY_MOVE_STALL, SINGLEPLAYER_OPPONENT_PLAY_MOVE}
12                         ↪ game_state;
13
14     typedef struct {
15         logic[3:0] x;
16         logic[3:0] y;
17         logic[3:0] length;
18         logic orientation; //1 for horizontal, 0 for vertical
19         logic[3:0] num_hits; //counter for number of ship squares hit
20     } ship;
21
22     typedef ship ship_array[5];
23 endpackage
```

### 7.1.16 game\_state\_controller.sv

```
1  `default_nettype none
2  import fsm_state::*;
3
4  module game_state_controller(input wire reset, clk,
5                                input wire btnd, btnc, btnu, btnl, btnr,
6                                input wire rotate_ship, input wire [3:0]
7                                ↪ game_size_switch,
8                                input wire rx_in, output logic tx_out,
9                                input wire [2:0] volume_control,
10                               input wire sounds_on, bgm_on,
11                               input wire bgm_test,
12                               input wire [11:0] mousex,
13                               input wire [11:0] mousey,
14                               input wire [2:0] num_rocks,
15                               input wire mouse_enable,
16                               input wire hard_mode,
17                               output logic [3:0] vga_r, vga_g, vga_b,
18                               output logic vga_hs, vga_vs,
19                               output logic aud_pwm, aud_sd,
20                               output logic [31:0] debug);
21
22   logic [15:0] rx_data;
23   logic [15:0] tx_data;
24   logic rx_valid;
25   logic tx_valid;
26   logic rx_ready;
27   logic tx_ready;
28
29   game_state state;
30   ship_array ships;
31   logic[2:0] curr_ship_ctrl; //which ship is being placed
32   logic placeable_out;
33   logic[3:0] x, y, game_size; //x, y coordinates and game size
34
35   logic [7:0] debug_g;
36   assign debug = {btnc, btnu, btnr, btl, btnd, x, y, 1'b0, curr_ship_ctrl,
37   ↪ 3'b000, state[4:0]}; //for debugging on 8hex display
38
39   //keeping track of board state
40   logic[2:0] opponent_board [11:0][11:0];
41   logic[2:0] player_board [11:0][11:0];
42
43   logic play_explosion;
44   logic play_miss;
45   logic play_bgm;
46
47   //reset logic for after game ends; passes normal sw[15] if game not over
48   logic reset_in;
49   logic game_end;
50   assign game_end = state == RESET_GAME;
51   parameter RESET_CLK_COUNT = 650000;
52   int reset_ctr;
53   always_ff @(posedge clk) begin
54     if(reset) begin
55       reset_in <= 1;
56       reset_ctr <= RESET_CLK_COUNT;
57     end
58   end
```

```

56     else if (game_end) begin
57         reset_in <= 1;
58         reset_ctr <= 0;
59     end
60     else if(reset_ctr >= RESET_CLK_COUNT)
61         reset_in <= 0;
62     else reset_ctr <= reset_ctr + 1;
63 end
64
65
66 rx rx_module (.clk_in(clk), .rst_in(reset_in), .rx_in(rx_in),
67                 .ready_in(rx_ready), .valid_out(rx_valid), .data_out(rx_data));
68 tx tx_module (.clk_in(clk), .rst_in(reset_in), .tx_out(tx_out),
69                 .ready_out(tx_ready), .valid_in(tx_valid), .data_in(tx_data));
70
71 user_input ui(.reset(reset_in), .clk(clk), .btnd(btnd),
72                 .btnu(btnu), .btnr(btnr), .btnl(btnl),
73                 .size(game_size), .x(x), .y(y), .rotate(rotate_ship),
74                 .ship_num(curr_ship_ctr), .ships(ships), .state(state),
75                 .mousex(mousex), .mousey(mousey), .mouse_enable(mouse_enable));
76
77 game_state_fsm fsm(.reset(reset_in), .clk(clk), .btnc(btnc), .x(x), .y(y),
78                     ↪ .rotate(rotate_ship), //user inputs
79                     .game_size_switch(game_size_switch),
80                     ↪ .hard_mode_in(hard_mode), //game settings input
81                     .game_size(game_size), .ship_state(curr_ship_ctr), //game
82                     ↪ settings output
83                     .ships(ships), .state(state), .num_rocks_in(num_rocks),
84                     .tx_data(tx_data), .tx_valid(tx_valid),
85                     ↪ .tx_ready(tx_ready), //communication
86                     .rx_data(rx_data), .rx_ready(rx_ready),
87                     ↪ .rx_valid(rx_valid), //communication
88                     .player_board(player_board),
89                     ↪ .opponent_board(opponent_board),
90                     .play_bgm(play_bgm), .play_miss(play_miss),
91                     ↪ .play_explosion(play_explosion));
92
93 graphics_prep gp (.clk_65mhz(clk), .rst_in(reset_in), .ships(ships),
94                     ↪ .game_size(game_size), .state_in(state),
95                     .x_in(x), .y_in(y), .vga_r(vga_r), .vga_g(vga_g),
96                     ↪ .vga_b(vga_b), .vga_hs(vga_hs), .vga_vs(vga_vs),
97                     .opponent_board(opponent_board),
98                     ↪ .player_board(player_board), .debug(debug_g),
99                     ↪ .ship_state(curr_ship_ctr),
100                    .mousex(mousex), .mousey(mousey),
101                    ↪ .mouse_enable(mouse_enable));
102
103 audio_gen audio (.clk(clk), .reset(reset_in), .bgm_on(bgm_on),
104                     ↪ .sounds_on(sounds_on), .play_explosion(play_explosion),
105                     ↪ .play_miss(play_miss),
106                     .play_bgm(play_bgm), .volume_control(volume_control),
107                     ↪ .aud_pwm(aud_pwm), .aud_sd(aud_sd));
108
109 endmodule
110
111 'default_nettype wire

```

### 7.1.17 game\_state.fsm.sv

```

1  `default_nettype none
2  import fsm_state::*;
3
4  module game_state_fsm(input wire reset, clk, btnc,
5                         input wire[3:0] x, y, game_size_switch,
6                         input wire rotate,
7                         input wire[15:0] rx_data,
8                         input wire rx_valid,
9                         input wire hard_mode_in,
10                        input wire[2:0] num_rocks_in,
11                        output logic rx_ready,
12                        output logic tx_valid,
13                        output logic[15:0] tx_data,
14                        input wire tx_ready,
15                        output game_state state,
16                        output logic[3:0] game_size,
17                        output logic[2:0] ship_state,
18                        output ship_array ships,
19                        output logic[2:0] player_board [11:0][11:0],
20                        output logic[2:0] opponent_board [11:0][11:0],
21                        output logic play_explosion, play_miss, play_bgm);
22
23  logic old_btnc;
24  logic start_rng;
25  // opponent_board = which positions have been fired at
26  // opponent_board[x][y] = 00 if not fired
27  // = 01 if HIT
28  // = 10 if MISS
29  // = 11 if ROCK
30  // = 100 if SUNK
31  // same for player_board
32
33  logic placeable_out;
34  logic rocks_out;
35  logic[7:0] player_rocks [7:0];
36  logic[2:0] num_rocks;
37  logic[7:0] random_rock;
38  logic start_rock_gen;
39  logic done_rock_gen;
40  rock_randomizer rock_randomizer(.reset(reset), .clk(clk),
41                                 .rng_input(rng_out), .start(start_rock_gen), .done(done_rock_gen),
42                                 .player_rocks_out(player_rocks),
43                                 .cpu_rocks_out(cpu_rocks));
44  rock_coords rocks_coord_checker(.x(rx_data[7:4]), .y(rx_data[3:0]),
45                                 .rocks_in(player_rocks),
46                                 .placeable(rocks_out), .num_rocks(num_rocks));
47  placeable coord_checker(.x(x), .y(y), .ship_num(ship_state),
48                          .rocks_in(player_rocks), // combinational check if ship is placeable at
49                          .that (x,y)
50                          .ships(ships), .check_one_coord(1'b0),
51                          .placeable(placeable_out),
52                          .num_rocks(num_rocks));
53
54  logic ship_out [4:0]; //1 if the (x,y) position is a hit, 0 otherwise.
55  // used when opponent is attacking to check positions
56  ship_coords ship0_coord_checker(.x(rx_data[7:4]), .y(rx_data[3:0]),
57                                 .ships(ships), .ship_num(4'd0), .ship_out(ship_out[0]));

```

```

50     ship_coords ship1_coord_checker(.x(rx_data[7:4]), .y(rx_data[3:0]),
51         ↪ .ships(ships), .ship_num(4'd1), .ship_out(ship_out[1]));
52     ship_coords ship2_coord_checker(.x(rx_data[7:4]), .y(rx_data[3:0]),
53         ↪ .ships(ships), .ship_num(4'd2), .ship_out(ship_out[2]));
54     ship_coords ship3_coord_checker(.x(rx_data[7:4]), .y(rx_data[3:0]),
55         ↪ .ships(ships), .ship_num(4'd3), .ship_out(ship_out[3]));
56     ship_coords ship4_coord_checker(.x(rx_data[7:4]), .y(rx_data[3:0]),
57         ↪ .ships(ships), .ship_num(4'd4), .ship_out(ship_out[4]));
58
59     logic[4:0] num_player_squares_hit;
60     logic[4:0] num_opponent_squares_hit;
61
62     logic[3:0] rng_out [31:0];
63     logic[31:0] unbounded_out;
64     logic[31:0] stall_ctr;
65     bounded_rng rng(.reset(reset), .clk(clk), .start(start_rng),
66                     .initial_seed(32'hDEADBEEF), .num_max(game_size), //random
67                         ↪ RNG starting seed, randomized by clock cycle
68                     .numbers_out(rng_out), .unbounded_out(unbounded_out));
69
70     ship_array cpu_ships;
71     logic[7:0] cpu_rocks[7:0];
72     logic[2:0] cpu_ship_state;
73     logic cpu_placeable;
74     logic cpu_rocks_out;
75     logic[2:0] cpu_sunk_ship; //ship number or
76                           //7 for no sunk ship
77     logic[7:0] sunk_ship_coords;
78     logic[2:0] sunk_ship_length;
79     logic hit_player_ship;
80     logic hard_mode;
81     logic[1:0] num_cpu_hits;
82     rock_coords cpu_rocks_coord_checker(.x(x), .y(y), .rocks_in(cpu_rocks),
83                                         ↪ .num_rocks(num_rocks), .placeable(cpu_rocks_out));
84     placeable cpu_coord_checker(.x(rng_out[0][3:0]), .y(rng_out[1][3:0]),
85                                 ↪ .ship_num(cpu_ship_state), .check_one_coord(1'b0),
86                                         .rocks_in(cpu_rocks), .num_rocks(num_rocks),
87                                         ↪ .ships(cpu_ships),
88                                         ↪ .placeable(cpu_placeable));
89
90     placeable cpu_hit_player(.x(rng_out[0][3:0]), .y(rng_out[1][3:0]),
91                             ↪ .ship_num(3'd5), .check_one_coord(1'b1),
92                                         .rocks_in(), .num_rocks(0), .ships(ships),
93                                         ↪ .placeable(hit_player_ship));
94                                         //check if CPU hits ship player
95                                         //set rocks to none so only checks if we hit ship
96
97     logic cpu_ship_out [4:0];
98     ship_coords cpu_ship0_coord_checker(.x(x), .y(y), .ships(cpu_ships),
99                                         ↪ .ship_num(4'd0), .ship_out(cpu_ship_out[0]));
100    ship_coords cpu_ship1_coord_checker(.x(x), .y(y), .ships(cpu_ships),
101                                         ↪ .ship_num(4'd1), .ship_out(cpu_ship_out[1]));
102    ship_coords cpu_ship2_coord_checker(.x(x), .y(y), .ships(cpu_ships),
103                                         ↪ .ship_num(4'd2), .ship_out(cpu_ship_out[2]));
104    ship_coords cpu_ship3_coord_checker(.x(x), .y(y), .ships(cpu_ships),
105                                         ↪ .ship_num(4'd3), .ship_out(cpu_ship_out[3]));
106    ship_coords cpu_ship4_coord_checker(.x(x), .y(y), .ships(cpu_ships),
107                                         ↪ .ship_num(4'd4), .ship_out(cpu_ship_out[4]));

```

