6.111 Final Report: Battleship

Kade Bose and Edward Jin

{kadebose, ehjin}@mit.edu

Contents

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.

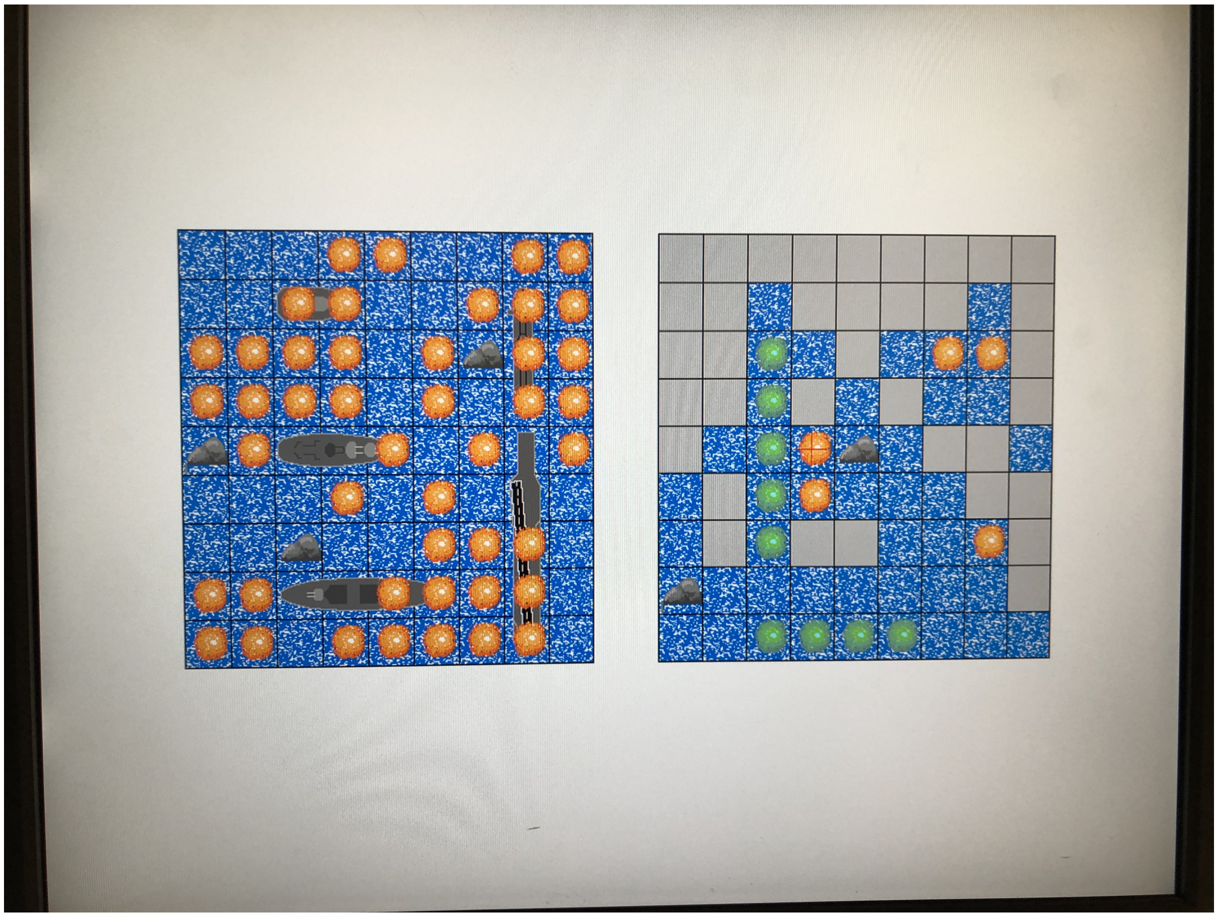


Figure 1: A game of battleship being played over two FPGAs

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.

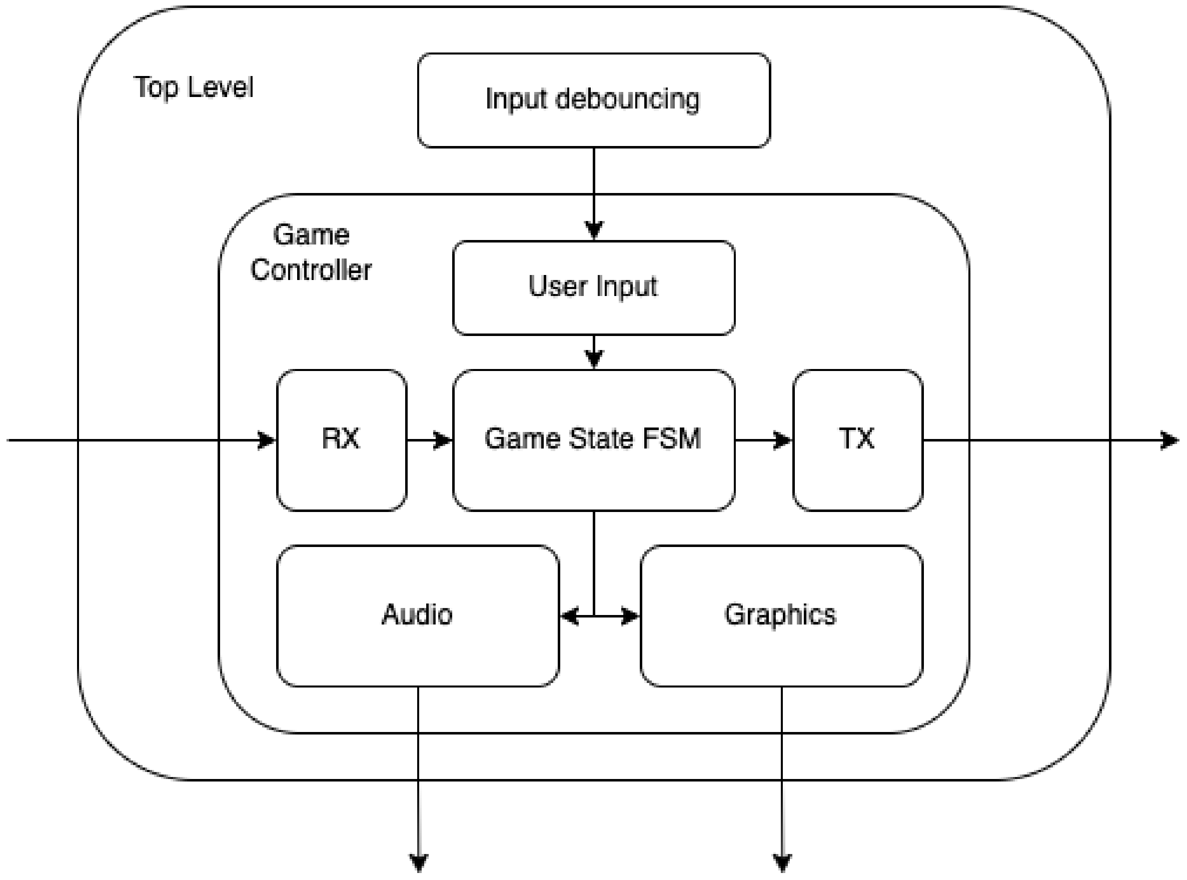


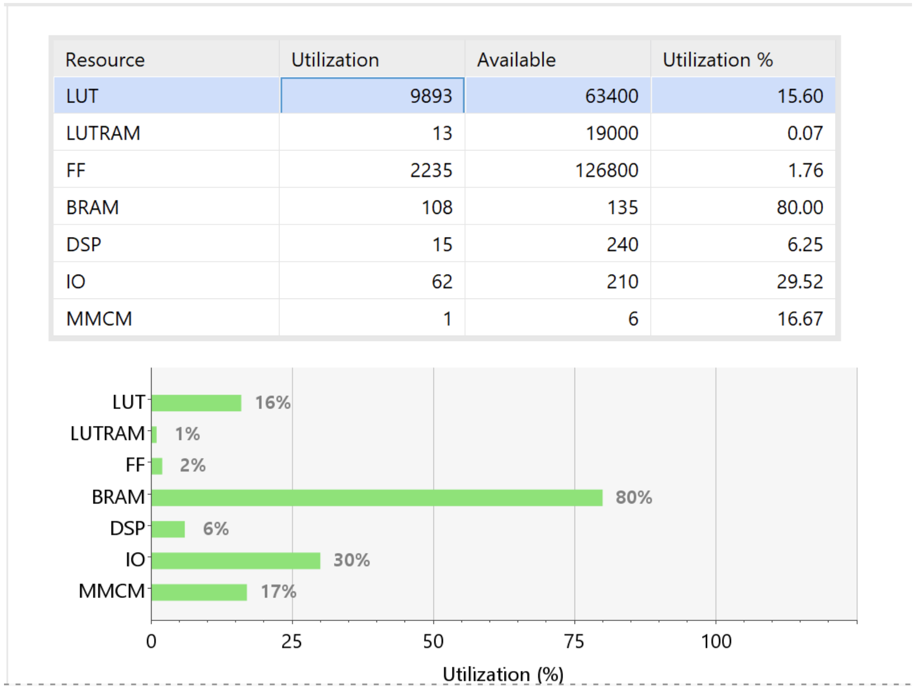
Figure 2: High level module diagram

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.

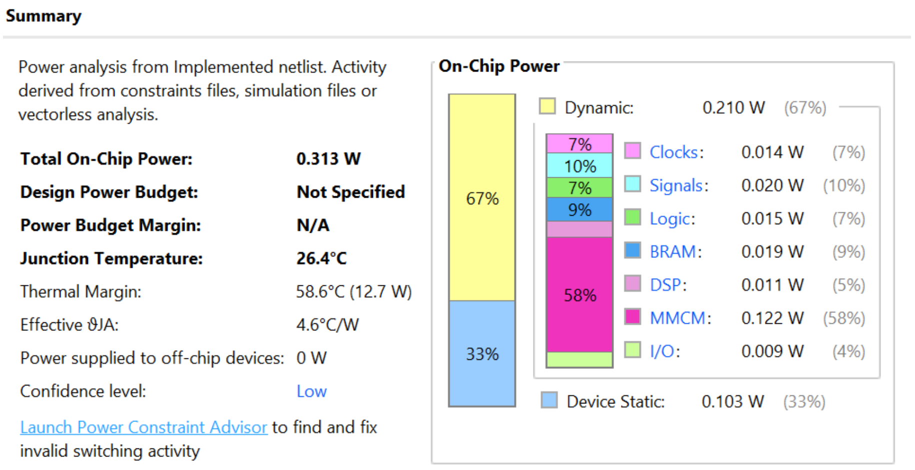
2.3 Utilization

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.



Resource	Utilization	Available	Utilization %
LUT	9893	63400	15.60
LUTRAM	13	19000	0.07
FF	2235	126800	1.76
BRAM	108	135	80.00
DSP	15	240	6.25
IO	62	210	29.52
MMCM	1	6	16.67

Figure 3: Utilization stats for battleship



Summary

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

Total On-Chip Power:	**0.313 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**26.4°C**
Thermal Margin:	58.6°C (12.7 W)
Effective ϑJA:	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

Dynamic:		0.210 W	(67%)
	Clocks:	0.014 W	(7%)
	Signals:	0.020 W	(10%)
	Logic:	0.015 W	(7%)
	BRAM:	0.019 W	(9%)
	DSP:	0.011 W	(5%)
	MMCM:	0.122 W	(58%)
	I/O:	0.009 W	(4%)
Device Static:		0.103 W	(33%)

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, btnl, btnr) and mouse position (mousex, mousey), and converts the inputs to (x, y) coordinates that correspond to map locations.[1] The mouse coordinates were obtained from a PS/2 mouse connected via USB and using the MouseCtl module.[2] 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 btnu 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

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

[2]found on the course website: https://web.mit.edu/6.111/volume2/www/f2020/ps2_mouse/

bit and an stop bit.

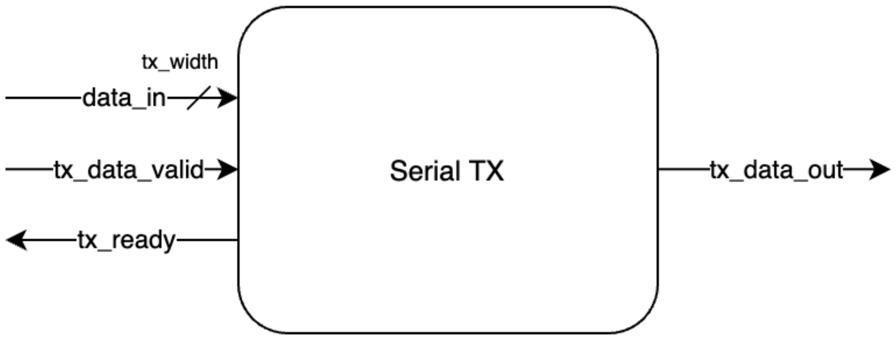


Figure 5: Serial TX module inputs and outputs

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.