```

93    //generate random rocks for CPU board
94    //generate random positions for CPU ships
95    //generate random coordinate for CPU attack
96    always_ff @(posedge clk) begin
97        if(reset) begin
98            state <= START_MENU; //if reset, then set state to start menu
99            ship_state <= 0;
100           game_size <= 12;
101           old_btnc <= 1;
102           num_opponent_squares_hit <= 0;
103           num_player_squares_hit <= 0;
104           num_rocks <= 0;
105           start_rock_gen <= 0;
106           ships <= '{'{0, 0, 4'd2, 0, 0}, // (x, y, length, orientation,
107                         ↪ num_hits)
108                         '{13, 1, 4'd3, 0, 0},
109                         '{13, 5, 4'd3, 0, 0},
110                         '{14, 0, 4'd4, 0, 0},
111                         '{14, 5, 4'd5, 0, 0}};
112           rx_ready <= 1'b0;
113           tx_data <= 16'b0;
114           tx_valid <= 1'b0;
115           for(int i = 0; i < 12; i++) begin
116               for(int j = 0; j < 12; j++) begin
117                   player_board[i][j] <= 3'b010; //show all water on player
118                         ↪ board
119                   opponent_board[i][j] <= 3'b000;
120               end
121           end
122           cpu_ships <= '{'{0, 0, 4'd2, 0, 0},
123                         '{0, 0, 4'd3, 0, 0},
124                         '{0, 0, 4'd3, 0, 0},
125                         '{0, 0, 4'd4, 0, 0},
126                         '{0, 0, 4'd5, 0, 0}};
127           cpu_ship_state <= 0;
128           stall_ctrl <= 0;
129           start_rng <= 0;
130           cpu_sunk_ship <= 0;
131           sunk_ship_coords <= 0;
132           sunk_ship_length <= 0;
133       end
134   else begin
135       old_btnc <= btnc;
136       case(state)
137           START_MENU: begin
138               start_rng <= 1;
139               if(~old_btnc & btnc) begin
140                   play_bgm <= 1'b1;
141                   case(y)
142                       0: //first menu option: become game host; set up menu
143                           ↪ options
144                           state <= SET_GAME_PARAMETERS;
145                       1: //second menu option: look for open game hosted by
146                           ↪ other FPGA to join
147                           begin
148                               state <= CONNECTION_SENDING;
149                               tx_valid <= 1'b1;
150                               tx_data <= 16'hFFFF;

```

```

148
149         end
150     2: //third menu option: go to singleplayer
151         state <= SINGLEPLAYER_START;
152     endcase
153 end
154 SET_GAME_PARAMETERS: begin
155     play_bgm <= 1'b0;
156     game_size <= game_size_switch + 4'd9;
157     num_rocks <= num_rocks_in;
158     start_rock_gen <= 1;
159     state <= CONNECTION_WAITING;
160 end
161 SINGLEPLAYER_START: begin
162     play_bgm <= 1'b0;
163     game_size <= game_size_switch + 4'd9;
164     hard_mode <= hard_mode_in;
165     num_rocks <= num_rocks_in;
166     start_rock_gen <= 1;
167     state <= SINGLEPLAYER_RANDOM_STALL;
168     stall_ctr <= 0;
169 end
170 SINGLEPLAYER_RANDOM_STALL: begin
171     if(done_rock_gen) begin
172         for(int i = 0; i < num_rocks; i++) begin //set rocks
173             player_board[player_rocks[i][3:0]][player_rocks[i][7:4]]
174                 <= 3'b011;
175         end //set random CPU ship orientation
176         for(int i = 0; i < 5; i++) begin
177             cpu_ships[i].orientation <= unbounded_out[i];
178         end
179         state <= SINGLEPLAYER_PLACE_CPU_SHIPS;
180     end
181 end
182 SINGLEPLAYER_PLACE_CPU_SHIPS: begin
183     stall_ctr <= stall_ctr + 1;
184     if(stall_ctr > 500) begin
185         if(rng_out[0][3:0] +
186             & (cpu_ships[cpu_ship_state].orientation == 0 ?
187                 cpu_ships[cpu_ship_state].length - 1
188                 && rng_out[1][3:0] +
189                     & (cpu_ships[cpu_ship_state].orientation == 1 ?
190                         cpu_ships[cpu_ship_state].length - 1
191                         && cpu_placeable) begin //if ship placeable, place ship
192                             cpu_ships[cpu_ship_state].x <= rng_out[0][3:0];
193                             cpu_ships[cpu_ship_state].y <= rng_out[1][3:0];
194                             cpu_ship_state <= cpu_ship_state + 1;
195                         end
196                         stall_ctr <= 0;
197                     end
198                     if(cpu_ship_state == 5)
199                         state <= SINGLEPLAYER_PLACE_SHIPS;
200                 end
201 SINGLEPLAYER_PLACE_SHIPS: begin
202             ships[ship_state].x <= x;
203             ships[ship_state].y <= y;
204             ships[ship_state].orientation <= rotate;

```

```

202     if((btnc & ~old_btnc) && placeable_out) begin //place ship on
203         ↪ center button, increment ship count if ship is in fact
204         ↪ placeable
205         ship_state <= ship_state + 1;
206     end else if(ship_state == 5) begin
207         state <= SINGLEPLAYER_PLAY_MOVE_STALL;
208     end
209 end
210
211 SINGLEPLAYER_PLAY_MOVE_STALL : state <= SINGLEPLAYER_PLAY_MOVE;
212
213 SINGLEPLAYER_PLAY_MOVE: begin
214     if(btnc & ~old_btnc)
215         if (opponent_board[x][y] == 3'b000) begin
216             opponent_board[x][y] <= 3'b010; //set to miss
217             cpu_sunk_ship <= 3'd7;
218             play_miss <= 1;
219
220             //check for hits
221             for(int i = 0; i < 5; i++) begin
222                 if(cpu_ship_out[i]) begin
223                     play_miss <= 0;
224                     play_explosion <= 1;
225                     cpu_ships[i].num_hits <=
226                         ↪ cpu_ships[i].num_hits + 1;
227                     num_opponent_squares_hit <=
228                         ↪ num_opponent_squares_hit + 1;
229                     if(cpu_ships[i].num_hits + 1 ==
230                         ↪ cpu_ships[i].length) begin //ship sunk
231                         cpu_sunk_ship <= i;
232                         sunk_ship_coords <= {cpu_ships[i].x,
233                             ↪ cpu_ships[i].y};
234                         sunk_ship_length <= cpu_ships[i].length;
235                     end
236                     else //only hit but no sink
237                         opponent_board[x][y] <= 3'b001;
238                 end
239             end
240             //check if rock position
241             if(!cpu_rocks_out) begin
242                 opponent_board[x][y] <= 3'b011;
243                 play_miss <= 1;
244             end
245             state <= SINGLEPLAYER_UPDATE_SUNK_SHIP;
246         end
247     end
248 end
249
250 SINGLEPLAYER_UPDATE_SUNK_SHIP : begin
251     play_miss <= 0;
252     play_explosion <= 0;
253     if(cpu_sunk_ship != 3'd7) begin
254         for(int i = 0; i < sunk_ship_length; i++) begin
255             if(cpu_ships[cpu_sunk_ship].orientation == 0)
256                 ↪ //horizontal ship
257                 opponent_board[sunk_ship_coords[7:4] +
258                     ↪ i][sunk_ship_coords[3:0]] <= 3'b100;
259             else
260                 opponent_board[sunk_ship_coords[7:4]][sunk_ship_coords[3:0]
261                     ↪ + i] <= 3'b100;
262         end
263         cpu_sunk_ship <= 3'd7;
264     end

```

```

252     if(num_opponent_squares_hit == 17)
253         state <= END_GAME_WIN;
254     else state <= SINGLEPLAYER OPPONENT_PLAY_MOVE;
255     stall_ctrl <= 0;
256     num_cpu_hits <= 0;
257 end
258 SINGLEPLAYER OPPONENT_PLAY_MOVE: begin
259     stall_ctrl <= stall_ctrl + 1;
260     if(stall_ctrl > 20000000) begin //add artificial delay
261         if(player_board[rng_out[0][3:0]][rng_out[1][3:0]] != 
262             ↪ 3'b001) begin
263             player_board[rng_out[0][3:0]][rng_out[1][3:0]] <=
264                 ↪ 3'b001;
265             stall_ctrl <= 0;
266             num_cpu_hits <= num_cpu_hits + 1;
267             if(!hit_player_ship) begin
268                 num_player_squares_hit <= num_player_squares_hit
269                     ↪ + 1;
270                 if(num_player_squares_hit == 16)
271                     state <= END_GAME_LOSE;
272                 else if(num_cpu_hits == 2 || !hard_mode)
273                     state <= SINGLEPLAYER_PLAY_MOVE;
274             end
275         end
276     end
277 CONNECTION_WAITING: begin
278     if(btnc & ~old_btnc)
279         state <= START_MENU;
280     else if(rx_valid && rx_data == 16'hFFFF && done_rock_gen)
281         ↪ begin
282         rx_ready <= 1'b1;
283         tx_data <= {game_size, num_rocks, 9'b0};
284         tx_valid <= 1'b1;
285         for(int i = 0; i < num_rocks; i++)
286             player_board[player_rocks[i][3:0]][player_rocks[i][7:4]]
287                 ↪ <= 3'b011;
288             //wait for UART START signal 16'hFFFF
289             //send game params {game_size[3:0], rock_num[2:0], 9'b0}
290             state <= SETUP_BOARD;
291         end
292     end //TODO: Integration with RX/TX
293 CONNECTION_SENDING: begin
294     play_bgm <= 1'b0;
295     tx_valid <= 1'b0;
296     if(btnc & ~old_btnc)
297         state <= START_MENU;
298     else if(rx_valid) begin
299         rx_ready <= 1'b1;
300         game_size <= rx_data[15:12];
301         num_rocks <= rx_data[11:9];
302         start_rock_gen <= 1;
303         state <= GEN_ROCKS_RECEIVER;
304     end
305     //send UART START signal 16'FFFF
306     //wait for game params signal

```

```

306          //set game params equal to those in signal
307      end
308      GEN_ROCKS_RECEIVER: begin
309          if(done_rock_gen) begin
310              for(int i = 0; i < num_rocks; i++)
311                  player_board[player_rocks[i][3:0]][player_rocks[i][7:4]]
312                      ↪ <= 3'b011;
313                  state <= SETUP_BOARD;
314          end
315      end
316      SETUP_BOARD: begin
317          ships[ship_state].x <= x;
318          ships[ship_state].y <= y;
319          ships[ship_state].orientation <= rotate;
320
321          tx_valid <= 1'b0;
322          rx_ready <= 1'b0;
323
324          if((btnc & ~old_btnc) && placeable_out) begin //place ship on
325              ↪ center button, increment ship count if ship is in fact
326              ↪ placeable
327              ship_state <= ship_state + 1;
328          end else if(ship_state == 5) begin
329              tx_valid <= 1'b1;
330              tx_data <= 16'hFF00;
331              if (rx_data == 16'hFF00 && rx_valid) begin
332                  rx_ready <= 1'b1;
333                  state <= WAIT_MOVE_WAITING;
334              end else begin
335                  state <= SETUP_WAITING;
336              end
337          end
338      end
339      SETUP_WAITING: begin
340          //wait for DONE signal. Once received, move to PLAY_MOVE
341          tx_valid <= 1'b0;
342          rx_ready <= 1'b0;
343          if (rx_data == 16'hFF00 && rx_valid) begin
344              rx_ready <= 1'b1;
345              state <= PLAY_STALL;
346          end
347      end
348      PLAY_STALL: begin
349          tx_valid <= 0;
350          rx_ready <= 1'b0;
351          if(num_player_squares_hit == 17) //all ships hit
352              state <= END_GAME_LOSE;
353          else state <= PLAY_STALL_2;
354      end
355      PLAY_STALL_2: begin
356          state <= PLAY_MOVE;
357      end
358      PLAY_MOVE: begin
359          rx_ready <= 0;
360          tx_valid <= 0;
361          if(~old_btnc & btnc) begin
362              if (opponent_board[x][y] == 3'b000) begin
363                  tx_data <= {8'b0, x, y};
364                  tx_valid <= 1'b1;

```

```

362         end
363         //check map if possible to fire
364         //fire at position (x,y)
365         //send (x,y) on TX
366         //interpret results (specified in MOVE_WAITING state)
367         //play audio
368         //update ship_positions on graphics
369         //check if the number of squares actually hit is == (2 +
370             ↪ 3 + 3 + 4 + 5), if so, then go to END_GAME_WIN state
371             ↪ //otherwise, go to MOVE_WAITING state
372     end
373     else if(rx_valid) begin
374         rx_ready <= 1;
375         if(rx_data == 16'b0) begin                         //miss
376             opponent_board[x][y] <= 3'b010;
377             play_miss <= 1;
378         end else if(rx_data == 16'b1000_0000_0000_0000) begin
379             ↪ //hit
380             opponent_board[x][y] <= 3'b001;
381             num_opponent_squares_hit <= num_opponent_squares_hit
382                 ↪ + 1;
383             play_explosion <= 1;
384         end
385         else if(rx_data == 16'b0100_0000_0000_0000) begin //rock
386             opponent_board[x][y] <= 3'b011;
387             play_miss <= 1;
388         end else if(rx_data[15:14] == 2'b11) begin
389             num_opponent_squares_hit <= num_opponent_squares_hit
390                 ↪ + 1;
391             play_explosion <= 1;
392             for(int i = 0; i < ships[rx_data[12:10]].length; i++)
393                 ↪ begin
394                     if(rx_data[13] == 0) //horizontal ship
395                         opponent_board[rx_data[7:4] +
396                             ↪ i][rx_data[3:0]] <= 3'b100;
397                     else
398                         opponent_board[rx_data[7:4]][rx_data[3:0] +
399                             ↪ i] <= 3'b100;
400                 end
401             end
402             state <= WAIT_STALL;
403         end
404     end
405     WAIT_MOVE_WAITING: begin
406         tx_valid <= 0;
407         rx_ready <= 1'b0;
408         state <= WAIT_STALL;
409     end
410     WAIT_STALL: begin
411         tx_valid <= 0;
412         rx_ready <= 1'b0;
413         play_miss <= 0;
414         play_explosion <= 0;
415         if(num_opponent_squares_hit == 17) //all ships hit
416             state <= END_GAME_WIN;
417         else state <= WAIT_STALL_2;
418     end
419     WAIT_STALL_2: begin
420         state <= MOVE_WAITING;

```

```

414     end
415     MOVE_WAITING: begin
416         //wait for (x,y) position of hit
417         //send on RX/TX:
418         //If missed: send 00000000
419         //If hit a ship only: send 10000000
420         //If hit and sunk a ship: send 11[rotate][# of ship sunk (3
421             ↪ bits)]00, then (x,y) coordinate of sunk ship
422         //check if the number of squares hit is == (2 + 3 + 3 + 4 +
423             ↪ 5), if so, then go to END_GAME_LOSE state
424         //update hit_positions (and graphic display)
425         //otherwise, update screen, move to PLAY_MOVE screen
426         rx_ready <= 0;
427         tx_valid <= 0;
428         if(rx_valid) begin
429             tx_valid <= 1'b1;
430             rx_ready <= 1'b1;
431             tx_data <= 16'b0; //default: send MISS
432             //update player board to show HIT
433             player_board[rx_data[7:4]][rx_data[3:0]] <= 3'b001;
434
435             //check for hits
436             for(int i = 0; i < 5; i++) begin
437                 if(ship_out[i]) begin
438                     ships[i].num_hits <= ships[i].num_hits + 1;
439                     num_player_squares_hit <= num_player_squares_hit
440                         ↪ + 1;
441                     if(ships[i].num_hits + 1 == ships[i].length)
442                         ↪ begin //ship sunk
443                             tx_data <= {2'b11, ships[i].orientation,
444                                 ↪ i[2:0], 2'b00, ships[i].x, ships[i].y};
445                         end
446                     else //only hit but no sink
447                         tx_data <= 16'b1000_0000_0000_0000;
448
449                 end
450             end
451             if(!rocks_out) begin
452                 tx_data <= 16'b0100_0000_0000_0000;
453             end
454             state <= PLAY_STALL;
455         end
456     end
457     END_GAME_WIN: if(~old_btnc & btnc) state <= RESET_GAME; //exit
458         ↪ from END_GAME state to start menu with btnc press, show a
459         ↪ green screen over board
460     END_GAME_LOSE: if(~old_btnc & btnc) state <= RESET_GAME; //exit
461         ↪ from END_GAME state to start menu with btnc press, show a
462         ↪ red screen over board
463     endcase
464     end
465   end
466 endmodule
467
468 'default_nettype wire

```

### 7.1.18 graphics\_prep.sv

```
1 `timescale 1ns / 1ps
2 ///////////////////////////////////////////////////////////////////
3 `default_nettype none
4 import fsm_state::*;
5
6 parameter CLEAR_PIXEL = 12'h0F0;
7
8 module graphics_prep( input wire clk_65mhz, rst_in,
9                         input game_state state_in,
10                        input wire [2:0] ship_state,
11                        input ship_array ships,
12                        input wire [3:0] x_in, y_in,
13                        input wire [3:0] game_size,
14                        input wire [2:0] player_board [11:0][11:0],
15                        input wire [2:0] opponent_board [11:0][11:0],
16                        input wire [11:0] mousex,
17                        input wire [11:0] mousey,
18                        input wire mouse_enable,
19                        output logic [3:0] vga_r,
20                        output logic [3:0] vga_g,
21                        output logic [3:0] vga_b,
22                        output logic vga_hs,
23                        output logic vga_vs,
24                        output logic [7:0] debug
25 );
26 localparam [5:0] sq_sz = 40;
27 localparam spacing = 62;
28
29 logic [10:0] hcount;
30 logic [9:0] vcount;
31 logic vsync, hsync, blank;
32 logic [5:0] vsync_q;
33 logic [5:0] hsync_q;
34 logic [5:0] blank_q;
35
36 logic [11:0] pixel_out;
37 logic [11:0] board_pixel;
38 logic [11:0] board_pixel_q [4:0];
39 logic [11:0] water_pixel;
40 logic [11:0] b2_pixel;
41 logic [11:0] b3a_pixel;
42 logic [11:0] b3b_pixel;
43 logic [11:0] b4_pixel;
44 logic [11:0] b5_pixel;
45 logic [11:0] cursor_pixel;
46 logic [11:0] cursor_pixel_q [3:0];
47 logic [11:0] coverage_pixel_q [2:0];
48 logic [11:0] coverage_pixel;
49 logic [11:0] menu_pixel_q [2:0];
50 logic [11:0] menu_pixel;
51
52 logic [10:0] x_offset;
53 logic [9:0] y_offset;
54
55 assign x_offset = (game_size == 4'd9) ? 10'd123 :
56                               (game_size == 4'd10) ? 10'd082 :
57                               (game_size == 4'd11) ? 10'd041 :
```

```

58             (game_size == 4'd12) ? 10'd000 :
59             10'b0;
60
61 assign y_offset = (game_size == 4'd12) ? 9'd137 :
62             (game_size == 4'd11) ? 9'd158 :
63             (game_size == 4'd10) ? 9'd179 :
64             (game_size == 4'd9) ? 9'd199 :
65             10'b0;
66
67 xvga xvga(.vclock_in(clk_65mhz), .hcount_out(hcount), .vcount_out(vcount),
68             .vsync_out(vsync), .hsync_out(hsync), .blank_out(blank));
69
70 base_board #(.SQ_SIZE(sq_sz)) bb (.clk_in(clk_65mhz), .rst_in(rst_in),
71             ↪ .hcount_in(hcount), .vcount_in(vcount),
72             .pixel_out(board_pixel),
73             ↪ .size(game_size),
74             ↪ .x_offset(x_offset),
75             ↪ .y_offset(y_offset));
76
76 boat2 b2 (.clk_in(clk_65mhz), .x_in(ships[0].x*sq_sz + x_offset),
77             ↪ .y_in(ships[0].y*sq_sz + y_offset),
78             .vcount_in(vcount), .hcount_in(hcount),
79             ↪ .flipped(ships[0].orientation), .pixel_out(b2_pixel));
80
81 boat3a b3a (.clk_in(clk_65mhz), .x_in(ships[1].x*sq_sz + x_offset),
82             ↪ .y_in(ships[1].y*sq_sz + y_offset),
83             .vcount_in(vcount), .hcount_in(hcount),
84             ↪ .flipped(ships[1].orientation), .pixel_out(b3a_pixel));
85
85 boat3b b3b (.clk_in(clk_65mhz), .x_in(ships[2].x*sq_sz + x_offset),
86             ↪ .y_in(ships[2].y*sq_sz + y_offset),
87             .vcount_in(vcount), .hcount_in(hcount),
88             ↪ .flipped(ships[2].orientation), .pixel_out(b3b_pixel));
89
90 boat4 b4 (.clk_in(clk_65mhz), .x_in(ships[3].x*sq_sz + x_offset),
91             ↪ .y_in(ships[3].y*sq_sz + y_offset),
92             .vcount_in(vcount), .hcount_in(hcount),
93             ↪ .flipped(ships[3].orientation), .pixel_out(b4_pixel));
94
94 boat5 b5 (.clk_in(clk_65mhz), .x_in(ships[4].x*sq_sz + x_offset),
95             ↪ .y_in(ships[4].y*sq_sz + y_offset),
96             .vcount_in(vcount), .hcount_in(hcount),
97             ↪ .flipped(ships[4].orientation), .pixel_out(b5_pixel));
98
98 select_square #(.SQ_SIZE(sq_sz)) cursor (.x_in(x_in*sq_sz + 512 + spacing/2),
99             ↪ .y_in(y_in*sq_sz + y_offset), .hcount_in(hcount),
100             .vcount_in(vcount),
101             ↪ .pixel_out(cursor_pixel));
102
102 coverage #(.SQ_SIZE(sq_sz)) coverages (.clk_in(clk_65mhz), .rst_in(rst_in),
103             ↪ .hcount_in(hcount), .vcount_in(vcount), .game_size(game_size),
104             .player_board(player_board),
105             ↪ .opponent_board(opponent_board),
106             ↪ .pixel_out(coverage_pixel),
107             .x_offset(x_offset),
108             ↪ .y_offset(y_offset));
109
109 start_menu menu (.clk_in(clk_65mhz), .rst_in(rst_in), .hcount_in(hcount),
110             ↪ .vcount_in(vcount), .selected(y_in), .pixel_out(menu_pixel));

```

```

96
97     logic mouse_cursor_pixel;
98     assign mouse_cursor_pixel = (hcount > mousex ? hcount - mousex :
99         ↪ hcount) <= 4
100        && (vcount > mousey ? vcount - mousey : mousey -
101            ↪ vcount) <= 4;
102
103    always_ff @(posedge clk_65mhz) begin
104        if (rst_in) begin
105            pixel_out <= 12'hFFF;
106        end else begin
107            pixel_out <=      (mouse_cursor_pixel & mouse_enable) ? 12'h0AO :
108                (menu_pixel_q[0] && state_in == START_MENU) ?
109                    ↪ menu_pixel_q[0] :
110                (cursor_pixel_q[0] && (state_in == SINGLEPLAYER_PLAY_MOVE
111                    || state_in == PLAY_MOVE)) ?
112                        ↪ cursor_pixel_q[0] :
113                            (coverage_pixel_q[0]) ?
114                                ↪ coverage_pixel_q[0] :
115                                    (ship_state >= 4 && b5_pixel) ? b5_pixel :
116                                        (ship_state >= 3 && b4_pixel) ? b4_pixel :
117                                            (ship_state >= 2 && b3b_pixel) ? b3b_pixel
118                                                ↪ :
119                                            (ship_state >= 1 && b3a_pixel) ? b3a_pixel
120                                                ↪ :
121                                            (ship_state >= 0 && b2_pixel) ? b2_pixel :
122                                                (board_pixel_q[0] != CLEAR_PIXEL) ?
123                                                    ↪ board_pixel_q[0] :
124                                                        (state_in == END_GAME_WIN) ? 12'h0FO :
125                                                            (state_in == END_GAME_LOSE) ? 12'hF0O :
126                                                                12'hFFF;
127
128    end
129
130    // the following lines are required for the Nexys4 VGA circuit - do not change
131    assign vga_r = ~blank_q[0] ? pixel_out[11:8] : 0;
132    assign vga_g = ~blank_q[0] ? pixel_out[7:4] : 0;
133    assign vga_b = ~blank_q[0] ? pixel_out[3:0] : 0;
134
135    assign vga_hs = ~hsync_q[0];
136    assign vga_vs = ~vsync_q[0];
137
138 endmodule

```

### 7.1.19 hit\_or\_miss.sv