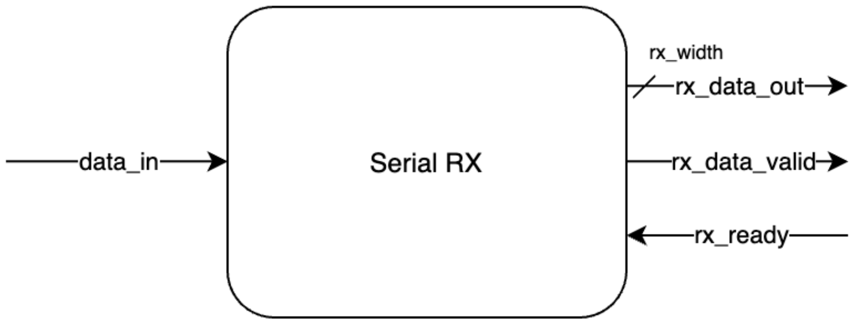


Figure 6: Serial RX module inputs and outputs

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:

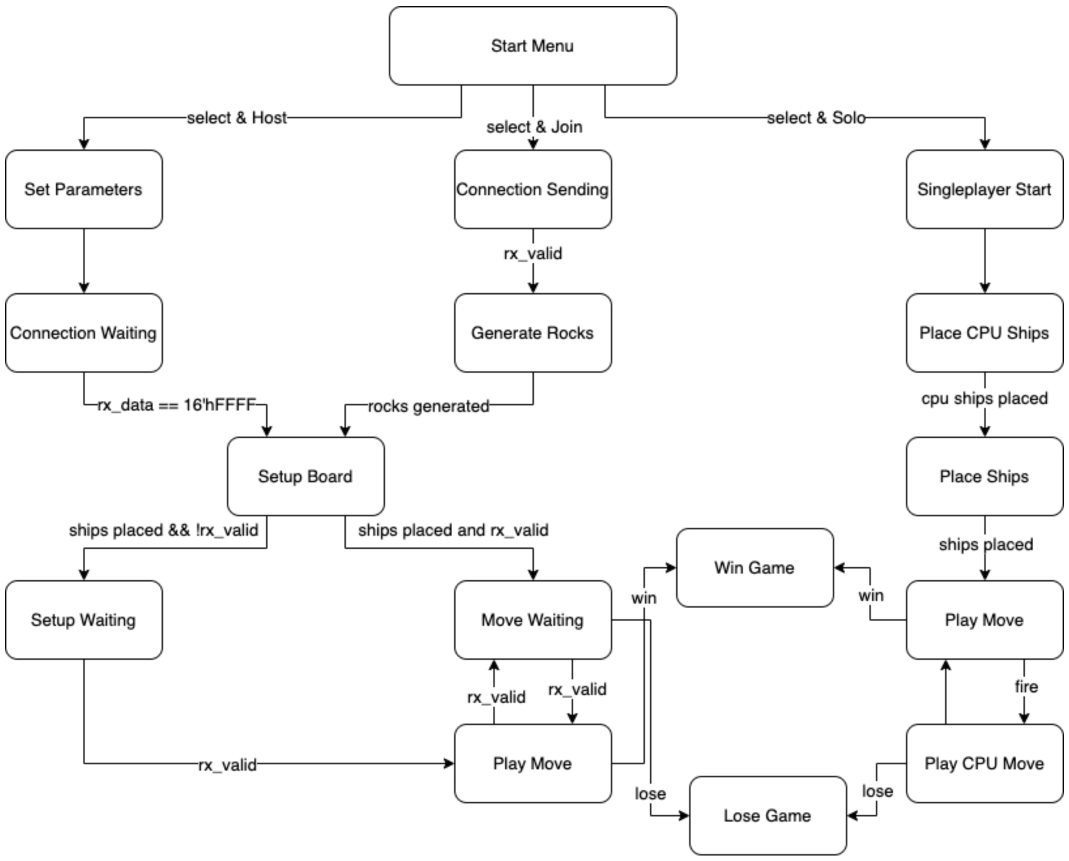


Figure 7: High level Game State FSM

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,[3] 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 generated 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 12×12 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 9×9 board, as there are enough open spaces in a 10×10 or bigger board to prevent this from ever happening.[4] 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.

[3]This could have been done, for example, by having the CPU shoot squares close to previous hits instead of random squares

[4]Proof: there are at least 20 disjoint 5x1 areas to place the last ship, and 19 total items placed before this (12 ship squares and 7 rocks), so one 5x1 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.

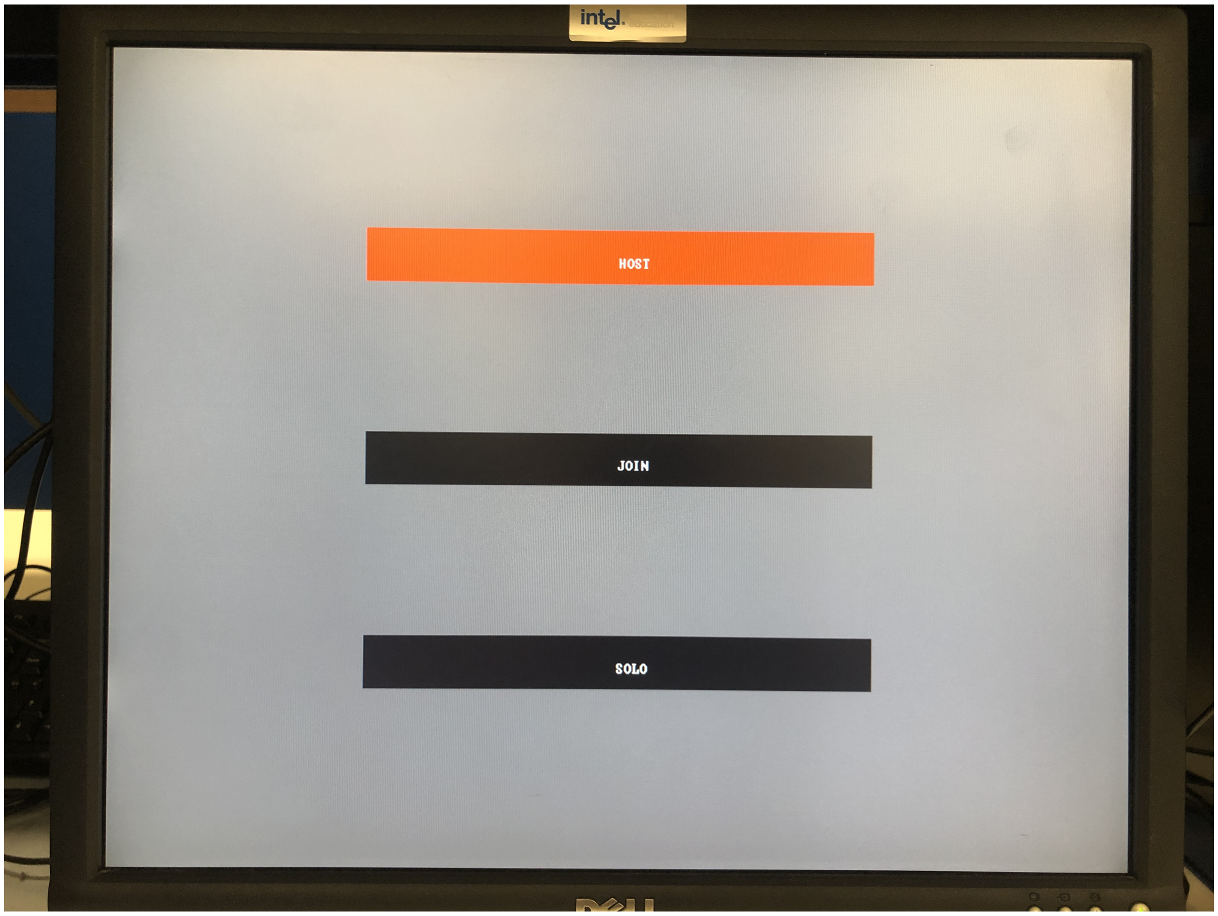


Figure 8: Main menu screen

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.

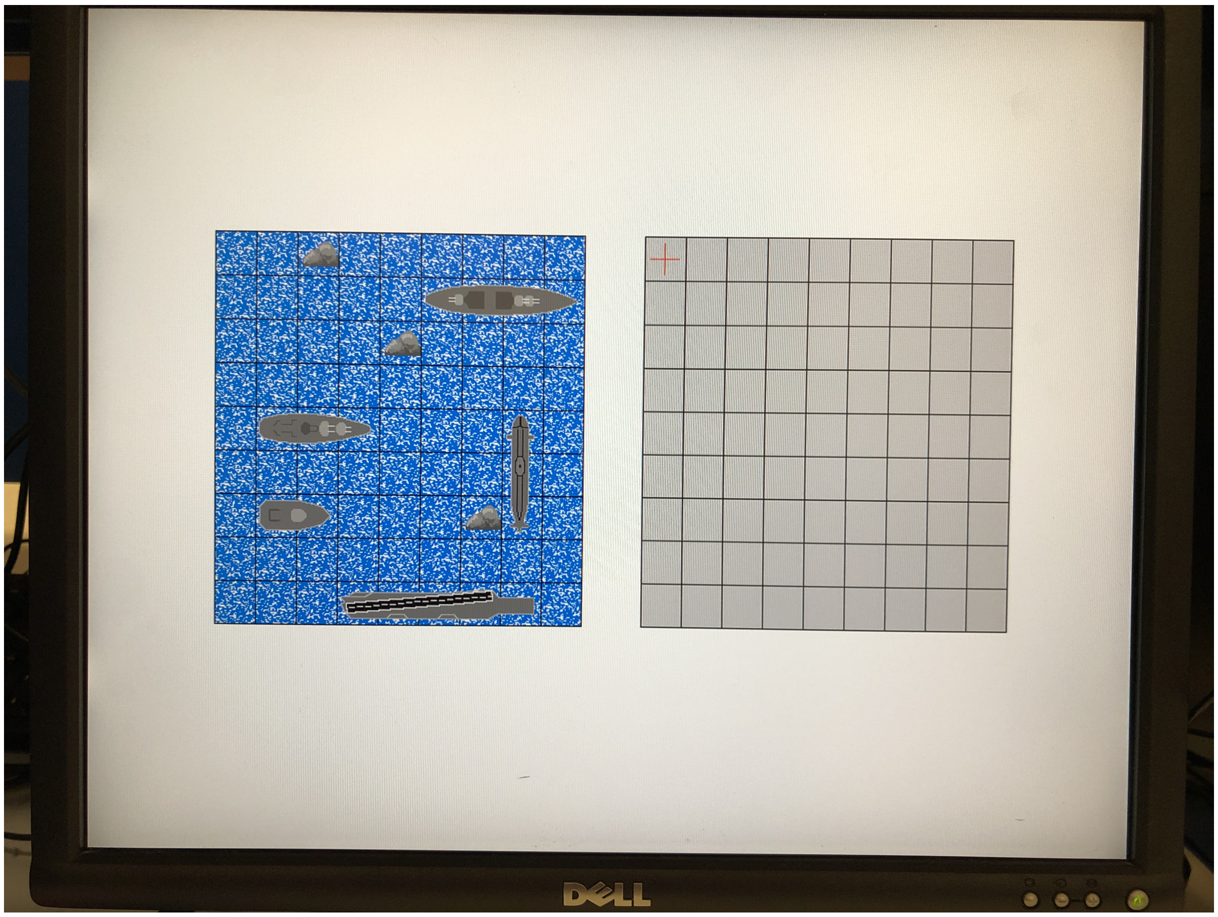


Figure 9: Game view after placing all ships and prepared to fire

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 12x12x3 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 12x12x3 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

```
`default_nettype none

module audio_bgm #(BROM_SIZE = 329286,
                   SAMPLE_RATE_PARAMETER = 10833) (
  input wire clk_in,                // 65MHz system clock
  input wire rst_in,                 // 1 to reset to initial state
  input wire play_audio,               // 1 when we want to start playing audio
  output logic signed [7:0] data_out,        // 8-bit PCM data to headphone
  output logic ready //1 when done with audio
);
    logic sample_clk;

    logic[$clog2(SAMPLE_RATE_PARAMETER)+1:0] sample_counter;
    assign sample_clk = (sample_counter == 0);
    logic prev_play_audio;

    always_ff @(posedge clk_in) begin
        prev_play_audio <= play_audio;
        if (rst_in || sample_counter == SAMPLE_RATE_PARAMETER - 1) begin
            sample_counter <= 0;
        end else begin
            sample_counter <= sample_counter + 1;
        end
    end

    logic [$clog2(BROM_SIZE) - 1 : 0] addr;
    logic [$clog2(BROM_SIZE) - 1 : 0] max_addr;
    logic [7:0] data_from_bram;
    bgm bgm (.addra(addr), .clka(clk_in), .douta(data_from_bram));

    enum {IDLE, PLAYING} state;
    always_ff @(posedge clk_in) begin
        if(rst_in) begin
            addr <= 0;
            data_out <= 0;
            state <= IDLE;
            ready <= 1;
        end
        else begin
            if(state == IDLE) begin
                if(~prev_play_audio & play_audio) begin
                    state <= PLAYING;
                    addr <= 0;
                    ready <= 0;
                end
            end
            else if(state == PLAYING) begin
                if(sample_clk) begin
                    if(addr < BROM_SIZE) begin
                        addr <= addr + 1;
                        data_out <= {~data_from_bram[7], data_from_bram[6:0]};
                        //convert from signed binary bram
                    end
                    else begin //looping bgm
                        addr <= 0;
//                        ready <= 1;
//                        state <= IDLE;
                    end
                end
            end
        end
    end
endmodule

`default_nettype wire
```