```
1  `default_nettype none
2
3  module hit_or_miss #(parameter SQ_SIZE = 40)
4      (input wire clk_in, rst_in,
5       input wire [10:0] hcount_in, x_in,
6       input wire [9:0] vcount_in, y_in,
7       input wire [2:0] hit,
8       output logic [11:0] pixel_out);
9
10    logic [11:0] e_pixel;
11    logic [11:0] r_pixel;
12
13    logic [10:0] x_in_q;
14    logic [9:0] y_in_q;
15
16    explosion ex1 (.clk_in(clk_in), .rst_in(rst_in), .hcount_in(hcount_in),
17                  ↪ .vcount_in(vcount_in), .x_in(x_in_q), .y_in(y_in_q),
18                  ↪ .pixel_out(e_pixel));
19    rock rock1 (.clk_in(clk_in), .rst_in(rst_in), .hcount_in(hcount_in),
20                  ↪ .vcount_in(vcount_in), .x_in(x_in_q), .y_in(y_in_q),
21                  ↪ .pixel_out(r_pixel));
22
23  always_ff @(posedge clk_in) begin
24    x_in_q <= x_in;
25    y_in_q <= y_in;
26    if ((hcount_in >= x_in && hcount_in < (x_in+SQ_SIZE)) &&
27        (vcount_in >= y_in && vcount_in < (y_in+SQ_SIZE)))
28      pixel_out <= (hit == 3'b000) ? 12'hccc : //unguessed
29                  (hit == 3'b001) ? e_pixel : //hit
30                  (hit == 3'b010) ? 12'h000 : //uncovered, miss
31                  (hit == 3'b011) ? r_pixel : //rock
32                  (hit == 3'b100) ? (e_pixel >> 1) + (12'h000 >> 1) :
33                  ↪ //sunk
34                  12'h000;
35    else pixel_out <= 0;
36  end
37
38 endmodule
39
40 `default_nettype wire
```

### 7.1.20 hv\_to\_xy.sv

```

1 module hv_to_xy #(parameter SQ_SIZE = 40)
2     (input wire [10:0] hcount_in, start_hcount,
3      input wire [9:0] vcount_in, start_vcount,
4      output logic [3:0] x_out,
5      output logic [3:0] y_out);
6
7 logic [10:0] hcount;
8 logic [9:0] vcount;
9
10 assign hcount = hcount_in >= start_hcount ? hcount_in - start_hcount :
11     ↪ 11'hFFF;
12 assign vcount = vcount_in >= start_vcount ? vcount_in - start_vcount: 10'hFFF;
13
14 always_comb begin
15     if(hcount > SQ_SIZE*0 && hcount <= SQ_SIZE*1) begin
16         x_out = 0;
17     end else if(hcount > SQ_SIZE*1 && hcount <= SQ_SIZE*2) begin
18         x_out = 1;
19     end else if(hcount > SQ_SIZE*2 && hcount <= SQ_SIZE*3) begin
20         x_out = 2;
21     end else if(hcount > SQ_SIZE*3 && hcount <= SQ_SIZE*4) begin
22         x_out = 3;
23     end else if(hcount > SQ_SIZE*4 && hcount <= SQ_SIZE*5) begin
24         x_out = 4;
25     end else if(hcount > SQ_SIZE*5 && hcount <= SQ_SIZE*6) begin
26         x_out = 5;
27     end else if(hcount > SQ_SIZE*6 && hcount <= SQ_SIZE*7) begin
28         x_out = 6;
29     end else if(hcount > SQ_SIZE*7 && hcount <= SQ_SIZE*8) begin
30         x_out = 7;
31     end else if(hcount > SQ_SIZE*8 && hcount <= SQ_SIZE*9) begin
32         x_out = 8;
33     end else if(hcount > SQ_SIZE*9 && hcount <= SQ_SIZE*10) begin
34         x_out = 9;
35     end else if(hcount > SQ_SIZE*10 && hcount <= SQ_SIZE*11) begin
36         x_out = 10;
37     end else if(hcount > SQ_SIZE*11 && hcount <= SQ_SIZE*12) begin
38         x_out = 11;
39     end else x_out = 4'hF;
40
41     if(vcount > SQ_SIZE*0 && vcount <= SQ_SIZE*1) begin
42         y_out = 0;
43     end else if(vcount > SQ_SIZE*1 && vcount <= SQ_SIZE*2) begin
44         y_out = 1;
45     end else if(vcount > SQ_SIZE*2 && vcount <= SQ_SIZE*3) begin
46         y_out = 2;
47     end else if(vcount > SQ_SIZE*3 && vcount <= SQ_SIZE*4) begin
48         y_out = 3;
49     end else if(vcount > SQ_SIZE*4 && vcount <= SQ_SIZE*5) begin
50         y_out = 4;
51     end else if(vcount > SQ_SIZE*5 && vcount <= SQ_SIZE*6) begin
52         y_out = 5;
53     end else if(vcount > SQ_SIZE*6 && vcount <= SQ_SIZE*7) begin
54         y_out = 6;
55     end else if(vcount > SQ_SIZE*7 && vcount <= SQ_SIZE*8) begin
56         y_out = 7;
57     end else if(vcount > SQ_SIZE*8 && vcount <= SQ_SIZE*9) begin
58

```

```
57     y_out = 8;
58 end else if(vcount > SQ_SIZE*9 && vcount <= SQ_SIZE*10) begin
59     y_out = 9;
60 end else if(vcount > SQ_SIZE*10 && vcount <= SQ_SIZE*11) begin
61     y_out = 10;
62 end else if(vcount > SQ_SIZE*11 && vcount <= SQ_SIZE*12) begin
63     y_out = 11;
64 end else y_out = 4'hF;
65 end
66 endmodule
```

### 7.1.21 picture\_blob.sv

```
1  `timescale 1ns / 1ps
2  ///////////////////////////////////////////////////////////////////
3  `default_nettype none
4
5  module picture_blob
6    #(parameter WIDTH = 10,
7      HEIGHT = 10,
8      GRAYSCALE = 1,
9      CLEAR_PIXEL = 12'h000)
10   (input wire pixel_clk_in,
11    input wire [10:0] x_in, hcount_in,
12    input wire [9:0] y_in, vcount_in,
13    input wire [7:0] red_mapped, blue_mapped, green_mapped,
14    input wire flip,
15
16    output logic [$clog2(WIDTH*HEIGHT)-1:0] image_addr,
17    output logic [11:0] pixel_out);
18
19  // calculate rom address and read the location
20  assign image_addr = !flip ? (hcount_in-x_in) + (vcount_in-y_in) * WIDTH :
21                                WIDTH-1-(vcount_in-y_in) + (hcount_in-x_in) *
22                                ↪ WIDTH;
23
24  always_ff @ (posedge pixel_clk_in) begin
25    if (flip) begin
26
27      if ((hcount_in >= x_in && hcount_in < (x_in+HEIGHT)) &&
28          (vcount_in >= y_in && vcount_in < (y_in+WIDTH)))
29        // use MSB 4 bits
30        pixel_out <= GRAYSCALE ? {red_mapped[7:4], red_mapped[7:4],
31                                  ↪ red_mapped[7:4]} :
32                                  {red_mapped[7:4], green_mapped[7:4],
33                                   ↪ blue_mapped[7:4]};
34
35      else pixel_out <= CLEAR_PIXEL;
36
37    end else begin
38
39      if ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
40          (vcount_in >= y_in && vcount_in < (y_in+HEIGHT)))
41        // use MSB 4 bits
42        pixel_out <= GRAYSCALE ? {red_mapped[7:4], red_mapped[7:4],
43                                  ↪ red_mapped[7:4]} :
44                                  {red_mapped[7:4], green_mapped[7:4],
45                                   ↪ blue_mapped[7:4]};
46
47      else pixel_out <= CLEAR_PIXEL;
48
49    end
50  end
51
52  endmodule
53
54  `default_nettype wire
```

### 7.1.22 placeable.sv

```
1  `default_nettype none
2  import fsm_state::*;
3
4  module placeable(input wire[3:0] x, y,
5                      input wire[2:0] ship_num,
6                      input wire[7:0] rocks_in [7:0],
7                      input wire[2:0] num_rocks,
8                      input wire check_one_coord,
9                      input ship_array ships,
10                     output logic placeable);
11
12    logic[3:0] xvar;
13    logic[3:0] yvar;
14    always_comb begin
15        placeable = 1;
16        for(int j = 0; j < ships[ship_num].length; j++) begin
17            xvar = check_one_coord ? x : ships[ship_num].orientation ? x : x + j;
18            yvar = check_one_coord ? y : ships[ship_num].orientation ? y + j : y;
19            for(int i = 0; i < ship_num; i++) begin
20                if(ships[i].orientation == 0) begin // horizontal ship
21                    if(ships[i].x <= xvar && xvar < ships[i].x + ships[i].length
22                        && yvar == ships[i].y)
23                        placeable = 0;
24                end
25                else if(ships[i].y <= yvar && yvar < ships[i].y + ships[i].length
26                        && xvar == ships[i].x)
27                        placeable = 0;
28            end
29        end
30    end
31    end
32  end
33
34 endmodule
35
36 `default_nettype wire
```

### 7.1.23 rng.sv

```
1 //xorshift RNG https://en.wikipedia.org/wiki/Xorshift
2 module rng(input wire rst, clk,
3             input wire[31:0] initial_seed,
4             input wire start,
5             output logic[31:0] number_out);
6   logic generating;
7   always_ff @(posedge clk) begin
8     if(rst) begin
9       number_out <= 0;
10      generating <= 0;
11    end
12    else if (generating) begin
13      number_out <= (number_out ^ (number_out << 13))
14        ^ ((number_out ^ (number_out << 13)) >> 17)
15        ^ (((number_out ^ (number_out << 13)) >> 17) << 5);
16    end
17    else if (start) begin
18      number_out <= initial_seed;
19      generating <= 1;
20    end
21  end
22 endmodule
23
24
25 module bounded_rng #(NUM_NUMBERS = 32) (input wire reset, clk,
26                                         input wire start,
27                                         input wire[31:0] initial_seed,
28                                         input wire[3:0] num_max,
29                                         output logic[3:0] numbers_out [NUM_NUMBERS - 1:0],
30                                         output logic[31:0] unbounded_out); //generates NUM_NUMBERS
31                                         // numbers from 0 to num_max - 1 with MCMC methods;
32
33   logic[31:0] mcmc_output;
34   logic generating;
35   rng rng(.rst(reset), .clk(clk), .initial_seed(initial_seed), .start(start),
36           .number_out(mcmc_output));
37   always_ff @(posedge clk) begin
38     if(reset) begin
39       unbounded_out <= 0;
40       generating <= 0;
41       for(int i = 0; i < NUM_NUMBERS; i++)
42         numbers_out[i] <= 4'b0;
43     end
44     else if (generating) begin
45       for(int i = 0; i < NUM_NUMBERS; i++)
46         if(mcmc_output[i])
47           numbers_out[i] <= numbers_out[i] == 0 ? 0 : numbers_out[i] -
48             1;
49         else
50           numbers_out[i] <= numbers_out[i] >= num_max - 1 ? num_max - 1
51             : numbers_out[i] + 1;
52       unbounded_out <= mcmc_output;
53     end
54     else if (start) begin
55       for(int i = 0; i < NUM_NUMBERS; i++) begin
56         numbers_out[i] <= (num_max >> 1);
57       end
58     end
59   end
60 endmodule
```

```
54         generating <= 1;  
55     end  
56 end  
57 endmodule
```

### 7.1.24 rock.sv

```
1 module rock (input wire clk_in, rst_in,
2                 input wire [10:0] hcount_in, x_in,
3                 input wire [9:0] vcount_in, y_in,
4                 output wire [11:0] pixel_out);
5
6     localparam w = 41;
7     localparam h = 41;
8
9     logic [7:0] image_bits, red_mapped, blue_mapped, green_mapped;
10    logic [$clog2(w*h)-1:0] image_addr;
11
12    picture_blob #(WIDTH(w), HEIGHT(h)) rock (.pixel_clk_in(clk_in),
13                                              .hcount_in(hcount_in), .vcount_in(vcount_in), .x_in(x_in), .y_in(y_in),
14                                              .flip(0), .red_mapped(red_mapped),
15                                              .blue_mapped(blue_mapped),
16                                              .green_mapped(green_mapped),
17                                              .image_addr(image_addr),
18                                              .pixel_out(pixel_out));
19
20    rock_image rock_image (.clka(clk_in), .addra(image_addr), .douta(image_bits));
21    rock_red rock_red (.clka(clk_in), .addra(image_bits), .douta(red_mapped));
22    rock_blue rock_blue (.clka(clk_in), .addra(image_bits), .douta(blue_mapped));
23    rock_green rock_green (.clka(clk_in), .addra(image_bits),
24                           .douta(green_mapped));
25
26 endmodule
```

### 7.1.25 rock\_coords.sv