7.1.2 audio_explosion.sv

```
module audio_explosion #(BROM_SIZE = 19511,
                 SAMPLE_RATE_PARAMETER = 10833) (
  input logic clk_in,              // 65MHz system clock
  input logic rst_in,               // 1 to reset to initial state
  input logic play_audio,             // 1 when we want to start playing audio
  output logic signed [7:0] data_out,       // 8-bit PCM data to headphone
  output logic ready //1 when done with audio
);
    logic[$clog2(SAMPLE_RATE_PARAMETER)+1:0] sample_counter;
    assign sample_clk = (sample_counter == 0);
    logic prev_play_audio;

    always_ff @(posedge clk_in) begin
        prev_play_audio <= play_audio;
        if (rst_in || sample_counter == SAMPLE_RATE_PARAMETER - 1) begin
            sample_counter <= 0;
        end else begin
            sample_counter <= sample_counter + 1;
        end
    end

    logic [$clog2(BROM_SIZE) - 1 : 0] addr;
    logic [$clog2(BROM_SIZE) - 1 : 0] max_addr;
    logic [7:0] data_from_bram;
    explosion_audio explosion_audio (.addra(addr), .clka(clk_in),
        ↪ .douta(data_from_bram));

    enum {IDLE, PLAYING} state;
    always_ff @(posedge clk_in) begin
        if(rst_in) begin
            addr <= 0;
            data_out <= 0;
            state <= IDLE;
            ready <= 1;
        end
        else begin
            if(state == IDLE) begin
                if(~prev_play_audio & play_audio) begin
                    state <= PLAYING;
                    addr <= 0;
                    ready <= 0;
                end
            end
            else if(state == PLAYING) begin
                if(sample_clk) begin
                    if(addr < BROM_SIZE) begin
                        addr <= addr + 1;
                        data_out <= {~data_from_bram[7], data_from_bram[6:0]};
                        //convert from signed binary bram
                    end
                    else begin
                        ready <= 1;
                        state <= IDLE;
                    end
                end
            end
        end
    end
endmodule
```

7.1.3 audio_gen.sv

```
module audio_gen(input wire clk,
                 input wire reset,
                 input wire bgm_on,
                 input wire sounds_on,
                 input wire play_explosion,
                 input wire play_miss,
                 input wire play_bgm,
                 input wire[2:0] volume_control,
                 output logic aud_pwm,
                 output logic aud_sd);
    assign aud_sd = 1;
    logic pwm_val;

    logic [7:0] vol_out;
    logic signed [7:0] bgm_out;
    logic signed [7:0] exp_out;
    logic signed [7:0] miss_out;

    audio_bgm bgm(.clk_in(clk), .rst_in(reset), .play_audio(play_bgm),
       ↪ .data_out(bgm_out), .ready());
    audio_explosion explosion(.clk_in(clk), .rst_in(reset),
       ↪ .play_audio(play_explosion), .data_out(exp_out), .ready());
    audio_miss miss(.clk_in(clk), .rst_in(reset), .play_audio(play_miss),
       ↪ .data_out(miss_out), .ready());

    logic signed [7:0] bgm_quiet;
    assign bgm_quiet = bgm_out >>> 1;
    logic signed [7:0] audio_signal;
    always_comb begin
        if(bgm_on && sounds_on)
            audio_signal = bgm_quiet + ((miss_out + exp_out) >>> 1);
        else if(!bgm_on && sounds_on)
            audio_signal = miss_out + exp_out;
        else if(bgm_on && !sounds_on)
            audio_signal = bgm_out;
        else audio_signal = 0;
    end

    volume_control vc (.vol_in(volume_control), .signal_in(audio_signal),
       ↪ .signal_out(vol_out));
    pwm pwm (.clk_in(clk), .rst_in(reset), .level_in({~vol_out[7],vol_out[6:0]}),
       ↪ .pwm_out(pwm_val));

    assign aud_pwm = pwm_val?1'bZ:1'b0;
endmodule



`default_nettype wire

//Volume Control
module volume_control (input [2:0] vol_in, input signed [7:0] signal_in, output
   ↪ logic signed[7:0] signal_out);
    logic [2:0] shift;
    assign shift = 3'd7 - vol_in;
    assign signal_out = signal_in>>>shift;
endmodule
```

```

//PWM generator for audio generation!
module pwm (input clk_in, input rst_in, input [7:0] level_in, output logic
    ↪ pwm_out);
    logic [7:0] count;
    assign pwm_out = count<level_in;
    always_ff @(posedge clk_in)begin
        if (rst_in)begin
            count <= 8'b0;
        end else begin
            count <= count+8'b1;
        end
    end
endmodule
```

7.1.4 audio_miss.sv

```
module audio_miss #(BROM_SIZE = 10943,
                SAMPLE_RATE_PARAMETER = 10833) (
  input logic clk_in,              // 65MHz system clock
  input logic rst_in,               // 1 to reset to initial state
  input logic play_audio,             // 1 when we want to start playing audio
  output logic signed [7:0] data_out,       // 8-bit PCM data to headphone
  output logic ready //1 when done with audio
);
    logic[$clog2(SAMPLE_RATE_PARAMETER)+1:0] sample_counter;
    assign sample_clk = (sample_counter == 0);
    logic prev_play_audio;

    always_ff @(posedge clk_in) begin
        prev_play_audio <= play_audio;
        if (rst_in || sample_counter == SAMPLE_RATE_PARAMETER - 1) begin
            sample_counter <= 0;
        end else begin
            sample_counter <= sample_counter + 1;
        end
    end

    logic [$clog2(BROM_SIZE) - 1 : 0] addr;
    logic [$clog2(BROM_SIZE) - 1 : 0] max_addr;
    logic [7:0] data_from_bram;
    miss miss (.addra(addr), .clka(clk_in), .douta(data_from_bram));

    enum {IDLE, PLAYING} state;
    always_ff @(posedge clk_in) begin
        if(rst_in) begin
            addr <= 0;
            data_out <= 0;
            state <= IDLE;
            ready <= 1;
        end
        else begin
            if(state == IDLE) begin
                if(~prev_play_audio & play_audio) begin
                    state <= PLAYING;
                    addr <= 0;
                    ready <= 0;
                end
            end
            else if(state == PLAYING) begin
                if(sample_clk) begin
                    if(addr < BROM_SIZE) begin
                        addr <= addr + 1;
                        data_out <= {~data_from_bram[7], data_from_bram[6:0]};
                        //convert from signed binary bram
                    end
                    else begin
                        ready <= 1;
                        state <= IDLE;
                    end
                end
            end
        end
    end
```

```
endmodule
```

7.1.5 base_board.sv

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
`default_nettype none

module base_board
    #(parameter SQ_SIZE = 40, CLEAR_PIXEL = 12'h0F0)
    (  input wire clk_in, rst_in,
       input wire [3:0] size,
       input wire [10:0] hcount_in, x_offset,
       input wire [9:0] vcount_in, y_offset,
       output logic [11:0] pixel_out
    );

    localparam spacing = 62;
    logic [11:0] grid_pixel;
    logic [11:0] water_pixel;

    water water_block(.clk_in(clk_in), .rst_in(rst_in), .hcount_in(hcount_in), .vcount_in(vcount_in), .pixel_out(water_pixel));

    always_comb begin
      if (
         ((vcount_in >= y_offset && vcount_in <= SQ_SIZE*size + y_offset && (hcount_in >= x_offset && hcount_in <= SQ_SIZE*size + x_offset))
          &&
         (((vcount_in == SQ_SIZE*0 + y_offset || vcount_in == SQ_SIZE*1 + y_offset || vcount_in == SQ_SIZE*2 + y_offset || vcount_in == SQ_SIZE*3 + y_offset || vcount_in == SQ_SIZE*4 + y_offset ||
          vcount_in == SQ_SIZE*5 + y_offset || vcount_in == SQ_SIZE*6 + y_offset || vcount_in == SQ_SIZE*7 + y_offset || vcount_in == SQ_SIZE*8 + y_offset || vcount_in == SQ_SIZE*9 + y_offset ||
          vcount_in == SQ_SIZE*10 + y_offset || vcount_in == SQ_SIZE*11 + y_offset|| vcount_in == SQ_SIZE*12 + y_offset)
          ||
         (hcount_in == SQ_SIZE*0 + x_offset || hcount_in == SQ_SIZE*1 + x_offset || hcount_in == SQ_SIZE*2 + x_offset || hcount_in == SQ_SIZE*3 + x_offset || hcount_in == SQ_SIZE*4 + x_offset ||
          hcount_in == SQ_SIZE*5 + x_offset || hcount_in == SQ_SIZE*6 + x_offset || hcount_in == SQ_SIZE*7 + x_offset || hcount_in == SQ_SIZE*8 + x_offset || hcount_in == SQ_SIZE*9 + x_offset ||
          hcount_in == SQ_SIZE*10 + x_offset || hcount_in == SQ_SIZE*11 + x_offset || hcount_in == SQ_SIZE*12 + x_offset))))

         ||

         (((vcount_in >= y_offset && vcount_in <= SQ_SIZE*size + y_offset) && (hcount_in >= 512 + spacing/2 && hcount_in <= 512 + spacing/2 + SQ_SIZE*size))
         &&
         (((vcount_in == SQ_SIZE*0 + y_offset || vcount_in == SQ_SIZE*1 + y_offset || vcount_in == SQ_SIZE*2 + y_offset || vcount_in == SQ_SIZE*3 + y_offset || vcount_in == SQ_SIZE*4 + y_offset ||
          vcount_in == SQ_SIZE*5 + y_offset || vcount_in == SQ_SIZE*6 + y_offset || vcount_in == SQ_SIZE*7 + y_offset || vcount_in == SQ_SIZE*8 + y_offset || vcount_in == SQ_SIZE*9 + y_offset ||
          vcount_in == SQ_SIZE*10 + y_offset || vcount_in == SQ_SIZE*11 + y_offset|| vcount_in == SQ_SIZE*12 + y_offset)
          ||
         (hcount_in == 512 + spacing/2 + SQ_SIZE*0 || hcount_in == 512 +
            ↪ spacing/2 + SQ_SIZE*1 || hcount_in == 512 + spacing/2 + SQ_SIZE*2
            ↪ ||
          hcount_in == 512 + spacing/2 + SQ_SIZE*3 || hcount_in == 512 +
             ↪ spacing/2 + SQ_SIZE*4 || hcount_in == 512 + spacing/2 + SQ_SIZE*5
             ↪ ||
          hcount_in == 512 + spacing/2 + SQ_SIZE*6 || hcount_in == 512 +
             ↪ spacing/2 + SQ_SIZE*7 || hcount_in == 512 + spacing/2 + SQ_SIZE*8
             ↪ ||
          hcount_in == 512 + spacing/2 + SQ_SIZE*9 || hcount_in == 512 +
             ↪ spacing/2 + SQ_SIZE*10 || hcount_in == 512 + spacing/2 +
             ↪ SQ_SIZE*11 ||
          hcount_in == 512 + spacing/2 + SQ_SIZE*12))))
          )

          pixel_out = 12'h000;
      else if(((vcount_in >= y_offset && vcount_in <= SQ_SIZE*size + y_offset) &&
         ↪ ((hcount_in >= x_offset && hcount_in <= SQ_SIZE*size + x_offset) ||
                                                                                  (hcount_in
                                                                                     ↪ >=
                                                                                     ↪ 512
                                                                                     ↪ +
                                                                                     ↪ spacing
                                                                                     ↪ &&
                                                                                     ↪ hcount_
                                                                                     ↪ <=
                                                                                     ↪ 512
                                                                                     ↪ +
                                                                                     ↪ spacing
                                                                                     ↪ +
                                                                                     ↪ SQ_SIZE
                                                                                     ↪ begin
          pixel_out = water_pixel;
      //Not the board
      end else pixel_out = CLEAR_PIXEL;
   end

endmodule

`default_nettype wire
```

7.1.6 blob.sv

```
`default_nettype none

module blob
   #(parameter WIDTH = 64,            // default width: 64 pixels
               HEIGHT = 64,           // default height: 64 pixels
               COLOR = 12'hFFF)  // default color: white
   (input wire [10:0] x_in,hcount_in,
    input wire [9:0] y_in,vcount_in,
    output logic [11:0] pixel_out);

   always_comb begin
      if  ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
            (vcount_in >= y_in && vcount_in < (y_in+HEIGHT)))
        pixel_out = COLOR;
      else
        pixel_out = 0;
   end
endmodule

`default_nettype wire
```

7.1.7 boat2.sv

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
`default_nettype none

module boat2( input wire clk_in,
              input wire [10:0] hcount_in, x_in,
              input wire [9:0] vcount_in, y_in,
              input wire flipped,

              output wire [11:0] pixel_out);

    localparam w = 81;
    localparam h = 40;

    logic [7:0] image_bits, red_mapped;
    logic [$clog2(w*h)-1:0] image_addr;

    picture_blob #(.WIDTH(w), .HEIGHT(h)) b2 (.pixel_clk_in(clk_in), .hcount_in(hcount_in), .vcount_in(vcount_in), .x_in(x_in), .y_in(y_in),
                                              .flip(flipped), .red_mapped(red_mapped), .image_addr(image_addr), .pixel_out(pixel_out));

    b2_image b2_image (.clka(clk_in), .addra(image_addr), .douta(image_bits));
    b2_red b2_color(.clka(clk_in), .addra(image_bits), .douta(red_mapped));

endmodule

`default_nettype wire
```

7.1.8 boat3a.sv

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
`default_nettype none

module boat3a(input wire clk_in, rst_in,
              input wire [10:0] hcount_in, x_in,
              input wire [9:0] vcount_in, y_in,
              input wire flipped,

              output wire [11:0] pixel_out);

    localparam w = 120;
    localparam h = 40;

    logic [7:0] image_bits, red_mapped;
    logic [$clog2(w*h)-1:0] image_addr;

    picture_blob #(.WIDTH(w), .HEIGHT(h)) b3a (.pixel_clk_in(clk_in), .hcount_in(hcount_in), .vcount_in(vcount_in), .x_in(x_in), .y_in(y_in),
                                               .flip(flipped), .red_mapped(red_mapped), .image_addr(image_addr), .pixel_out(pixel_out));

    b3a_image b3a_image (.clka(clk_in), .addra(image_addr), .douta(image_bits));
    b3a_red b3a_color(.clka(clk_in), .addra(image_bits), .douta(red_mapped));

endmodule

`default_nettype wire
```

7.1.9 boat3b.sv

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
`default_nettype none

module boat3b(input wire clk_in, rst_in,
              input wire [10:0] hcount_in, x_in,
              input wire [9:0] vcount_in, y_in,
              input wire flipped,

              output wire [11:0] pixel_out);

    localparam w = 120;
    localparam h = 40;

    logic [7:0] image_bits, red_mapped;
    logic [$clog2(w*h)-1:0] image_addr;

    picture_blob #(.WIDTH(w), .HEIGHT(h)) b3b (.pixel_clk_in(clk_in), .hcount_in(hcount_in), .vcount_in(vcount_in), .x_in(x_in), .y_in(y_in),
                                               .flip(flipped), .red_mapped(red_mapped), .image_addr(image_addr), .pixel_out(pixel_out));

    b3b_image b3b_image (.clka(clk_in), .addra(image_addr), .douta(image_bits));
    b3b_red b3b_color(.clka(clk_in), .addra(image_bits), .douta(red_mapped));

endmodule

`default_nettype wire
```

7.1.10 boat4.sv

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
`default_nettype none

module boat4(input wire clk_in, rst_in,
              input wire [10:0] hcount_in, x_in,
              input wire [9:0] vcount_in, y_in,
              input wire flipped,

              output wire [11:0] pixel_out);

    localparam w = 161;
    localparam h = 40;

    logic [7:0] image_bits, red_mapped;
    logic [$clog2(w*h)-1:0] image_addr;

    picture_blob #(.WIDTH(w), .HEIGHT(h)) b4 (.pixel_clk_in(clk_in), .hcount_in(hcount_in), .vcount_in(vcount_in), .x_in(x_in), .y_in(y_in),
                                               .flip(flipped), .red_mapped(red_mapped), .image_addr(image_addr), .pixel_out(pixel_out));

    b4_image b4_image (.clka(clk_in), .addra(image_addr), .douta(image_bits));
    b4_red b4_color(.clka(clk_in), .addra(image_bits), .douta(red_mapped));

endmodule

`default_nettype wire
```

7.1.11 boat5.sv

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
`default_nettype none

module boat5(input wire clk_in, rst_in,
              input wire [10:0] hcount_in, x_in,
              input wire [9:0] vcount_in, y_in,
              input wire flipped,

              output wire [11:0] pixel_out);

    localparam w = 201;
    localparam h = 41;

    logic [7:0] image_bits, red_mapped;
    logic [$clog2(w*h)-1:0] image_addr;

    picture_blob #(.WIDTH(w), .HEIGHT(h)) b5 (.pixel_clk_in(clk_in), .hcount_in(hcount_in), .vcount_in(vcount_in), .x_in(x_in), .y_in(y_in),
                                               .flip(flipped), .red_mapped(red_mapped), .image_addr(image_addr), .pixel_out(pixel_out));

    b5_image b5_image (.clka(clk_in), .addra(image_addr), .douta(image_bits));
    b5_red b5_color(.clka(clk_in), .addra(image_bits), .douta(red_mapped));

endmodule

`default_nettype wire
```

7.1.12 coverage.sv

```
`default_nettype none

module coverage #(parameter SQ_SIZE = 40)
                (input wire clk_in, rst_in,
                input wire [10:0] hcount_in, x_offset,
                input wire [9:0] vcount_in, y_offset,
                input wire [2:0] player_board [11:0][11:0],
                input wire [2:0] opponent_board [11:0][11:0],
                input wire [3:0] game_size,
                output logic [11:0] pixel_out
                );

    logic [11:0] p_pixel;
    logic [11:0] o_pixel;

    logic [10:0] px;
    logic [9:0] py;

    logic [10:0] ox;
    logic [9:0] oy;

    localparam spacing = 62;

    hv_to_xy #(.SQ_SIZE(SQ_SIZE)) player (.hcount_in(hcount_in),
        ↪ .start_hcount(x_offset), .vcount_in(vcount_in),
        ↪ .start_vcount(y_offset), .x_out(px), .y_out(py));
    hv_to_xy #(.SQ_SIZE(SQ_SIZE)) opponent (.hcount_in(hcount_in),
        ↪ .start_hcount(512+spacing/2), .vcount_in(vcount_in),
        ↪ .start_vcount(y_offset), .x_out(ox), .y_out(oy));

    hit_or_miss hmp (.clk_in(clk_in), .rst_in(rst_in), .x_in(px*SQ_SIZE +
        ↪ x_offset),  .y_in(py*SQ_SIZE + y_offset), .hcount_in(hcount_in),
        ↪ .vcount_in(vcount_in), .hit(player_board[px][py]), .pixel_out(p_pixel));
    hit_or_miss hmo (.clk_in(clk_in), .rst_in(rst_in),
        ↪ .x_in(ox*SQ_SIZE+512+spacing/2),  .y_in(oy*SQ_SIZE + y_offset),
        ↪ .hcount_in(hcount_in), .vcount_in(vcount_in),
        ↪ .hit(opponent_board[ox][oy]), .pixel_out(o_pixel));

    always_ff @(posedge clk_in) begin
        if (rst_in) begin
            pixel_out <= 12'hFFF;
        end else begin
            pixel_out <= (px < game_size && py < game_size && p_pixel) ? p_pixel :
                         (ox < game_size && oy < game_size && o_pixel) ? o_pixel :
                         12'h000;
        end
    end

endmodule

`default_nettype wire
```

7.1.13 debouncer.sv

```
module debouncer (input wire reset, clk_in, noisy_in,
                  output logic clean_out);

   parameter DEBOUNCE_COUNT = 650000;
   logic [19:0] count;
   logic new_input;

   always_ff @(posedge clk_in)
     if (reset) begin
        new_input <= noisy_in;
        clean_out <= 0;
        count <= 0; end
     else if (noisy_in != new_input) begin new_input<=noisy_in; count <= 0; end
     else if (count == DEBOUNCE_COUNT) clean_out <= new_input;
     else count <= count+1;
endmodule
```

7.1.14 explosion.sv

```systemverilog
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 11/23/2021 10:22:29 PM
// Design Name:
// Module Name: explosion
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
`default_nettype none

module explosion( input wire clk_in, rst_in,
                  input wire [10:0] hcount_in, x_in,
                  input wire [9:0] vcount_in, y_in,

                  output wire [11:0] pixel_out);

    localparam w = 40;
    localparam h = 40;

    logic [7:0] red_mapped, green_mapped, blue_mapped, image_bits;
    logic [$clog2(w*h)-1:0] image_addr;

    picture_blob #(.WIDTH(w), .HEIGHT(h), .GREYSCALE(1'b0)) e1 (.pixel_clk_in(clk_in), .hcount_in(hcount_in), .vcount_in(vcount_in),
                                                                .x_in(x_in), .y_in(y_in), .flip(1'b0), .red_mapped(red_mapped),
                                                                .green_mapped(green_mapped), .blue_mapped(blue_mapped) .image_addr(image_addr),
                                                                .pixel_out(pixel_out));

    explosion1_image e1_image(.clka(clk_in), .addra(image_addr), .douta(image_bits));
    explosion1_red e1_r(.clka(clk_in), .addra(image_bits), .douta(red_mapped));
    explosion1_blue e1_b(.clka(clk_in), .addra(image_bits), .douta(blue_mapped));
    explosion1_green e1_g(.clka(clk_in), .addra(image_bits), .douta(green_mapped));

endmodule

`default_nettype wire
```

7.1.15 fsm_state_pkg.sv

```
package fsm_state; // package name
    typedef enum {START_MENU, SET_GAME_PARAMETERS,
                CONNECTION_WAITING, CONNECTION_SENDING,
                SETUP_BOARD, SETUP_WAITING, PLAY_MOVE,
                MOVE_WAITING, END_GAME_WIN, END_GAME_LOSE,
                PLAY_STALL, PLAY_STALL_2, WAIT_MOVE_WAITING,
                WAIT_STALL, WAIT_STALL_2, RESET_GAME, GEN_ROCKS_RECEIVER,
                SINGLEPLAYER_START, SINGLEPLAYER_RANDOM_STALL,
                SINGLEPLAYER_PLACE_CPU_SHIPS, SINGLEPLAYER_PLACE_SHIPS,
                SINGLEPLAYER_PLAY_MOVE, SINGLEPLAYER_UPDATE_SUNK_SHIP,
                SINGLEPLAYER_PLAY_MOVE_STALL, SINGLEPLAYER_OPPONENT_PLAY_MOVE}
                    ↪ game_state;

    typedef struct {
        logic[3:0] x;
        logic[3:0] y;
        logic[3:0] length;
        logic orientation; //1 for horizontal, 0 for vertical
        logic[3:0] num_hits; //counter for number of ship squares hit
    } ship;

    typedef ship ship_array[5];
endpackage
```