```
1 `timescale 1ns / 1ps
2 ///////////////////////////////////////////////////////////////////
3 // Company:
4 // Engineer:
5 //
6 // Create Date: 12/06/2021 01:42:40 AM
7 // Design Name:
8 // Module Name: rock_coords
9 // Project Name:
10 // Target Devices:
11 // Tool Versions:
12 // Description:
13 //
14 // Dependencies:
15 //
16 // Revision:
17 // Revision 0.01 - File Created
18 // Additional Comments:
19 //
20 ///////////////////////////////////////////////////////////////////
21
22
23 module rock_coords(input wire[3:0] x, y,
24                     input wire[7:0] rocks_in [7:0],
25                     input wire[2:0] num_rocks,
26                     output logic placeable);
27     always_comb begin
28         placeable = 1;
29         for(int i = 0; i < num_rocks; i++)
30             if(rocks_in[i][3:0] == x && rocks_in[i][7:4] == y)
31                 placeable = 0;
32     end
33 endmodule
```

### 7.1.26 rock\_randomizer.sv

```
1  `default_nettype none
2
3  module rock_randomizer (input wire reset, clk,
4                           input wire [3:0] rng_input [31:0],
5                           input wire start,
6                           output logic done,
7                           output logic[7:0] player_rocks_out[7:0],
8                           output logic[7:0] cpu_rocks_out[7:0]);
9
10 int counter = 0;
11 logic prev_start;
12 logic generating;
13 always_ff @(posedge clk) begin
14     prev_start <= start;
15     if(reset) begin
16         done <= 0;
17         counter <= 0;
18         generating <= 0;
19         for(int i = 0; i < 8; i++) begin
20             player_rocks_out[i] <= 8'b0;
21             cpu_rocks_out[i] <= 8'b0;
22         end
23     end
24     else if((start & ~prev_start)) begin
25         generating <= 1;
26         counter <= 0;
27         done <= 0;
28     end
29     else if(generating) begin
30         counter <= counter + 1;
31         if(counter > 500) begin
32             for(int i = 0; i < 8; i++) begin
33                 player_rocks_out[i] <= {rng_input[i], rng_input[2 * i]};
34                 cpu_rocks_out[i] <= {rng_input[15 + i], rng_input[15 + 2 *
35                                     ↪ i]};
36             end
37             done <= 1;
38             generating <= 0;
39         end
40     end
41 endmodule
42 `default_nettype wire
```

### 7.1.27 rx.sv

```
1  `default_nettype none
2
3  module rx #(parameter BUFFER_SIZE = 16,
4              CLOCK = 65000000,
5              BAUD = 9600)
6      (input wire clk_in, rst_in, rx_in, ready_in,
7       output logic valid_out,
8       output logic [BUFFER_SIZE-1:0] data_out
9      );
10
11  localparam DIVISOR = CLOCK/BAUD;
12  localparam DATA_FRAME = 8;
13
14  typedef enum {IDLE, RX_START, RX_DATA, RX_STOP, END_RX} T_state;
15
16  logic [$clog2(DIVISOR)-1:0] baud_count;
17  logic [$clog2(BUFFER_SIZE):0] data_ptr;
18  logic [$clog2(DATA_FRAME):0] frame_count;
19  logic [DATA_FRAME + 1: 0] frame;
20
21  logic rx_dirty;
22  logic rx_sync;
23  logic old_rx_sync;
24
25  T_state state;
26
27  always_ff @(posedge clk_in) begin
28      old_rx_sync <= rx_sync;
29      rx_sync <= rx_dirty;
30      rx_dirty <= rx_in;
31
32      if(rst_in) begin
33          baud_count <= 0;
34          frame_count <= 0;
35          frame <= 0;
36          data_ptr <= 0;
37          state <= IDLE;
38          data_out <= 0;
39          valid_out <= 1'b0;
40      end else begin
41          case(state)
42              IDLE: begin
43                  valid_out <= 1'b0;
44                  if(!rx_sync && old_rx_sync) begin
45                      state <= RX_START;
46                      baud_count <= 0;
47                  end
48              end
49
50              RX_START: begin
51                  baud_count <= baud_count + 1'b1;
52
53                  if(baud_count == DIVISOR/2) begin
54                      state <= rx_sync ? IDLE : RX_DATA;
55                      baud_count <= 0;
56                  end
57              end
58          end
59      end
60  end
```

```

58
59     RX_DATA: begin
60         if (baud_count == DIVISOR - 1) begin
61             baud_count <= 0;
62             frame[frame_count] <= rx_sync;
63             if(frame_count == DATA_FRAME) begin
64                 data_out[data_ptr +: DATA_FRAME] <=
65                     ↪ frame[DATA_FRAME-1:0];
66                 data_ptr <= data_ptr + 8;
67                 frame_count <= 0;
68                 state <= RX_STOP;
69             end else begin
70                 frame_count <= frame_count + 1'b1;
71             end
72         end else baud_count <= baud_count + 1'b1;
73     end
74
74     RX_STOP: begin
75         if(data_ptr == BUFFER_SIZE) begin
76             state <= ready_in ? IDLE : RX_STOP;
77             valid_out <= 1'b1;
78             data_ptr <= ready_in ? 1'b0 : data_ptr;
79         end else begin
80             state <= IDLE;
81             valid_out <= 1'b0;
82         end
83     end
84     endcase
85   end
86 end
87 endmodule
88
89 `default_nettype wire

```

### 7.1.28 select\_square.sv

```
1  `default_nettype none
2
3  module select_square #(parameter SQ_SIZE = 40)
4      (input wire [10:0] hcount_in, x_in,
5       input wire [9:0] vcount_in, y_in,
6       output logic [11:0] pixel_out);
7
8  always_comb begin
9      if ((hcount_in > x_in + 5 && hcount_in + 5 < (x_in+SQ_SIZE)) &&
10         (vcount_in > y_in + 5 && vcount_in + 5 < (y_in+SQ_SIZE))) begin
11         if((hcount_in == x_in + SQ_SIZE/2) || vcount_in == y_in + SQ_SIZE/2) begin
12             pixel_out = 12'hFOO;
13         end else pixel_out = 12'h000;
14         end else pixel_out = 0;
15     end
16
17 endmodule
18
19 `default_nettype wire
```

### 7.1.29 ship\_coords.sv

```
1  `default_nettype none
2  import fsm_state::*;
3
4  module ship_coords(input wire[3:0] x, y,
5                      input ship_array ships,
6                      input wire[3:0] ship_num,
7                      output logic ship_out);
8
9    always_comb begin
10      ship_out = 0;
11      if(ships[ship_num].orientation == 0) begin // horizontal ship
12        if(ships[ship_num].x <= x && x < ships[ship_num].x +
13          ↪ ships[ship_num].length && y == ships[ship_num].y)
14          ship_out = 1;
15      end
16      else if(ships[ship_num].y <= y && y < ships[ship_num].y +
17          ↪ ships[ship_num].length && x == ships[ship_num].x)
18          ship_out = 1;
19    end
20  endmodule
21
22 `default_nettype wire
```

### 7.1.30 start\_menu.sv