7.1.16 game_state_controller.sv

```
`default_nettype none
import fsm_state::*;

module game_state_controller(input wire reset, clk,
                             input wire btnd, btnc, btnu, btnl, btnr,
                             input wire rotate_ship, input wire [3:0]
                                 ↪ game_size_switch,
                             input wire rx_in, output logic tx_out,
                             input wire [2:0] volume_control,
                             input wire sounds_on, bgm_on,
                             input wire bgm_test,
                             input wire [11:0] mousex,
                             input wire [11:0] mousey,
                             input wire [2:0] num_rocks,
                             input wire mouse_enable,
                             input wire hard_mode,
                             output logic [3:0] vga_r, vga_g, vga_b,
                             output logic vga_hs, vga_vs,
                             output logic aud_pwm, aud_sd,
                             output logic [31:0] debug);

    logic [15:0] rx_data;
    logic [15:0] tx_data;
    logic rx_valid;
    logic tx_valid;
    logic rx_ready;
    logic tx_ready;

    game_state state;
    ship_array ships;
    logic[2:0] curr_ship_ctr; //which ship is being placed
    logic placeable_out;
    logic[3:0] x, y, game_size; //x, y coordinates and game size

    logic [7:0] debug_g;
    assign debug = {btnc, btnu, btnr, btnl, btnd, x, y, 1'b0, curr_ship_ctr,
        ↪ 3'b000, state[4:0]}; //for debugging on 8hex display

    //keeping track of board state
    logic[2:0] opponent_board [11:0][11:0];
    logic[2:0] player_board [11:0][11:0];

    logic play_explosion;
    logic play_miss;
    logic play_bgm;

    //reset logic for after game ends; passes normal sw[15] if game not over
    logic reset_in;
    logic game_end;
    assign game_end = state == RESET_GAME;
    parameter RESET_CLK_COUNT = 650000;
    int reset_ctr;
    always_ff @(posedge clk) begin
        if(reset) begin
            reset_in <= 1;
            reset_ctr <= RESET_CLK_COUNT;
        end
```

```verilog
        else if (game_end) begin
            reset_in <= 1;
            reset_ctr <= 0;
        end
        else if(reset_ctr >= RESET_CLK_COUNT)
            reset_in <= 0;
        else reset_ctr <= reset_ctr + 1;
    end


    rx rx_module (.clk_in(clk), .rst_in(reset_in), .rx_in(rx_in),
                  .ready_in(rx_ready), .valid_out(rx_valid), .data_out(rx_data));
    tx tx_module (.clk_in(clk), .rst_in(reset_in), .tx_out(tx_out),
                  .ready_out(tx_ready), .valid_in(tx_valid), .data_in(tx_data));

    user_input ui(.reset(reset_in), .clk(clk), .btnd(btnd),
               .btnu(btnu), .btnr(btnr), .btnl(btnl),
               .size(game_size), .x(x), .y(y), .rotate(rotate_ship),
               .ship_num(curr_ship_ctr), .ships(ships), .state(state),
               .mousex(mousex), .mousey(mousey), .mouse_enable(mouse_enable));

    game_state_fsm fsm(.reset(reset_in), .clk(clk), .btnc(btnc), .x(x), .y(y), .rotate(rotate_ship), //user inputs
                     .game_size_switch(game_size_switch), .hard_mode_in(hard_mode), //game settings input
                     .game_size(game_size), .ship_state(curr_ship_ctr), //game settings output
                     .ships(ships), .state(state), .num_rocks_in(num_rocks),
                     .tx_data(tx_data), .tx_valid(tx_valid), .tx_ready(tx_ready), //communication
                     .rx_data(rx_data), .rx_ready(rx_ready), .rx_valid(rx_valid), //communication
                     .player_board(player_board), .opponent_board(opponent_board),
                     .play_bgm(play_bgm), .play_miss(play_miss), .play_explosion(play_explosion));

    graphics_prep gp (.clk_65mhz(clk), .rst_in(reset_in), .ships(ships), .game_size(game_size), .state_in(state),
                    .x_in(x), .y_in(y), .vga_r(vga_r), .vga_g(vga_g), .vga_b(vga_b), .vga_hs(vga_hs), .vga_vs(vga_vs),
                    .opponent_board(opponent_board), .player_board(player_board), .debug(debug_g), .ship_state(curr_ship_ctr),
                    .mousex(mousex), .mousey(mousey), .mouse_enable(mouse_enable));

    audio_gen audio (.clk(clk), .reset(reset_in), .bgm_on(bgm_on), .sounds_on(sounds_on), .play_explosion(play_explosion), .play_miss(play_miss),
              .play_bgm(play_bgm), .volume_control(volume_control), .aud_pwm(aud_pwm), .aud_sd(aud_sd));

endmodule

`default_nettype wire
```

7.1.17 game_state_fsm.sv

```
`default_nettype none
import fsm_state::*;

module game_state_fsm(input wire reset, clk, btnc,
                      input wire[3:0] x, y, game_size_switch,
                      input wire rotate,
                      input wire[15:0] rx_data,
                      input wire rx_valid,
                      input wire hard_mode_in,
                      input wire[2:0] num_rocks_in,
                      output logic rx_ready,
                      output logic tx_valid,
                      output logic[15:0] tx_data,
                      input wire tx_ready,
                      output game_state state,
                      output logic[3:0] game_size,
                      output logic[2:0] ship_state,
                      output ship_array ships,
                      output logic[2:0] player_board [11:0][11:0],
                      output logic[2:0] opponent_board [11:0][11:0],
                      output logic play_explosion, play_miss, play_bgm);

    logic old_btnc;
    logic start_rng;
    // opponent_board = which positions have been fired at
    // opponent_board[x][y] = 00 if not fired
    //                      = 01 if HIT
    //                      = 10 if MISS
    //                      = 11 if ROCK
    //                      = 100 if SUNK
    // same for player_board

    logic placeable_out;
    logic rocks_out;
    logic[7:0] player_rocks [7:0];
    logic[2:0] num_rocks;
    logic[7:0] random_rock;
    logic start_rock_gen;
    logic done_rock_gen;
    rock_randomizer rock_randomizer(.reset(reset), .clk(clk),
       ↪ .rng_input(rng_out), .start(start_rock_gen), .done(done_rock_gen),
                                    .player_rocks_out(player_rocks),
                                       ↪ .cpu_rocks_out(cpu_rocks));
    rock_coords rocks_coord_checker(.x(rx_data[7:4]), .y(rx_data[3:0]),
       ↪ .rocks_in(player_rocks),
                                    .placeable(rocks_out), .num_rocks(num_rocks));
    placeable coord_checker(.x(x), .y(y), .ship_num(ship_state),
       ↪ .rocks_in(player_rocks), // combinational check if ship is placeable at
       ↪ that (x,y)
                            .ships(ships), .check_one_coord(1'b0),
                               ↪ .placeable(placeable_out),
                               ↪ .num_rocks(num_rocks));

    logic ship_out [4:0]; //1 if the (x,y) position is a hit, 0 otherwise.
    // used when opponent is attacking to check positions
    ship_coords ship0_coord_checker(.x(rx_data[7:4]), .y(rx_data[3:0]),
       ↪ .ships(ships), .ship_num(4'd0), .ship_out(ship_out[0]));
    ship_coords ship1_coord_checker(.x(rx_data[7:4]), .y(rx_data[3:0]),
       ↪ .ships(ships), .ship_num(4'd1), .ship_out(ship_out[1]));
    ship_coords ship2_coord_checker(.x(rx_data[7:4]), .y(rx_data[3:0]),
       ↪ .ships(ships), .ship_num(4'd2), .ship_out(ship_out[2]));
    ship_coords ship3_coord_checker(.x(rx_data[7:4]), .y(rx_data[3:0]),
       ↪ .ships(ships), .ship_num(4'd3), .ship_out(ship_out[3]));
    ship_coords ship4_coord_checker(.x(rx_data[7:4]), .y(rx_data[3:0]),
       ↪ .ships(ships), .ship_num(4'd4), .ship_out(ship_out[4]));

    logic[4:0] num_player_squares_hit;
    logic[4:0] num_opponent_squares_hit;

    logic[3:0] rng_out [31:0];
    logic[31:0] unbounded_out;
    logic[31:0] stall_ctr;
    bounded_rng rng(.reset(reset), .clk(clk), .start(start_rng),
                    .initial_seed(32'hDEADBEEF), .num_max(game_size), //random
                       ↪ RNG starting seed, randomized by clock cycle
                    .numbers_out(rng_out), .unbounded_out(unbounded_out));

    ship_array cpu_ships;
    logic[7:0] cpu_rocks[7:0];
    logic[2:0] cpu_ship_state;
    logic cpu_placeable;
    logic cpu_rocks_out;
    logic[2:0] cpu_sunk_ship; //ship number or
                              //7 for no sunk ship
    logic[7:0] sunk_ship_coords;
    logic[2:0] sunk_ship_length;
    logic hit_player_ship;
    logic hard_mode;
    logic[1:0] num_cpu_hits;
    rock_coords cpu_rocks_coord_checker(.x(x), .y(y), .rocks_in(cpu_rocks),
       ↪ .num_rocks(num_rocks), .placeable(cpu_rocks_out));
    placeable cpu_coord_checker(.x(rng_out[0][3:0]), .y(rng_out[1][3:0]),
       ↪ .ship_num(cpu_ship_state), .check_one_coord(1'b0),
                                .rocks_in(cpu_rocks), .num_rocks(num_rocks),
                                   ↪ .ships(cpu_ships),
                                   ↪ .placeable(cpu_placeable));

    placeable cpu_hit_player(.x(rng_out[0][3:0]), .y(rng_out[1][3:0]),
       ↪ .ship_num(3'd5), .check_one_coord(1'b1),
                            .rocks_in(), .num_rocks(0), .ships(ships),
                               ↪ .placeable(hit_player_ship));
                            //check if CPU hits ship player
                            //set rocks to none so only checks if we hit ship

    logic cpu_ship_out [4:0];
    ship_coords cpu_ship0_coord_checker(.x(x), .y(y), .ships(cpu_ships),
       ↪ .ship_num(4'd0), .ship_out(cpu_ship_out[0]));
    ship_coords cpu_ship1_coord_checker(.x(x), .y(y), .ships(cpu_ships),
       ↪ .ship_num(4'd1), .ship_out(cpu_ship_out[1]));
    ship_coords cpu_ship2_coord_checker(.x(x), .y(y), .ships(cpu_ships),
       ↪ .ship_num(4'd2), .ship_out(cpu_ship_out[2]));
    ship_coords cpu_ship3_coord_checker(.x(x), .y(y), .ships(cpu_ships),
       ↪ .ship_num(4'd3), .ship_out(cpu_ship_out[3]));
    ship_coords cpu_ship4_coord_checker(.x(x), .y(y), .ships(cpu_ships),
       ↪ .ship_num(4'd4), .ship_out(cpu_ship_out[4]));

```

```
//generate random rocks for CPU board
//generate random positions for CPU ships
//generate random coordinate for CPU attack
always_ff @(posedge clk) begin
    if(reset) begin
        state <= START_MENU; //if reset, then set state to start menu
        ship_state <= 0;
        game_size <= 12;
        old_btnc <= 1;
        num_opponent_squares_hit <= 0;
        num_player_squares_hit <= 0;
        num_rocks <= 0;
        start_rock_gen <= 0;
        ships <= '{'{0, 0, 4'd2, 0, 0}, //(x, y, length, orientation,
           ↪ num_hits)
                   '{13, 1, 4'd3, 0, 0},
                   '{13, 5, 4'd3, 0, 0},
                   '{14, 0, 4'd4, 0, 0},
                   '{14, 5, 4'd5, 0, 0}};
        rx_ready <= 1'b0;
        tx_data <= 16'b0;
        tx_valid <= 1'b0;
        for(int i = 0; i < 12; i++) begin
            for(int j = 0; j < 12; j++) begin
                player_board[i][j] <= 3'b010;  //show all water on player
                   ↪ board
                opponent_board[i][j] <= 3'b000;
            end
        end
        cpu_ships <= '{'{0, 0, 4'd2, 0, 0},
                     '{0, 0, 4'd3, 0, 0},
                     '{0, 0, 4'd3, 0, 0},
                     '{0, 0, 4'd4, 0, 0},
                     '{0, 0, 4'd5, 0, 0}};
        cpu_ship_state <= 0;
        stall_ctr <= 0;
        start_rng <= 0;
        cpu_sunk_ship <= 0;
        sunk_ship_coords <= 0;
        sunk_ship_length <= 0;

    end
    else begin
        old_btnc <= btnc;
        case(state)
            START_MENU: begin
                start_rng <= 1;
                if(~old_btnc & btnc) begin
                    play_bgm <= 1'b1;
                    case(y)
                        0: //first menu option: become game host; set up menu
                           ↪ options
                            state <= SET_GAME_PARAMETERS;
                        1: //second menu option: look for open game hosted by
                           ↪ other FPGA to join
                            begin
                            state <= CONNECTION_SENDING;
                            tx_valid <= 1'b1;
                            tx_data <= 16'hFFFF;
                            end
                        2: //third menu option: go to singleplayer
                            state <= SINGLEPLAYER_START;
                    endcase
                end
            end
            SET_GAME_PARAMETERS: begin
                play_bgm <= 1'b0;
                game_size <= game_size_switch + 4'd9;
                num_rocks <= num_rocks_in;
                start_rock_gen <= 1;
                state <= CONNECTION_WAITING;
            end
            SINGLEPLAYER_START: begin
                play_bgm <= 1'b0;
                game_size <= game_size_switch + 4'd9;
                hard_mode <= hard_mode_in;
                num_rocks <= num_rocks_in;
                start_rock_gen <= 1;
                state <= SINGLEPLAYER_RANDOM_STALL;
                stall_ctr <= 0;
            end
            SINGLEPLAYER_RANDOM_STALL: begin
                if(done_rock_gen) begin
                    for(int i = 0; i < num_rocks; i++) begin //set rocks
                        player_board[player_rocks[i][3:0]][player_rocks[i][7:4]]
                           ↪ <= 3'b011;
                    end //set random CPU ship orientation
                    for(int i = 0; i < 5; i++) begin
                        cpu_ships[i].orientation <= unbounded_out[i];
                    end
                    state <= SINGLEPLAYER_PLACE_CPU_SHIPS;
                end
            end
            SINGLEPLAYER_PLACE_CPU_SHIPS: begin
                stall_ctr <= stall_ctr + 1;
                if(stall_ctr > 500) begin
                    if(rng_out[0][3:0] +
                       ↪ (cpu_ships[cpu_ship_state].orientation == 0 ?
                                         cpu_ships[cpu_ship_state].length - 1
                                            ↪ : 0) < game_size
                    && rng_out[1][3:0] +
                       ↪ (cpu_ships[cpu_ship_state].orientation == 1 ?
                                         cpu_ships[cpu_ship_state].length - 1
                                            ↪ : 0) < game_size
                    && cpu_placeable) begin //if ship placeable, place ship
                        cpu_ships[cpu_ship_state].x <= rng_out[0][3:0];
                        cpu_ships[cpu_ship_state].y <= rng_out[1][3:0];
                        cpu_ship_state <= cpu_ship_state + 1;
                    end
                    stall_ctr <= 0;
                end
                if(cpu_ship_state == 5)
                    state <= SINGLEPLAYER_PLACE_SHIPS;
            end
            SINGLEPLAYER_PLACE_SHIPS: begin
                ships[ship_state].x <= x;
                ships[ship_state].y <= y;
                ships[ship_state].orientation <= rotate;
                if((btnc & ~old_btnc) && placeable_out) begin //place ship on center button, increment ship count if ship is in fact placeable
                    ship_state <= ship_state + 1;
                end else if(ship_state == 5) begin
                    state <= SINGLEPLAYER_PLAY_MOVE_STALL;
                end
            end
            SINGLEPLAYER_PLAY_MOVE_STALL : state <= SINGLEPLAYER_PLAY_MOVE;
            SINGLEPLAYER_PLAY_MOVE: begin
                if(btnc & ~old_btnc)
                    if (opponent_board[x][y] == 3'b000) begin
                        opponent_board[x][y] <= 3'b010; //set to miss
                        cpu_sunk_ship <= 3'd7;
                        play_miss <= 1;

                        //check for hits
                        for(int i = 0; i < 5; i++) begin
                            if(cpu_ship_out[i]) begin
                                play_miss <= 0;
                                play_explosion <= 1;
                                cpu_ships[i].num_hits <= cpu_ships[i].num_hits + 1;
                                num_opponent_squares_hit <= num_opponent_squares_hit + 1;
                                if(cpu_ships[i].num_hits + 1 == cpu_ships[i].length) begin //ship sunk
                                    cpu_sunk_ship <= i;
                                    sunk_ship_coords <= {cpu_ships[i].x, cpu_ships[i].y};
                                    sunk_ship_length <= cpu_ships[i].length;
                                end
                                else //only hit but no sink
                                    opponent_board[x][y] <= 3'b001;
                            end
                        //check if rock position
                        if(!cpu_rocks_out) begin
                            opponent_board[x][y] <= 3'b011;
                            play_miss <= 1;
                        end
                        state <= SINGLEPLAYER_UPDATE_SUNK_SHIP;
                    end
                end
            end
            SINGLEPLAYER_UPDATE_SUNK_SHIP : begin
                play_miss <= 0;
                play_explosion <= 0;
                if(cpu_sunk_ship != 3'd7) begin
                    for(int i = 0; i < sunk_ship_length; i++) begin
                        if(cpu_ships[cpu_sunk_ship].orientation == 0) //horizontal ship
                            opponent_board[sunk_ship_coords[7:4] + i][sunk_ship_coords[3:0]] <= 3'b100;
                        else
                            opponent_board[sunk_ship_coords[7:4]][sunk_ship_coords[3:0] + i] <= 3'b100;
                    end
                    cpu_sunk_ship <= 3'd7;
                end
                if(num_opponent_squares_hit == 17)
                    state <= END_GAME_WIN;
                else state <= SINGLEPLAYER_OPPONENT_PLAY_MOVE;
                stall_ctr <= 0;
                num_cpu_hits <= 0;
            end
            SINGLEPLAYER_OPPONENT_PLAY_MOVE: begin
                stall_ctr <= stall_ctr + 1;
                if(stall_ctr > 20000000) begin //add artificial delay
                    if(player_board[rng_out[0][3:0]][rng_out[1][3:0]] != 3'b001) begin
                        player_board[rng_out[0][3:0]][rng_out[1][3:0]] <= 3'b001;
                        stall_ctr <= 0;
                        num_cpu_hits <= num_cpu_hits + 1;
                        if(!hit_player_ship) begin
                            num_player_squares_hit <= num_player_squares_hit + 1;
                            if(num_player_squares_hit == 16)
                                state <= END_GAME_LOSE;
                            else if(num_cpu_hits == 2 || !hard_mode)
                                state <= SINGLEPLAYER_PLAY_MOVE;
                        end
                        else if(num_cpu_hits == 2 || !hard_mode)
                            state <= SINGLEPLAYER_PLAY_MOVE;
                    end
                end
            end
            CONNECTION_WAITING: begin
                if(btnc & ~old_btnc)
                    state <= START_MENU;
                else if (rx_valid && rx_data == 16'hFFFF && done_rock_gen) begin
                    rx_ready <= 1'b1;
                    tx_data <= {game_size, num_rocks, 9'b0};
                    tx_valid <= 1'b1;
                    for(int i = 0; i < num_rocks; i++)
                        player_board[player_rocks[i][3:0]][player_rocks[i][7:4]] <= 3'b011;
                    //wait for UART START signal 16'hFFFF
                    //send game params {game_size[3:0], rock_num[2:0], 9'b0}
                    state <= SETUP_BOARD;
                end

            end //TODO: Integration with RX/TX
            CONNECTION_SENDING: begin
                play_bgm <= 1'b0;
                tx_valid <= 1'b0;
                if(btnc & ~old_btnc)
                    state <= START_MENU;
                else if (rx_valid) begin
                    rx_ready <= 1'b1;
                    game_size <= rx_data[15:12];
                    num_rocks <= rx_data[11:9];
                    start_rock_gen <= 1;
                    state <= GEN_ROCKS_RECEIVER;
                end
                //send UART START signal 16'FFFF
                //wait for game params signal
                //set game params equal to those in signal
            end
            GEN_ROCKS_RECEIVER: begin
                if(done_rock_gen) begin
                    for(int i = 0; i < num_rocks; i++)
                        player_board[player_rocks[i][3:0]][player_rocks[i][7:4]]
                          ↪ <= 3'b011;
                    state <= SETUP_BOARD;
                end
            end
            SETUP_BOARD: begin
                ships[ship_state].x <= x;
                ships[ship_state].y <= y;
                ships[ship_state].orientation <= rotate;

                tx_valid <= 1'b0;
                rx_ready <= 1'b0;

                if((btnc & ~old_btnc) && placeable_out) begin //place ship on
                    ↪ center button, increment ship count if ship is in fact
                    ↪ placeable
                    ship_state <= ship_state + 1;
                end else if(ship_state == 5) begin
                    tx_valid <= 1'b1;
                    tx_data <= 16'hFF00;
                    if (rx_data == 16'hFF00 && rx_valid) begin
                        rx_ready <= 1'b1;
                        state <= WAIT_MOVE_WAITING;
                    end else begin
                        state <= SETUP_WAITING;
                    end
                end
            end
            SETUP_WAITING: begin
                //wait for DONE signal. Once received, move to PLAY_MOVE
                tx_valid <= 1'b0;
                rx_ready <= 1'b0;
                if (rx_data == 16'hFF00 && rx_valid) begin
                    rx_ready <= 1'b1;
                    state <= PLAY_STALL;
                end
            end
            PLAY_STALL: begin
                tx_valid <= 0;
                rx_ready <= 1'b0;
                if(num_player_squares_hit == 17) //all ships hit
                    state <= END_GAME_LOSE;
                else state <= PLAY_STALL_2;
            end
            PLAY_STALL_2: begin
                state <= PLAY_MOVE;
            end
            PLAY_MOVE: begin
                rx_ready <= 0;
                tx_valid <= 0;
                if(~old_btnc & btnc) begin
                    if (opponent_board[x][y] == 3'b000) begin
                        tx_data <= {8'b0, x, y};
                        tx_valid <= 1'b1;
                    end
                    //check map if possible to fire
                    //fire at position (x,y)
                    //send (x,y) on TX
                    //interpret results (specified in MOVE_WAITING state)
                    //play audio
                    //update ship_positions on graphics
                    //check if the number of squares actually hit is == (2 +
                        ↪ 3 + 3 + 4 + 5), if so, then go to END_GAME_WIN state
                    //otherwise, go to MOVE_WAITING state
                end
                else if(rx_valid) begin
                    rx_ready <= 1;
                    if(rx_data == 16'b0) begin                //miss
                        opponent_board[x][y] <= 3'b010;
                        play_miss <= 1;
                    end else if(rx_data == 16'b1000_0000_0000_0000) begin
                        ↪ //hit
                        opponent_board[x][y] <= 3'b001;
                        num_opponent_squares_hit <= num_opponent_squares_hit
                            ↪ + 1;
                        play_explosion <= 1;
                    end
                    else if(rx_data == 16'b0100_0000_0000_0000) begin //rock
                        opponent_board[x][y] <= 3'b011;
                        play_miss <= 1;
                    end else if(rx_data[15:14] == 2'b11) begin
                        num_opponent_squares_hit <= num_opponent_squares_hit
                            ↪ + 1;
                        play_explosion <= 1;
                        for(int i = 0; i < ships[rx_data[12:10]].length; i++)
                            ↪ begin
                            if(rx_data[13] == 0) //horizontal ship
                                opponent_board[rx_data[7:4] +
                                    ↪ i][rx_data[3:0]] <= 3'b100;
                            else
                                opponent_board[rx_data[7:4]][rx_data[3:0] +
                                    ↪ i] <= 3'b100;
                        end
                    end
                    state <= WAIT_STALL;
                end
            end
            WAIT_MOVE_WAITING: begin
                tx_valid <= 0;
                rx_ready <= 1'b0;
                state <= WAIT_STALL;
            end
            WAIT_STALL: begin
                tx_valid <= 0;
                rx_ready <= 1'b0;
                play_miss <= 0;
                play_explosion <= 0;
                if(num_opponent_squares_hit == 17) //all ships hit
                    state <= END_GAME_WIN;
                else state <= WAIT_STALL_2;
            end
            WAIT_STALL_2: begin
                state <= MOVE_WAITING;
                end
                MOVE_WAITING: begin
                    //wait for (x,y) position of hit
                    //send on RX/TX:
                    //If missed: send 00000000
                    //If hit a ship only: send 10000000
                    //If hit and sunk a ship: send 11[rotate][# of ship sunk (3 bits)]00, then (x,y) coordinate of sunk ship
                    //check if the number of squares hit is == (2 + 3 + 3 + 4 + 5), if so, then go to END_GAME_LOSE state
                    //update hit_positions (and graphic display)
                    //otherwise, update screen, move to PLAY_MOVE screen
                    rx_ready <= 0;
                    tx_valid <= 0;
                    if(rx_valid) begin
                        tx_valid <= 1'b1;
                        rx_ready <= 1'b1;
                        tx_data <= 16'b0; //default: send MISS
                        //update player board to show HIT
                        player_board[rx_data[7:4]][rx_data[3:0]] <= 3'b001;

                        //check for hits
                        for(int i = 0; i < 5; i++) begin
                            if(ship_out[i]) begin
                                ships[i].num_hits <= ships[i].num_hits + 1;
                                num_player_squares_hit <= num_player_squares_hit + 1;
                                if(ships[i].num_hits + 1 == ships[i].length) begin //ship sunk
                                    tx_data <= {2'b11, ships[i].orientation, i[2:0], 2'b00, ships[i].x, ships[i].y};
                                end
                                else //only hit but no sink
                                    tx_data <= 16'b1000_0000_0000_0000;
                            end
                        end
                        if(!rocks_out) begin
                            tx_data <= 16'b0100_0000_0000_0000;
                        end
                        state <= PLAY_STALL;
                    end
                end
                END_GAME_WIN: if(~old_btnc & btnc) state <= RESET_GAME; //exit from END_GAME state to start menu with btnc press, show a green screen over board
                END_GAME_LOSE: if(~old_btnc & btnc) state <= RESET_GAME; //exit from END_GAME state to start menu with btnc press, show a red scren over board
            endcase
        end
    end
endmodule

`default_nettype wire
```