```

1  `default_nettype none
2
3  module start_menu (input wire clk_in, rst_in,
4                      input wire [3:0] selected,
5                      input wire [10:0] hcount_in,
6                      input wire [9:0] vcount_in,
7                      output logic [11:0] pixel_out);
8
9      localparam HCOUNT_MAX = 1023;
10     localparam VCOUNT_MAX = 767;
11
12    logic [2:0] char_pixel;
13    logic [63:0] cstring;
14    logic [10:0] text_x;
15    logic [9:0] text_y;
16
17    logic [10:0] select_x;
18    logic [9:0] select_y;
19
20    logic [11:0] box1_pixel;
21    logic [11:0] box2_pixel;
22    logic [11:0] box3_pixel;
23    logic [11:0] select_pixel;
24
25    assign select_x = HCOUNT_MAX/4;
26    assign select_y = (1'b1+selected) * VCOUNT_MAX/4 - 25;
27
28    char_string_display text(.vclock(clk_in), .hcount(hcount_in),
29                           ↪ .vcount(vcount_in),
30                           .cstring(cstring), .cx(text_x), .cy(text_y),
31                           ↪ .pixel(char_pixel));
32
33    blob #(.WIDTH(HCOUNT_MAX/2), .HEIGHT(50), .COLOR(12'h222)) box1
34                           ↪ (.x_in(HCOUNT_MAX/4), .y_in(VCOUNT_MAX/4-25),
35                                         .vcount_in(vcount_in),
36                                         ↪ .hcount_in(hcount_in),
37                                         .pixel_out(box1_pixel));
38
39    blob #(.WIDTH(HCOUNT_MAX/2), .HEIGHT(50), .COLOR(12'h222)) box2
40                           ↪ (.x_in(HCOUNT_MAX/4), .y_in(VCOUNT_MAX/2-25),
41                                         .vcount_in(vcount_in),
42                                         ↪ .hcount_in(hcount_in),
43                                         .pixel_out(box2_pixel));
44
45    blob #(.WIDTH(HCOUNT_MAX/2), .HEIGHT(50), .COLOR(12'h222)) box3
46                           ↪ (.x_in(HCOUNT_MAX/4), .y_in(3*VCOUNT_MAX/4-25),
47                                         .vcount_in(vcount_in),
48                                         ↪ .hcount_in(hcount_in),
49                                         .pixel_out(box3_pixel));
50
51    blob #(.WIDTH(HCOUNT_MAX/2), .HEIGHT(50), .COLOR(12'hF00)) select_box
52                           ↪ (.x_in(select_x), .y_in(select_y),
53                                         .vcount_in(vcount_in),
54                                         ↪ .hcount_in(hcount_in),
55                                         .pixel_out(select_pixel));
56
57
58    always_ff @(posedge clk_in) begin
59
60        if(hcount_in == HCOUNT_MAX/2 - 32 && vcount_in == VCOUNT_MAX/4 - 16) begin
61            cstring <= "HOST";
62
63
64
65
66
67
68
69
69
70
71
72
73
74
75
76
77
78
79
79
80
81
82
83
84
85
86
87
88
89
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
229
230
231
232
233
234
235
236
237
238
239
239
240
241
242
243
244
245
246
247
248
249
249
250
251
252
253
254
255
256
257
258
259
259
260
261
262
263
264
265
266
267
268
269
269
270
271
272
273
274
275
276
277
278
279
279
280
281
282
283
284
285
286
287
288
289
289
290
291
292
293
294
295
296
297
298
299
299
300
301
302
303
304
305
306
307
308
309
309
310
311
312
313
314
315
316
317
318
319
319
320
321
322
323
324
325
326
327
328
329
329
330
331
332
333
334
335
336
337
338
339
339
340
341
342
343
344
345
346
347
348
349
349
350
351
352
353
354
355
356
357
358
359
359
360
361
362
363
364
365
366
367
368
369
369
370
371
372
373
374
375
376
377
378
379
379
380
381
382
383
384
385
386
387
388
389
389
390
391
392
393
394
395
396
397
398
399
399
400
401
402
403
404
405
406
407
408
409
409
410
411
412
413
414
415
416
417
418
419
419
420
421
422
423
424
425
426
427
428
429
429
430
431
432
433
434
435
436
437
438
439
439
440
441
442
443
444
445
446
447
448
449
449
450
451
452
453
454
455
456
457
458
459
459
460
461
462
463
464
465
466
467
468
469
469
470
471
472
473
474
475
476
477
478
479
479
480
481
482
483
484
485
486
487
488
489
489
490
491
492
493
494
495
496
497
498
499
499
500
501
502
503
504
505
506
507
508
509
509
510
511
512
513
514
515
516
517
518
519
519
520
521
522
523
524
525
526
527
528
529
529
530
531
532
533
534
535
536
537
538
539
539
540
541
542
543
544
545
546
547
548
549
549
550
551
552
553
554
555
556
557
558
559
559
560
561
562
563
564
565
566
567
568
569
569
570
571
572
573
574
575
576
577
578
579
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
628
629
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
688
689
689
690
691
692
693
694
695
696
697
697
698
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
749
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
779
779
780
781
782
783
784
785
786
787
788
789
789
790
791
792
793
794
795
796
797
797
798
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
818
819
819
820
821
822
823
824
825
826
827
828
829
829
830
831
832
833
834
835
836
837
838
839
839
840
841
842
843
844
845
846
847
848
849
849
850
851
852
853
854
855
856
857
858
859
859
860
861
862
863
864
865
866
867
868
869
869
870
871
872
873
874
875
876
877
878
879
879
880
881
882
883
884
885
886
887
887
888
889
889
890
891
892
893
894
895
896
897
897
898
899
899
900
901
902
903
904
905
906
907
908
909
909
910
911
912
913
914
915
916
917
918
919
919
920
921
922
923
924
925
926
927
928
929
929
930
931
932
933
934
935
936
937
938
939
939
940
941
942
943
944
945
946
947
948
949
949
950
951
952
953
954
955
956
957
958
959
959
960
961
962
963
964
965
966
967
968
969
969
970
971
972
973
974
975
976
977
978
979
979
980
981
982
983
984
985
986
987
987
988
989
989
990
991
992
993
994
995
996
997
998
999
999
1000
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1088
1089
1089
1090
1091
1092
1093
1094
1095
1096
1096
1097
1098
1098
1099
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1147
1148
1149
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1168
1169
1170
1171
1172
1173
1174
1175
1176
1176
1177
1178
1178
1179
1179
1180
1181
1182
1183
1184
1185
1186
1186
1187
1188
1188
1189
1189
1190
1191
1192
1193
1194
1195
1195
1196
1197
1197
1198
1198
1199
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1228
1229
1229
1230
1231
1232
1233
1234
1235
1236
1237
1237
1238
1239
1239
1240
1240
1241
1242
1243
1244
1245
1246
1247
1247
1248
1249
1249
1250
1250
1251
1252
1253
1254
1255
1256
1257
1258
1258
1259
1259
1260
1261
1262
1263
1264
1265
1266
1266
1267
1268
1268
1269
1269
1270
1271
1272
1273
1274
1275
1276
1276
1277
1278
1278
1279
1279
1280
1281
1282
1283
1284
1285
1286
1286
1287
1288
1288
1289
1289
1290
1291
1292
1293
1294
1295
1295
1296
1297
1297
1298
1298
1299
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1308
1309
1310
1311
1312
1313
1314
1315
1316
1316
1317
1318
1318
1319
1319
1320
1321
1322
1323
1324
1325
1326
1326
1327
1328
1328
1329
1329
1330
1331
1332
1333
1334
1335
1336
1336
1337
1338
1338
1339
1339
1340
1341
1342
1343
1344
1345
1345
1346
1347
1347
1348
1348
1349
1349
1350
1351
1352
1353
1354
1355
1356
1356
1357
1358
1358
1359
1359
1360
1361
1362
1363
1364
1365
1365
1366
1367
1367
1368
1368
1369
1369
1370
1371
1372
1373
1374
1375
1375
1376
1377
1377
1378
1378
1379
1379
1380
1381
1382
1383
1384
1385
1385
1386
1387
1387
1388
1388
1389
1389
1390
1391
1392
1393
1394
1394
1395
1396
1396
1397
1397
1398
1398
1399
1399
1400
1401
1402
1403
1404
1405
1405
1406
1407
1407
1408
1408
1409
1409
1410
1411
1412
1413
1414
1415
1415
1416
1417
1417
1418
1418
1419
1419
1420
1421
1422
1423
1424
1425
1425
1426
1427
1427
1428
1428
1429
1429
1430
1431
1432
1433
1434
1435
1435
1436
1437
1437
1438
1438
1439
1439
1440
1441
1442
1443
1444
1445
1445
1446
1447
1447
1448
1448
1449
1449
1450
1451
1452
1453
1454
1455
1455
1456
1457
1457
1458
1458
1459
1459
1460
1461
1462
1463
1464
1465
1465
1466
1467
1467
1468
1468
1469
1469
1470
1471
1472
1473
1474
1475
1475
1476
1477
1477
1478
1478
1479
1479
1480
1481
1482
1483
1484
1485
1485
1486
1487
1487
1488
1488
1489
1489
1490
1491
1492
1493
1494
1494
1495
1496
1496
1497
1497
1498
1498
1499
1499
1500
1501
1502
1503
1504
1505
1505
1506
1507
1507
1508
1508
1509
1509
1510
1511
1512
1513
1514
1515
1515
1516
1517
1517
1518
1518
1519
1519
1520
1521
1522
1523
1524
1525
1525
1526
1527
1527
1528
1528
1529
1529
1530
1531
1532
1533
1534
1535
1535
1536
1537
1537
1538
1538
1539
1539
1540
1541
1542
1543
1544
1545
1545
1546
1547
1547
1548
1548
1549
1549
1550
1551
1552
1553
1554
1555
1555
1556
1557
1557
1558
1558
1559
1559
1560
1561
1562
1563
1564
1565
1565
1566
1567
1567
1568
1568
1569
1569
1570
1571
1572
1573
1574
1575
1575
1576
1577
1577
1578
1578
1579
1579
1580
1581
1582
1583
1584
1585
1585
1586
1587
1587
1588
1588
1589
1589
1590
1591
1592
1593
1594
1594
1595
1596
1596
1597
1597
1598
1598
1599
1599
1600
1601
1602
1603
1604
1605
1605
1606
1607
1607
1608
1608
1609
1609
1610
1611
1612
1613
1614
1615
1615
1616
1617
1617
1618
1618
1619
1619
1620
1621
1622
1623
1624
1625
1625
1626
1627
1627
1628
1628
1629
1629
1630
1631
1632
1633
1634
1635
1635
1636
1637
1637
1638
1638
1639
1639
1640
1641
1642
1643
1644
1645
1645
1646
1647
1647
1648
1648
1649
1649
1650
1651
1652
1653
1654
1655
1655
1656
1657
1657
1658
1658
1659
1659
1660
1661
1662
1663
1664
1665
1665
1666
1667
1667
1668
1668
1669
1669
1670
1671
1672
1673
1674
1675
1675
1676
1677
1677
1678
1678
1679
1679
1680
1681
1682
1683
1684
1685
1685
1686
1687
1687
1688
1688
1689
1689
1690
1691
1692
1693
1694
1694
1695
1696
1696
1697
1697
1698
1698
1699
1699
1700
1701
1702
1703
1704
1705
1705
1706
1707
1707
1708
1708
1709
1709
1710
1711
1712
1713
1714
1715
1715
1716
1717
1717
1718
1718
1719
1719
1720
1721
1722
1723
1724
1725
1725
1726
1727
1727
1728
1728
1729
1729
1730
1731
1732
1733
1734
1735
1735
1736
1737
1737
1738
1738
1739
1739
1740
1741
1742
1743
1744
1745
1745
1746
1747
1747
1748
1748
1749
1749
1750
1751
1752
1753
1754
1755
1755
1756
1757
1757
1758
1758
1759
1759
1760
1761
1762
1763
1764
1765
1765
1766
1767
1767
1768
1768
1769
1769
1770
1771
1772
1773
1774
1775
1775
1776
1777
1777
1778
1778
1779
1779
1780
1781
1782
1783
1784
1785
1785
1786
1787
1787
1788
1788
1789
1789
1790
1791
1792
1793
1794
1795
1795
1796
1797
1797
1798
1798
1799
1799
1800
1801
1802
1803
1804
1805
1805
1806
1807
1807
1808
1808
1809
1809
1810
1811
1812
1813
1814
1815
1815
1816
1817
1817
1818
1818
1819
1819
1820
1821
1822
1823
1824
1825
1825
1826
1827
1827
1828
1828
1829
1829
1830
1831
1832
1833
1834
1835
1835
1836
1837
1837
1838
1838
1839
1839
1840
1841
1842
1843
1844
1845
1845
1846
1847
1847
1848
1848
1849
1849
1850
1851
1852
1853
1854
1855
1855
1856
1857
1857
1858
1858
1859
1859
1860
1861
1862
1863
1864
1865
1865
1866
1867
1867
1868
1868
1869
1869
1870
1871
1872
1873
1874
1875
1875
1876
1877
1877
1878
1878
1879
1879
1880
1881
1882
1883
1884
1885
1885
1886
1887
1887
1888
1888
1889
1889
1890
1891
1892
1893
1894
1895
1895
1896
1897
1897
1898
1898
1899
1899
1900
1901
1902
1903
1904
1905
1905
1906
1907
1907
1908
1908
1909
1909
1910
1911
1912
1913
1914
1915
1915
1916
1917
1917
1918
1918
1919
1919
1920
1921
1922
1923
1924
1925
1925
1926
1927
1927
1928
1928
1929
1929
1930
1931
1932
1933
1934
1935
1935
1936
1937
1937
1938
1938
1939
1939
1940
1941
1942
1943
1944
1945
1945
1946
1947
1947
1948
1948
1949
1949
1950
1951
1952
1953
1954
1955
1955
1956
1957
1957
1958
1958
1959
1959
1960
1961
1962
1963
1964
1965
1965
1966
1967
1967
1968
1968
1969
1969
1970
1971
1972
1973
1974
1975
1975
1976
1977
1977
1978
1978
1979
1979
1980
1981
1982
1983
1984
1985
1985
1986
1987
1987
1988
1988
1989
1989
1990
1991
1992
1993
1994
1995
1995
1996
1997
1997
1998
1998
1999
1999
2000
2001
2002
2003
2004
2005
2005
2006
2007
2007
2008
2008
2009
2009
2010
2011
2012
2013
2014
2015
2015
2016
2017
2017
2018
2018
2019
2019
2020
2021
2022
2023
2024
2025
2025
2026
2027
2027
2028
2028
2029
2029
2030
2031
2032
2033
2034
2035
2035
2036
2037
2037
2038
2038
2039
2039
2040
2041
2042
2043
2044
2045
2045
2046
2047
2047
2048
2048
2049
2049
2050
2051
2052
2053
2054
2055
2055
2056
2057
2057
2058
2058
2059
2059
2060
2061
2062
2063
2064
2065
2065
2066
2067
2067
2068
2068
2069
2069
2070
2071
2072
2073
2074
2075
2075
2076
2077
2077
2078
2078
2079
2079
2080
2081
2082
2083
2084
2085
2085
2086
2087
2087
2088
2088
2089
2089
2090
2091
2092
2093
2094
2095
2095
2096
2097
2097
2098
2098
2099
2099
2100
2101
2102
2103
2104
2105
2105
2106
2107
2107
2108
2108
2109
2109
2110
2111
2112
2113
2114
2115
2115
2116
2117
2117
2118
2118
2119
2119
```

```

48      text_x <= hcount_in;
49      text_y <= vcount_in;
50  end else if (hcount_in == HCOUNT_MAX/2 - 32 && vcount_in == VCOUNT_MAX/2
51    ↵ - 16) begin
52      cstring <= "JOIN";
53      text_x <= hcount_in;
54      text_y <= vcount_in;
55  end else if (hcount_in == HCOUNT_MAX/2 - 32 && vcount_in ==
56    ↵ 3*VCOUNT_MAX/4 - 16) begin
57      cstring <= "SOLO";
58      text_x <= hcount_in;
59      text_y <= vcount_in;
60  end
61
62      pixel_out <= char_pixel[2] ? {12{char_pixel[2]}} :
63          select_pixel ? select_pixel :
64          box1_pixel ? box1_pixel :
65          box2_pixel ? box2_pixel :
66          box3_pixel ? box3_pixel :
67          12'h777;
68
69  endmodule
70
71 `default_nettype wire

```

### 7.1.31 top\_level.sv