7.1.18 graphics_prep.sv

```
`timescale 1ns / 1ps
/////////////////////////////////////////////////////////////////////////////////
`default_nettype none
import fsm_state::*;

parameter CLEAR_PIXEL = 12'h0F0;

module graphics_prep( input wire clk_65mhz, rst_in,
                      input game_state state_in,
                      input wire [2:0] ship_state,
                      input ship_array ships,
                      input wire [3:0] x_in, y_in,
                      input wire [3:0] game_size,
                      input wire [2:0] player_board [11:0][11:0],
                      input wire [2:0] opponent_board [11:0][11:0],
                      input wire [11:0] mousex,
                      input wire [11:0] mousey,
                      input wire mouse_enable,
                      output logic [3:0] vga_r,
                      output logic [3:0] vga_g,
                      output logic [3:0] vga_b,
                      output logic vga_hs,
                      output logic vga_vs,
                      output logic [7:0] debug
    );
    localparam [5:0] sq_sz = 40;
    localparam spacing = 62;

    logic [10:0] hcount;
    logic [9:0] vcount;
    logic vsync, hsync, blank;
    logic [5:0] vsync_q;
    logic [5:0] hsync_q;
    logic [5:0] blank_q;

    logic [11:0] pixel_out;
    logic [11:0] board_pixel;
    logic [11:0] board_pixel_q [4:0];
    logic [11:0] water_pixel;
    logic [11:0] b2_pixel;
    logic [11:0] b3a_pixel;
    logic [11:0] b3b_pixel;
    logic [11:0] b4_pixel;
    logic [11:0] b5_pixel;
    logic [11:0] cursor_pixel;
    logic [11:0] cursor_pixel_q [3:0];
    logic [11:0] coverage_pixel_q [2:0];
    logic [11:0] coverage_pixel;
    logic [11:0] menu_pixel_q [2:0];
    logic [11:0] menu_pixel;

    logic [10:0] x_offset;
    logic [9:0] y_offset;

    assign x_offset = (game_size ==  4'd9) ? 10'd123 :
                      (game_size == 4'd10) ? 10'd082 :
                      (game_size == 4'd11) ? 10'd041 :
```

```
                (game_size == 4'd12) ? 10'd000 :
                10'b0;

assign y_offset = (game_size ==  4'd12) ? 9'd137 :
                (game_size == 4'd11) ? 9'd158 :
                (game_size == 4'd10) ? 9'd179 :
                (game_size == 4'd9) ? 9'd199 :
                10'b0;

xvga xvga(.vclock_in(clk_65mhz), .hcount_out(hcount), .vcount_out(vcount),
            .vsync_out(vsync), .hsync_out(hsync), .blank_out(blank));

base_board #(.SQ_SIZE(sq_sz)) bb (.clk_in(clk_65mhz), .rst_in(rst_in),
   ↪ .hcount_in(hcount), .vcount_in(vcount),
                                    .pixel_out(board_pixel),
                                       ↪ .size(game_size),
                                       ↪ .x_offset(x_offset),
                                       ↪ .y_offset(y_offset));

boat2 b2 (.clk_in(clk_65mhz), .x_in(ships[0].x*sq_sz + x_offset),
   ↪ .y_in(ships[0].y*sq_sz + y_offset),
            .vcount_in(vcount), .hcount_in(hcount),
               ↪ .flipped(ships[0].orientation), .pixel_out(b2_pixel));

boat3a b3a (.clk_in(clk_65mhz), .x_in(ships[1].x*sq_sz + x_offset),
   ↪ .y_in(ships[1].y*sq_sz + y_offset),
            .vcount_in(vcount), .hcount_in(hcount),
               ↪ .flipped(ships[1].orientation), .pixel_out(b3a_pixel));

boat3b b3b (.clk_in(clk_65mhz), .x_in(ships[2].x*sq_sz + x_offset),
   ↪ .y_in(ships[2].y*sq_sz + y_offset),
            .vcount_in(vcount), .hcount_in(hcount),
               ↪ .flipped(ships[2].orientation), .pixel_out(b3b_pixel));

boat4 b4 (.clk_in(clk_65mhz), .x_in(ships[3].x*sq_sz + x_offset),
   ↪ .y_in(ships[3].y*sq_sz + y_offset),
            .vcount_in(vcount), .hcount_in(hcount),
               ↪ .flipped(ships[3].orientation), .pixel_out(b4_pixel));

boat5 b5 (.clk_in(clk_65mhz), .x_in(ships[4].x*sq_sz + x_offset),
   ↪ .y_in(ships[4].y*sq_sz + y_offset),
            .vcount_in(vcount), .hcount_in(hcount),
               ↪ .flipped(ships[4].orientation), .pixel_out(b5_pixel));

select_square #(.SQ_SIZE(sq_sz)) cursor (.x_in(x_in*sq_sz + 512 + spacing/2),
   ↪ .y_in(y_in*sq_sz + y_offset), .hcount_in(hcount),
                                            .vcount_in(vcount),
                                               ↪ .pixel_out(cursor_pixel));

coverage #(.SQ_SIZE(sq_sz)) coverages (.clk_in(clk_65mhz), .rst_in(rst_in),
   ↪ .hcount_in(hcount), .vcount_in(vcount), .game_size(game_size),
                                       .player_board(player_board),
                                          ↪ .opponent_board(opponent_board),
                                          ↪ .pixel_out(coverage_pixel),
                                       .x_offset(x_offset),
                                          ↪ .y_offset(y_offset));

start_menu menu (.clk_in(clk_65mhz), .rst_in(rst_in), .hcount_in(hcount),
   ↪ .vcount_in(vcount), .selected(y_in), .pixel_out(menu_pixel));
```

```

    logic mouse_cursor_pixel;
    assign mouse_cursor_pixel = (hcount > mousex ? hcount - mousex : mousex -
       ↪ hcount) <= 4
                             && (vcount > mousey ? vcount - mousey : mousey -
                                ↪ vcount) <= 4;

    always_ff @(posedge clk_65mhz) begin
        if (rst_in) begin
            pixel_out <= 12'hFFF;
        end else begin
            pixel_out <=          (mouse_cursor_pixel & mouse_enable) ? 12'h0A0 :
                          (menu_pixel_q[0] && state_in == START_MENU) ?
                             ↪ menu_pixel_q[0] :
           (cursor_pixel_q[0] && (state_in == SINGLEPLAYER_PLAY_MOVE
                                           || state_in == PLAY_MOVE)) ?
                                              ↪ cursor_pixel_q[0]:
                                                (coverage_pixel_q[0]) ?
                                                   ↪ coverage_pixel_q[0]:
                                        (ship_state >= 4 && b5_pixel) ? b5_pixel :
                                        (ship_state >= 3 && b4_pixel) ? b4_pixel :
                                       (ship_state >= 2 && b3b_pixel) ? b3b_pixel
                                          ↪ :
                                       (ship_state >= 1 && b3a_pixel) ? b3a_pixel
                                          ↪ :
                                        (ship_state >= 0 && b2_pixel) ? b2_pixel :
                                    (board_pixel_q[0] != CLEAR_PIXEL) ?
                                       ↪ board_pixel_q[0] :
                                           (state_in == END_GAME_WIN) ? 12'h0F0 :
                                          (state_in == END_GAME_LOSE) ? 12'hF00 :
                                                                        12'hFFF;

            vsync_q <= {vsync, vsync_q[5:1]};
            hsync_q <= {hsync, hsync_q[5:1]};
            blank_q <= {blank, blank_q[5:1]};
            cursor_pixel_q <= {cursor_pixel, cursor_pixel_q[3:1]};
            coverage_pixel_q <= {coverage_pixel, coverage_pixel_q[2:1]};
            menu_pixel_q <= {menu_pixel, menu_pixel_q[2:1]};
            board_pixel_q <= {board_pixel, board_pixel_q[4:1]};
        end
    end

    // the following lines are required for the Nexys4 VGA circuit - do not change
    assign vga_r = ~blank_q[0] ? pixel_out[11:8]: 0;
    assign vga_g = ~blank_q[0] ? pixel_out[7:4] : 0;
    assign vga_b = ~blank_q[0] ? pixel_out[3:0] : 0;

    assign vga_hs = ~hsync_q[0];
    assign vga_vs = ~vsync_q[0];

endmodule
```

7.1.19 hit_or_miss.sv

```
`default_nettype none

module hit_or_miss #(parameter SQ_SIZE = 40)
                     (input wire clk_in, rst_in,
                      input wire [10:0] hcount_in, x_in,
                      input wire [9:0] vcount_in, y_in,
                      input wire [2:0] hit,
                      output logic [11:0] pixel_out);

    logic [11:0] e_pixel;
    logic [11:0] r_pixel;

    logic [10:0] x_in_q;
    logic [9:0] y_in_q;

    explosion ex1 (.clk_in(clk_in), .rst_in(rst_in), .hcount_in(hcount_in),
       ↪ .vcount_in(vcount_in), .x_in(x_in_q), .y_in(y_in_q),
       ↪ .pixel_out(e_pixel));
    rock rock1 (.clk_in(clk_in), .rst_in(rst_in), .hcount_in(hcount_in),
       ↪ .vcount_in(vcount_in), .x_in(x_in_q), .y_in(y_in_q),
       ↪ .pixel_out(r_pixel));

    always_ff @(posedge clk_in) begin
      x_in_q <= x_in;
      y_in_q <= y_in;
      if ((hcount_in >= x_in && hcount_in < (x_in+SQ_SIZE)) &&
            (vcount_in >= y_in && vcount_in < (y_in+SQ_SIZE)))
            pixel_out <= (hit == 3'b000) ? 12'hccc : //unguessed
                         (hit == 3'b001) ? e_pixel : //hit
                         (hit == 3'b010) ? 12'h000 : //uncovered, miss
                         (hit == 3'b011) ? r_pixel : //rock
                         (hit == 3'b100) ? (e_pixel >> 1) + (12'h000 >> 1) :
                            ↪ //sunk
                         12'h000;
      else pixel_out <= 0;
   end

endmodule

`default_nettype wire
```

7.1.20 hv_to_xy.sv

```
module hv_to_xy #(parameter SQ_SIZE = 40)
                  (input wire [10:0] hcount_in, start_hcount,
                   input wire [9:0] vcount_in, start_vcount,
                   output logic [3:0] x_out,
                   output logic [3:0] y_out);

    logic [10:0] hcount;
    logic [9:0] vcount;

    assign hcount = hcount_in >= start_hcount ? hcount_in - start_hcount :
        ↪ 11'hFFF;
    assign vcount = vcount_in >= start_vcount ? vcount_in - start_vcount: 10'hFFF;

    always_comb begin
      if(hcount > SQ_SIZE*0 && hcount <= SQ_SIZE*1) begin
        x_out = 0;
      end else if(hcount > SQ_SIZE*1 && hcount <= SQ_SIZE*2) begin
        x_out = 1;
      end else if(hcount > SQ_SIZE*2 && hcount <= SQ_SIZE*3) begin
        x_out = 2;
      end else if(hcount > SQ_SIZE*3 && hcount <= SQ_SIZE*4) begin
        x_out = 3;
      end else if(hcount > SQ_SIZE*4 && hcount <= SQ_SIZE*5) begin
        x_out = 4;
      end else if(hcount > SQ_SIZE*5 && hcount <= SQ_SIZE*6) begin
        x_out = 5;
      end else if(hcount > SQ_SIZE*6 && hcount <= SQ_SIZE*7) begin
        x_out = 6;
      end else if(hcount > SQ_SIZE*7 && hcount <= SQ_SIZE*8) begin
        x_out = 7;
      end else if(hcount > SQ_SIZE*8 && hcount <= SQ_SIZE*9) begin
        x_out = 8;
      end else if(hcount > SQ_SIZE*9 && hcount <= SQ_SIZE*10) begin
        x_out = 9;
      end else if(hcount > SQ_SIZE*10 && hcount <= SQ_SIZE*11) begin
        x_out = 10;
      end else if(hcount > SQ_SIZE*11 && hcount <= SQ_SIZE*12) begin
        x_out = 11;
      end else x_out = 4'hF;

      if(vcount > SQ_SIZE*0 && vcount <= SQ_SIZE*1) begin
        y_out = 0;
      end else if(vcount > SQ_SIZE*1 && vcount <= SQ_SIZE*2) begin
        y_out = 1;
      end else if(vcount > SQ_SIZE*2 && vcount <= SQ_SIZE*3) begin
        y_out = 2;
      end else if(vcount > SQ_SIZE*3 && vcount <= SQ_SIZE*4) begin
        y_out = 3;
      end else if(vcount > SQ_SIZE*4 && vcount <= SQ_SIZE*5) begin
        y_out = 4;
      end else if(vcount > SQ_SIZE*5 && vcount <= SQ_SIZE*6) begin
        y_out = 5;
      end else if(vcount > SQ_SIZE*6 && vcount <= SQ_SIZE*7) begin
        y_out = 6;
      end else if(vcount > SQ_SIZE*7 && vcount <= SQ_SIZE*8) begin
        y_out = 7;
      end else if(vcount > SQ_SIZE*8 && vcount <= SQ_SIZE*9) begin
```

```
        y_out = 8;
      end else if(vcount > SQ_SIZE*9 && vcount <= SQ_SIZE*10) begin
        y_out = 9;
      end else if(vcount > SQ_SIZE*10 && vcount <= SQ_SIZE*11) begin
        y_out = 10;
      end else if(vcount > SQ_SIZE*11 && vcount <= SQ_SIZE*12) begin
        y_out = 11;
      end else y_out = 4'hF;
    end
endmodule
```