```
1  `default_nettype none
2  import fsm_state::*;
3  //https://stackoverflow.com/questions/59041579/can-enum-be-made-an-output-in-systemverilog
4
5  module top_level (input wire clk_100mhz ,
6                      input wire[15:0] sw,
7                      input wire btnc, btnu, btnd, btnr, btnl,
8                      input wire [7:0] ja,
9                      inout wire ps2_clk, ps2_data, // data to/from PS/2 mouse
10                     output logic [7:0] jb,
11                     output logic ca,cb,cc,cd,ce,cf,cg,
12                     output logic [3:0] vga_r, vga_g, vga_b,
13                     output logic vga_hs, vga_vs,
14                     output logic [7:0] an,
15                     output logic aud_pwm, aud_sd);
16
17     //button/switch inputs:
18     //sw[0] = rotate ship
19     //sw[2:1] + 9 = game size
20     //sw[3] = singleplayer hard mode enable
21     //sw[6:4] = number of rocks
22     //sw[9] = mouse enable/disable
23     //sw[10] = sounds on/off
24     //sw[11] = bgm on/off
25     //sw[14:12] = volume control
26     //sw[15] = reset
27     //btnc = main button
28     //btnd, bbtnl, btnr, btnu for movement
29
30     logic clk_65mhz;
31     clk_wiz_0 clk65 (.clk_in1(clk_100mhz), .clk_out1(clk_65mhz));
32
33     logic mouse_click;
34     logic mouse_rightclick;
35     logic mouse_rightclick_clean;
36     logic btnc_clean, btnd_clean, btnu_clean, btnr_clean, bbtnl_clean, mouse_clean;
37     debouncer mouse_debouncer (.reset(sw[15]), .clk_in(clk_65mhz),
38         ↳ .noisy_in(mouse_click), .clean_out(mouse_clean));
39     debouncer mouse_rightclick_debouncer (.reset(sw[15]), .clk_in(clk_65mhz),
40         ↳ .noisy_in(mouse_rightclick), .clean_out(mouse_rightclick_clean));
41
42     logic [11:0] mx, my; // current mouse position, 12 bits
43     MouseCtl mymouse(
44         .clk (clk_65mhz) , // in std_logic;
45         .rst (sw[15]) , // in std_logic;
46         .xpos (mx) , // out std_logic_vector(11 downto 0);
47         .ypos (my) , // out std_logic_vector(11 downto 0);
48         .zpos () , // out std_logic_vector(3 downto 0);
49         .left (mouse_click) , // out std_logic;
50         .middle () , // out std_logic;
51         .right (mouse_rightclick) , // out std_logic;
52         .new_event () , // out std_logic;
53         .value (12'b0) , // in std_logic_vector(11 downto 0);
54         .setx (1'b0) , // in std_logic;
55         .sety (1'b0) , // in std_logic;
56         .setmax_x (1'b0) , // in std_logic;
57         .setmax_y (1'b0) , // in std_logic;
```

```

56     .ps2_clk      (ps2_clk)          , // inout std_logic;
57     .ps2_data     (ps2_data)         // inout std_logic
58 );
59
60 debouncer btnc_debouncer (.reset(sw[15]), .clk_in(clk_65mhz),
61   ↪ .noisy_in(btnc), .clean_out(btnc_clean));
62 debouncer btnd_debouncer (.reset(sw[15]), .clk_in(clk_65mhz),
63   ↪ .noisy_in(btnd), .clean_out(btnd_clean));
64 debouncer btnu_debouncer (.reset(sw[15]), .clk_in(clk_65mhz),
65   ↪ .noisy_in(btnu), .clean_out(btnu_clean));
66 debouncer btnr_debouncer (.reset(sw[15]), .clk_in(clk_65mhz),
67   ↪ .noisy_in(btnr), .clean_out(btnr_clean));
68 debouncer btnl_debouncer (.reset(sw[15]), .clk_in(clk_65mhz),
69   ↪ .noisy_in(btnl), .clean_out(btnl_clean));
70
71 logic [31:0] debug;
72 game_state_controller gsc (.reset(sw[15]), .clk(clk_65mhz),
73   .btnc(btnc_clean | mouse_clean), .btnd(btnd_clean),
74     ↪ .btnu(btnu_clean), .btnl(btnl_clean),
75     ↪ .btnr(btnr_clean), //button inputs
76   .rotate_ship(sw[0] ^ (mouse_rightclick_clean & sw[9])), //ship
77     ↪ .game_size_switch({2'b00, sw[2:1]}), //size options
78   .vga_r(vga_r), .vga_g(vga_g), .vga_b(vga_b),
79     ↪ .vga_hs(vga_hs), .vga_vs(vga_vs),
80   .num_rocks(sw[6:4]),
81   .rx_in(ja[0]), .tx_out(jb[0]), .debug(debug),
82     ↪ .mousex(mx), .mousey(my),
83   .volume_control(sw[14:12]), .bgm_on(sw[11]),
84     ↪ .sounds_on(sw[10]),
85   .aud_pwm(aud_pwm), .aud_sd(aud_sd), .bgm_test(sw[5]),
86   .mouse_enable(sw[9]), .hard_mode(sw[3])); //board state
87
88 logic [6:0] segments;
89 assign {cg, cf, ce, cd, cc, cb, ca} = segments[6:0];
90 display_8hex display(.clk_in(clk_65mhz), .data_in(debug), .seg_out(segments),
91   ↪ .strobe_out(an)); //for debugging
92
93 endmodule
94
95 module display_8hex(
96   input wire clk_in,           // system clock
97   input wire [31:0] data_in,    // 8 hex numbers, msb first
98   output logic [6:0] seg_out,   // seven segment display output
99   output logic [7:0] strobe_out // digit strobe
100 );
101
102 localparam bits = 13;
103
104 logic [bits:0] counter = 0; // clear on power up
105
106 logic [6:0] segments[15:0]; // 16 7 bit memorys
107 assign segments[0] = 7'b100_0000; // inverted logic
108 assign segments[1] = 7'b111_1001; // gfedcba
109 assign segments[2] = 7'b010_0100;
110 assign segments[3] = 7'b011_0000;
111 assign segments[4] = 7'b001_1001;
112 assign segments[5] = 7'b001_0010;
113 assign segments[6] = 7'b000_0010;

```

```

102     assign segments[7] = 7'b111_1000;
103     assign segments[8] = 7'b000_0000;
104     assign segments[9] = 7'b001_1000;
105     assign segments[10] = 7'b000_1000;
106     assign segments[11] = 7'b000_0011;
107     assign segments[12] = 7'b010_0111;
108     assign segments[13] = 7'b010_0001;
109     assign segments[14] = 7'b000_0110;
110     assign segments[15] = 7'b000_1110;
111
112     always_ff @(posedge clk_in) begin
113         // Here I am using a counter and select 3 bits which provides
114         // a reasonable refresh rate starting the left most digit
115         // and moving left.
116         counter <= counter + 1;
117         case (counter[bits:bits-2])
118             3'b000: begin // use the MSB 4 bits
119                 seg_out <= segments[data_in[31:28]];
120                 strobe_out <= 8'b0111_1111 ;
121             end
122
123             3'b001: begin
124                 seg_out <= segments[data_in[27:24]];
125                 strobe_out <= 8'b1011_1111 ;
126             end
127
128             3'b010: begin
129                 seg_out <= segments[data_in[23:20]];
130                 strobe_out <= 8'b1101_1111 ;
131             end
132             3'b011: begin
133                 seg_out <= segments[data_in[19:16]];
134                 strobe_out <= 8'b1110_1111;
135             end
136             3'b100: begin
137                 seg_out <= segments[data_in[15:12]];
138                 strobe_out <= 8'b1111_0111;
139             end
140
141             3'b101: begin
142                 seg_out <= segments[data_in[11:8]];
143                 strobe_out <= 8'b1111_1011;
144             end
145
146             3'b110: begin
147                 seg_out <= segments[data_in[7:4]];
148                 strobe_out <= 8'b1111_1101;
149             end
150             3'b111: begin
151                 seg_out <= segments[data_in[3:0]];
152                 strobe_out <= 8'b1111_1110;
153             end
154
155         endcase
156     end
157 endmodule
158
159 'default_nettype wire

```

### 7.1.32 tx.sv

```

1  `default_nettype none
2
3 module tx
4   #(parameter BUFFER_SIZE = 16,
5    CLOCK = 65000000,
6    BAUD = 9600)
7   (input wire clk_in, rst_in, valid_in,
8    input wire [BUFFER_SIZE-1:0] data_in,
9    output logic tx_out, ready_out
10 );
11
12 localparam DIVISOR = CLOCK/BAUD;
13 localparam DATA_FRAME = 8;
14
15 typedef enum {IDLE, SEND_DATA, END_SEND} T_state;
16
17 logic [BUFFER_SIZE-1:0] data;
18 logic [$clog2(BUFFER_SIZE):0] data_ptr;
19 logic [$clog2(DIVISOR)-1:0] baud_count;
20 logic [9:0] frame;
21 logic [3:0] frame_count;
22
23 T_state state;
24
25 always_ff @(posedge clk_in) begin
26   if (rst_in) begin
27     state <= IDLE;
28     baud_count <= 0;
29     frame <= 10'b11_1111_1111;
30     frame_count <= 0;
31     data_ptr <= 0;
32     ready_out <= 1'b0;
33     tx_out <= 1'b1;
34   end else begin
35     case (state)
36       IDLE:
37         begin
38           if(valid_in) begin
39             state <= SEND_DATA;
40             frame_count <= 0;
41             frame <= {1'b1, data_in[data_ptr +: DATA_FRAME],
42                         ↪ 1'b0};
43             data <= data_in;
44             ready_out <= 1'b0;
45             baud_count <= 0;
46           end
47         end
48       SEND_DATA: begin
49         if (baud_count == DIVISOR - 1) begin
50           baud_count <= 0;
51           tx_out <= frame[0];
52           if (frame_count == DATA_FRAME + 1) begin
53             data_ptr <= data_ptr + DATA_FRAME;
54             if (data_ptr == BUFFER_SIZE - DATA_FRAME) begin
55               state <= END_SEND;
56             end else begin

```

```

57          frame <= {1'b1, data[data_ptr + DATA_FRAME +:
58                          ↪ DATA_FRAME], 1'b0};
59          frame_count <= 0;
60      end
61  end else begin
62      frame_count <= frame_count + 1'b1;
63      frame <= frame >> 1;
64  end
65  else baud_count <= baud_count + 1'b1;
66 end
67
68 END_SEND: begin
69     state <= IDLE;
70     ready_out <= 1'b1;
71     tx_out <= 1'b1;
72     data_ptr <= 0;
73     frame_count <= 0;
74     baud_count <= 0;
75 end
76 endcase
77 end
78 end
79
80 endmodule
81
82 'default_nettype wire

```

### 7.1.33 user\_input.sv

```
1  `default_nettype none
2  import fsm_state::*;
3
4  module user_input(input wire reset, btnd, btnu, btnr, btnl, clk,
5                      input game_state state,
6                      input wire[3:0] size,
7                      input wire[2:0] ship_num,
8                      input wire rotate,
9                      input wire mouse_enable,
10                     input wire[11:0] mousex, mousey,
11                     input ship_array ships,
12                     output logic [3:0] x, y);
13
14    localparam xsize = 1024;
15    localparam ysize = 768;
16    localparam spacing = 62;
17    localparam [5:0] sq_sz = 40;
18    logic old_btnd, old_btnu, old_btnr, old_btnl;
19    logic [10:0] x_offset;
20    logic [9:0] y_offset;
21
22    assign x_offset = (size == 4'd9) ? 10'd123 :
23                  (size == 4'd10) ? 10'd082 :
24                  (size == 4'd11) ? 10'd041 :
25                  (size == 4'd12) ? 10'd000 :
26                  10'b0;
27
28    assign y_offset = (size == 4'd12) ? 9'd137 :
29                  (size == 4'd11) ? 9'd158 :
30                  (size == 4'd10) ? 9'd179 :
31                  (size == 4'd9) ? 9'd199 :
32                  10'b0;
33
34    logic [11:0] px, py, ox, oy;
35    hv_to_xy #(.SQ_SIZE(sq_sz)) player (.hcount_in(mousex),
36                                         ↳ .start_hcount(x_offset), .vcount_in(mousey), .start_vcount(y_offset),
37                                         ↳ .x_out(px), .y_out(py));
38    hv_to_xy #(.SQ_SIZE(sq_sz)) opponent (.hcount_in(mousex),
39                                         ↳ .start_hcount(512+spacing/2), .vcount_in(mousey),
40                                         ↳ .start_vcount(y_offset), .x_out(ox), .y_out(oy));
41
42    always_ff @(posedge clk) begin
43      if(reset) begin
44        x <= 0;
45        y <= 0;
46        old_btnd <= 1;
47        old_btnu <= 1;
48        old_btnr <= 1;
49        old_btnl <= 1;
50      end
51      else begin
52        old_btnd <= btnd;
53        old_btnu <= btnu;
54        old_btnr <= btnr;
55        old_btnl <= btnl;
56
57        if(state == START_MENU) begin
58          if(btnd)
59            state = MENU;
60          else if(btnu)
61            state = GAME;
62          else if(btnr)
63            state = PAUSE;
64          else if(btnl)
65            state = EXIT;
66        end
67      end
68    end
69  endmodule
```

```

54 //only vertical menu scroll
55 //here is menu with 3 options (x,y) = (0, 0) (0, 1) (0, 2)
56 if(~old_btnd & btnd) y <= y == 2 ? y : y + 1;
57 else if(~old_btnd & btnd) y <= y == 0 ? y : y - 1;
58 else if(mouse_enable) begin
59     if(xsize/4 <= mousex && mousex <= xsize * 3/4) begin
60         if (mousey >= ysize/4 - 25 && mousey <= ysize/4 + 25) y
61             ↪ <= 0;
62         else if (mousey >= ysize/2 - 25 && mousey <= ysize/2 +
63             ↪ 25) y <= 1;
64         else if (mousey >= 3*ysize/4 - 25 && mousey <= 3*ysize/4
65             ↪ + 25) y <= 2;
66     end
67 end
68 end
69 else if(state == CONNECTION_WAITING || state == CONNECTION_SENDING
70     || state == SETUP_WAITING || state == WAIT_MOVE_WAITING
71     || state == SINGLEPLAYER_PLACE_CPU_SHIPS
72     || state == SINGLEPLAYER_PLAY_MOVE_STALL) begin
73     //reset all x, y pos
74     x <= 0;
75     y <= 0;
76 end
77 else if(state == SETUP_BOARD || state == SINGLEPLAYER_PLACE_SHIPS)
78     ↪ begin
79         // (x, y) -> {0, 1, ..., size - 1}^2
80         // if rotate = 0, then the ship is horizontal. y should have no
81             ↪ additional bounds, while x must be reduced by the ship
82             ↪ length
83         // similarly, if rotate = 1, then the ship is vertical, and x
84             ↪ should have the additional bound
85         if(~old_btnd & btnd) y <= y == size - 1 - (rotate ?
86             ↪ ships[ship_num].length - 1: 0) ? y : y + 1;
87         else if(~old_btnd & btnd) y <= y == 0 ? y : y - 1;
88         else if(~old_btnd & btnd) x <= x == size - 1 - (rotate ? 0 :
89             ↪ ships[ship_num].length - 1) ? x : x + 1;
90         else if(~old_btnd & btnd) x <= x == 0 ? x : x - 1;
91         else if(mouse_enable && (px != 4'hF) && (py != 4'hF)) begin
92             x <= px > size - 1 - (rotate ? 0 : ships[ship_num].length -
93                 ↪ 1) ? size - 1 - (rotate ? 0 : ships[ship_num].length -
94                 ↪ 1) : px;
95             y <= py > size - 1 - (rotate ? ships[ship_num].length - 1 :
96                 ↪ 0) ? size - 1 - (rotate ? ships[ship_num].length - 1 :
97                 ↪ 0) : py;
98         end
99         else begin
100             //if (x, y) coordinates out of bounds, then move to closest
101                 ↪ valid boundary
102             x <= x > size - 1 - (rotate ? 0 : ships[ship_num].length - 1)
103                 ↪ ? size - 1 - (rotate ? 0 : ships[ship_num].length - 1) :
104                 ↪ x;
105             y <= y > size - 1 - (rotate ? ships[ship_num].length - 1 : 0)
106                 ↪ ? size - 1 - (rotate ? ships[ship_num].length - 1 : 0) :
107                 ↪ y;
108         end
109     end
110 else if(state == PLAY_MOVE || state == SINGLEPLAYER_PLAY_MOVE) begin
111     // (x, y) -> {0, 1, ..., size - 1}^2

```

```

95      // no further restrictions on shape due to ship size
96      if(~old_btnd & btnd) y <= y == size - 1 ? y : y + 1;
97      else if(~old_btnu & btnu) y <= y == 0 ? y : y - 1;
98      else if(~old_btnr & btnr) x <= x == size - 1 ? x : x + 1;
99      else if(~old_btnl & btnl) x <= x == 0 ? x : x - 1;
100     else if(mouse_enable && (ox != 4'hF) && (oy != 4'hF) ) begin
101         x <= ox > size - 1 ? size - 1 : ox;
102         y <= oy > size - 1 ? size - 1 : oy;
103     end
104   end
105
106   //for further functionality if needed
107 end
108
109
110 endmodule
111 'default_nettype wire

```

### 7.1.34 water.sv

```
1 //////////////////////////////////////////////////////////////////
2 //
3 //////////////////////////////////////////////////////////////////
4 `default_nettype none
5
6 module water(    input wire clk_in, rst_in,
7                  input wire [10:0] hcount_in,
8                  input wire [9:0] vcount_in,
9                  output logic [11:0] pixel_out
10 );
11
12     logic [9:0] sr;
13
14     always_ff @(posedge clk_in) begin
15         if (rst_in) begin
16             sr <= 10'b000_0000_0000;
17             pixel_out <= 12'hFFF;
18         end else begin
19             if ((hcount_in == 0 || hcount_in == 50 || hcount_in == 100 ||
20                  ↪ hcount_in == 150 || hcount_in == 200 ||
21                  ↪ hcount_in == 250 || hcount_in == 350 || hcount_in == 400 ||
22                  ↪ hcount_in == 450 || hcount_in == 500 ||
23                  ↪ hcount_in == 550 || hcount_in == 600 || hcount_in == 650 ||
24                  ↪ hcount_in == 700 || hcount_in == 750 ||
25                  ↪ hcount_in == 800 || hcount_in == 850 || hcount_in == 900 ||
26                  ↪ hcount_in == 950 || hcount_in == 1000) && (
27                     vcount_in == 0 || vcount_in == 50 || vcount_in == 100 ||
28                     ↪ vcount_in == 150 || vcount_in == 200 ||
29                     vcount_in == 250 || vcount_in == 300 || vcount_in == 350 ||
30                     ↪ vcount_in == 400 || vcount_in == 450 ||
31                     vcount_in == 500))
32             begin
33                 sr <= vcount_in;
34                 pixel_out <= (sr > 768) ? 12'hFFF : 12'h07F;
35             end else begin
36                 sr <= {sr[8:0], ~(sr[9] ^ sr[6])};
37                 pixel_out <= (sr > 768) ? 12'hFFF : 12'h07F;
38             end
39         end
40     end
41
42 endmodule
43
44 `default_nettype wire
```

### 7.1.35 xvga.sv

```

1  `timescale 1ns / 1ps
2  ///////////////////////////////////////////////////////////////////
3  // Update: 8/8/2019 GH
4  // Create Date: 10/02/2015 02:05:19 AM
5  // Module Name: xvga
6  //
7  // xvga: Generate VGA display signals (1024 x 768 @ 60Hz)
8  //
9  //          ----- HORIZONTAL -----      ----- VERTICAL -----
10 //          Active   Video    FP   Sync   BP   Active   Video    FP   Sync   BP
11 //       Freq
12 //   640x480, 60Hz   25.175   640    16    96    48   480    11    2    31
13 //   800x600, 60Hz   40.000   800    40   128    88   600     1    4    23
14 //   1024x768, 60Hz  65.000  1024    24   136   160   768     3    6    29
15 //   1280x1024, 60Hz 108.00  1280    48   112   248   768     1    3    38
16 //   1280x720p 60Hz  75.25  1280    72    80   216   720     3    5    30
17 //   1920x1080 60Hz 148.5   1920    88    44   148  1080     4    5    36
18 //
19 // change the clock frequency, front porches, sync's, and back porches to create
20 // other screen resolutions
21 ///////////////////////////////////////////////////////////////////
22 `default_nettype none
23
24 module xvga(input wire vclock_in,
25               output logic [10:0] hcount_out,      // pixel number on current line
26               output logic [9:0] vcount_out,       // line number
27               output logic vsync_out, hsync_out,
28               output logic blank_out);
29
30 parameter DISPLAY_WIDTH = 1024;           // display width
31 parameter DISPLAY_HEIGHT = 768;          // number of lines
32
33 parameter H_FP = 24;                     // horizontal front porch
34 parameter H_SYNC_PULSE = 136;           // horizontal sync
35 parameter H_BP = 160;                    // horizontal back porch
36
37 parameter V_FP = 3;                     // vertical front porch
38 parameter V_SYNC_PULSE = 6;             // vertical sync
39 parameter V_BP = 29;                    // vertical back porch
40
41 // horizontal: 1344 pixels total
42 // display 1024 pixels per line
43 logic hblank, vblank;
44 logic hsyncon, hsyncoff, hreset, hblankon;
45 assign hblankon = (hcount_out == (DISPLAY_WIDTH - 1));
46 assign hsyncon = (hcount_out == (DISPLAY_WIDTH + H_FP - 1)); //1047
47 assign hsyncoff = (hcount_out == (DISPLAY_WIDTH + H_FP + H_SYNC_PULSE - 1));
48   ↪ // 1183
49 assign hreset = (hcount_out == (DISPLAY_WIDTH + H_FP + H_SYNC_PULSE + H_BP -
50   ↪ 1)); //1343
51
52 // vertical: 806 lines total
53 // display 768 lines
54 logic vsyncon, vsyncoff, vreset, vblankon;
55 assign vblankon = hreset & (vcount_out == (DISPLAY_HEIGHT - 1)); // 767
56 assign vsyncon = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP - 1)); // 771
57 assign vsyncoff = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP +

```

```

56      ↪ V_SYNC_PULSE - 1)); // 777
57 assign vreset = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP + V_SYNC_PULSE
58      ↪ + V_BP - 1)); // 805
59
60 // sync and blanking
61 logic next_hblank,next_vblank;
62 assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
63 assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
64 always_ff @(posedge vclock_in) begin
65     hcount_out <= hreset ? 0 : hcount_out + 1;
66     hblank <= next_hblank;
67     hsync_out <= hsyncon ? 0 : hsyncoff ? 1 : hsync_out; // active low
68
69     vcount_out <= hreset ? (vreset ? 0 : vcount_out + 1) : vcount_out;
70     vblank <= next_vblank;
71     vsync_out <= vsyncon ? 0 : vsyncoff ? 1 : vsync_out; // active low
72
73     blank_out <= next_vblank | (next_hblank & ~hreset);
74 end
75 endmodule
76
77 `default_nettype wire

```

## 7.2 Python Scripts

### 7.2.1 Audio Generation Script

```
1 from lib6003.fft import fft,ifft
2 from lib6003.audio import wav_read,wav_write
3 from math import pi
4 import matplotlib.pyplot as plt
5 import numpy as np
6
7
8 def filter(wav, fs, interval):
9     a, b = interval[0], interval[1]
10    band_filter = list(map(lambda x: 1 if 2*pi/fs*a <= min(abs(x-pi), abs(x+pi))
11                           ↪ <= 2*pi/fs*b else 0, np.linspace(-pi, pi, len(wav) + 1)))
12    output = ifft([i*j for i, j in zip(fft(wav), band_filter)])
13    print(output[:20]) #verify that there are no imaginary parts
14    #take real part of output only
15    output = np.array(list(map(lambda x:x.real, output)))
16    #normalize to maximize volume
17    output = output / np.max(abs(output))
18    return output
19
20 nr = 6000 #desired sampling frequency
21
22 filepath = 'FILEPATH'
23 wav,fs = wav_read(filepath)
24 output = filter(wav, fs, (0, nr / 2))[:fs//nr] #filter out all frequencies above
25   ↪ Nyquist frequency
26 #print(output[:20])
27 f = lambda x: round(x.real*2**7) / 2**7 #to change to 8-bit file
28 no = list(map(f, output))
29 print(no[:20])
30 wav_write(no, nr, 'out.wav')
31 with open('out.txt', 'w') as f:
32     f.write("""memory_initialization_radix=2;\nmemory_initialization_vector=\n""")
33     for i, l in enumerate(no):
34         val = bin(round(l * 2 ** 7 + 2**7))[2:] # offset binary
35         f.write(val.zfill(8) + ',\n')
36
37 print(len(no))
```

### 7.2.2 Random Walk Verification

```
1 import numpy as np
2
3 def generate_walk_matrix(n):
4     M = [1/2 if (i == 0 and j == 0)
5          or (i == n-1 and j == n-1)
6          or abs(i - j) == 1
7          else 0 for i in range(n) for j in range(n)]
8     return np.array(M).reshape(n, n)
9
10 game_sizes = [9, 10, 11, 12]
11 for n in game_sizes:
12     print(np.linalg.matrix_power(generate_walk_matrix(n), 500))
```