7.1.21 picture_blob.sv

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
`default_nettype none

module picture_blob
   #(parameter WIDTH = 10,
               HEIGHT = 10,
               GREYSCALE = 1,
               CLEAR_PIXEL = 12'h000)
   (input wire pixel_clk_in,
    input wire [10:0] x_in, hcount_in,
    input wire [9:0] y_in, vcount_in,
    input wire [7:0] red_mapped, blue_mapped, green_mapped,
    input wire flip,

    output logic [$clog2(WIDTH*HEIGHT)-1:0] image_addr,
    output logic [11:0] pixel_out);

   // calculate rom address and read the location
   assign image_addr = !flip ? (hcount_in-x_in) + (vcount_in-y_in) * WIDTH :
                               WIDTH-1-(vcount_in-y_in) + (hcount_in-x_in) *
                                  ↪ WIDTH;

   always_ff @ (posedge pixel_clk_in) begin
     if (flip) begin

        if ((hcount_in >= x_in && hcount_in < (x_in+HEIGHT)) &&
              (vcount_in >= y_in && vcount_in < (y_in+WIDTH)))
             // use MSB 4 bits
             pixel_out <= GREYSCALE ? {red_mapped[7:4], red_mapped[7:4],
                 ↪ red_mapped[7:4]} :
                                  {red_mapped[7:4], green_mapped[7:4],
                                      ↪ blue_mapped[7:4]};
        else pixel_out <= CLEAR_PIXEL;

     end else begin

         if ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
              (vcount_in >= y_in && vcount_in < (y_in+HEIGHT)))
             // use MSB 4 bits
             pixel_out <= GREYSCALE ? {red_mapped[7:4], red_mapped[7:4],
                 ↪ red_mapped[7:4]} :
                                  {red_mapped[7:4], green_mapped[7:4],
                                      ↪ blue_mapped[7:4]};
         else pixel_out <= CLEAR_PIXEL;

      end
   end
endmodule

`default_nettype wire
```

7.1.22 placeable.sv

```
`default_nettype none
import fsm_state::*;

module placeable(input wire[3:0] x, y,
                 input wire[2:0] ship_num,
                 input wire[7:0] rocks_in [7:0],
                 input wire[2:0] num_rocks,
                 input wire check_one_coord,
                 input ship_array ships,
                 output logic placeable);

    logic[3:0] xvar;
    logic[3:0] yvar;
    always_comb begin
        placeable = 1;
        for(int j = 0; j < ships[ship_num].length; j++) begin
            xvar = check_one_coord ? x : ships[ship_num].orientation ? x : x + j;
            yvar = check_one_coord ? y : ships[ship_num].orientation ? y + j : y;
            for(int i = 0; i < ship_num; i++) begin
                if(ships[i].orientation == 0) begin // horizontal ship
                    if(ships[i].x <= xvar && xvar < ships[i].x + ships[i].length
                        ↪ && yvar == ships[i].y)
                        placeable = 0;
                end
                else if(ships[i].y <= yvar && yvar < ships[i].y + ships[i].length
                    ↪ && xvar == ships[i].x)
                    placeable = 0;
            end
            for(int i = 0; i < num_rocks; i++) begin
                if(rocks_in[i][3:0] == xvar && rocks_in[i][7:4] == yvar)
                    placeable = 0;
            end
        end
    end

endmodule

`default_nettype wire
```

7.1.23 rng.sv

```
//xorshift RNG https://en.wikipedia.org/wiki/Xorshift
module rng(input wire rst, clk,
           input wire[31:0] initial_seed,
           input wire start,
           output logic[31:0] number_out);
    logic generating;
    always_ff @(posedge clk) begin
        if(rst) begin
            number_out <= 0;
            generating <= 0;
        end
        else if (generating) begin
            number_out <= (number_out ^ (number_out << 13))
                        ^ ((number_out ^ (number_out << 13)) >> 17)
                        ^ (((number_out ^ (number_out << 13)) >> 17) << 5);
        end
        else if (start) begin
            number_out <= initial_seed;
            generating <= 1;
        end
    end
endmodule


module bounded_rng #(NUM_NUMBERS = 32) (input wire reset, clk,
                      input wire start,
                      input wire[31:0] initial_seed,
                      input wire[3:0] num_max,
                      output logic[3:0] numbers_out [NUM_NUMBERS - 1:0],
                      output logic[31:0] unbounded_out); //generates NUM_NUMBERS
                         ↪ numbers from 0 to num_max - 1 with MCMC methods;

    logic[31:0] mcmc_output;
    logic generating;
    rng rng(.rst(reset), .clk(clk), .initial_seed(initial_seed), .start(start),
        ↪ .number_out(mcmc_output));
    always_ff @(posedge clk) begin
        if(reset) begin
            unbounded_out <= 0;
            generating <= 0;
            for(int i = 0; i < NUM_NUMBERS; i++)
                numbers_out[i] <= 4'b0;
        end
        else if (generating) begin
            for(int i = 0; i < NUM_NUMBERS; i++)
                if(mcmc_output[i])
                    numbers_out[i] <= numbers_out[i] == 0 ? 0 : numbers_out[i] -
                        ↪ 1;
                else
                    numbers_out[i] <= numbers_out[i] >= num_max - 1 ? num_max - 1
                        ↪ : numbers_out[i] + 1;
            unbounded_out <= mcmc_output;
        end
        else if (start) begin
            for(int i = 0; i < NUM_NUMBERS; i++) begin
                numbers_out[i] <= (num_max >> 1);
            end
            generating <= 1;
        end
    end
endmodule
```

7.1.24 rock.sv

```
module rock (input wire clk_in, rst_in,
             input wire [10:0] hcount_in, x_in,
             input wire [9:0] vcount_in, y_in,
             output wire [11:0] pixel_out);

    localparam w = 41;
    localparam h = 41;

    logic [7:0] image_bits, red_mapped, blue_mapped, green_mapped;
    logic [$clog2(w*h)-1:0] image_addr;

    picture_blob #(.WIDTH(w), .HEIGHT(h)) rock (.pixel_clk_in(clk_in),
        ↪ .hcount_in(hcount_in), .vcount_in(vcount_in), .x_in(x_in), .y_in(y_in),
                                             .flip(0), .red_mapped(red_mapped),
                                                ↪ .blue_mapped(blue_mapped),
                                                ↪ .green_mapped(green_mapped),
                                             .image_addr(image_addr),
                                                ↪ .pixel_out(pixel_out));

    rock_image rock_image (.clka(clk_in), .addra(image_addr), .douta(image_bits));
    rock_red rock_red (.clka(clk_in), .addra(image_bits), .douta(red_mapped));
    rock_blue rock_blue (.clka(clk_in), .addra(image_bits), .douta(blue_mapped));
    rock_green rock_green (.clka(clk_in), .addra(image_bits),
        ↪ .douta(green_mapped));

endmodule
```

7.1.25 rock_coords.sv

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 12/06/2021 01:42:40 AM
// Design Name:
// Module Name: rock_coords
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module rock_coords(input wire[3:0] x, y,
                 input wire[7:0] rocks_in [7:0],
                 input wire[2:0] num_rocks,
                 output logic placeable);
    always_comb begin
        placeable = 1;
        for(int i = 0; i < num_rocks; i++)
            if(rocks_in[i][3:0] == x && rocks_in[i][7:4] == y)
                placeable = 0;
    end
endmodule
```

7.1.26 rock_randomizer.sv

```
`default_nettype none

module rock_randomizer (input wire reset, clk,
                        input wire [3:0] rng_input [31:0],
                        input wire start,
                        output logic done,
                        output logic[7:0] player_rocks_out[7:0],
                        output logic[7:0] cpu_rocks_out[7:0]);
    int counter = 0;
    logic prev_start;
    logic generating;
    always_ff @(posedge clk) begin
        prev_start <= start;
        if(reset) begin
            done <= 0;
            counter <= 0;
            generating <= 0;
            for(int i = 0; i < 8; i++) begin
                player_rocks_out[i] <= 8'b0;
                cpu_rocks_out[i] <= 8'b0;
            end
        end
        else if((start & ~prev_start)) begin
            generating <= 1;
            counter <= 0;
            done <= 0;
        end
        else if(generating) begin
            counter <= counter + 1;
            if(counter > 500) begin
                for(int i = 0; i < 8; i++) begin
                    player_rocks_out[i] <= {rng_input[i], rng_input[2 * i]};
                    cpu_rocks_out[i] <= {rng_input[15 + i], rng_input[15 + 2 *
                        ↪ i]};
                end
                done <= 1;
                generating <= 0;
            end
        end
    end

endmodule
`default_nettype wire
```

7.1.27 rx.sv

```
`default_nettype none

module rx #(parameter BUFFER_SIZE = 16,
                      CLOCK = 65000000,
                      BAUD = 9600)
           (input wire clk_in, rst_in, rx_in, ready_in,
            output logic valid_out,
            output logic [BUFFER_SIZE-1:0] data_out
            );

    localparam DIVISOR = CLOCK/BAUD;
    localparam DATA_FRAME = 8;

    typedef enum {IDLE, RX_START, RX_DATA, RX_STOP, END_RX} T_state;

    logic [$clog2(DIVISOR)-1:0] baud_count;
    logic [$clog2(BUFFER_SIZE):0] data_ptr;
    logic [$clog2(DATA_FRAME):0] frame_count;
    logic [DATA_FRAME + 1: 0] frame;

    logic rx_dirty;
    logic rx_sync;
    logic old_rx_sync;

    T_state state;

    always_ff @(posedge clk_in) begin
        old_rx_sync <= rx_sync;
        rx_sync <= rx_dirty;
        rx_dirty <= rx_in;

        if(rst_in) begin
            baud_count <= 0;
            frame_count <= 0;
            frame <= 0;
            data_ptr <= 0;
            state <= IDLE;
            data_out <= 0;
            valid_out <= 1'b0;
        end else begin
            case(state)
                IDLE: begin
                    valid_out <= 1'b0;
                    if(!rx_sync && old_rx_sync) begin
                        state <= RX_START;
                        baud_count <= 0;
                    end
                end

                RX_START: begin
                    baud_count <= baud_count + 1'b1;

                    if(baud_count == DIVISOR/2) begin
                        state <= rx_sync ? IDLE : RX_DATA;
                        baud_count <= 0;
                    end
                end
```

```verilog

                RX_DATA: begin
                    if (baud_count == DIVISOR - 1) begin
                        baud_count <= 0;
                        frame[frame_count] <= rx_sync;
                        if(frame_count == DATA_FRAME) begin
                            data_out[data_ptr +: DATA_FRAME] <= frame[DATA_FRAME-1:0];
                            data_ptr <= data_ptr + 8;
                            frame_count <= 0;
                            state <= RX_STOP;
                        end else begin
                            frame_count <= frame_count + 1'b1;
                        end
                    end else baud_count <= baud_count + 1'b1;
                end

                RX_STOP: begin
                    if(data_ptr == BUFFER_SIZE) begin
                        state <= ready_in ? IDLE : RX_STOP;
                        valid_out <= 1'b1;
                        data_ptr <= ready_in ? 1'b0 : data_ptr;
                    end else begin
                        state <= IDLE;
                        valid_out <= 1'b0;
                    end
                end
            endcase
        end
    end
endmodule

`default_nettype wire
```

7.1.28 select_square.sv

```
`default_nettype none

module select_square #(parameter SQ_SIZE = 40)
                     (input wire [10:0] hcount_in, x_in,
                      input wire [9:0] vcount_in, y_in,
                      output logic [11:0] pixel_out);

    always_comb begin
      if  ((hcount_in > x_in + 5 && hcount_in + 5 < (x_in+SQ_SIZE)) &&
            (vcount_in > y_in + 5 && vcount_in + 5 < (y_in+SQ_SIZE))) begin
        if((hcount_in == x_in + SQ_SIZE/2) || vcount_in == y_in + SQ_SIZE/2) begin
            pixel_out = 12'hF00;
        end else pixel_out = 12'h000;
      end else pixel_out = 0;
   end

endmodule

`default_nettype wire
```

7.1.29 ship_coords.sv

```
`default_nettype none
import fsm_state::*;

module ship_coords(input wire[3:0] x, y,
                   input ship_array ships,
                   input wire[3:0] ship_num,
                   output logic ship_out);

    always_comb begin
        ship_out = 0;
        if(ships[ship_num].orientation == 0) begin // horizontal ship
            if(ships[ship_num].x <= x && x < ships[ship_num].x +
                ↪ ships[ship_num].length && y == ships[ship_num].y)
                ship_out = 1;
        end
        else if(ships[ship_num].y <= y && y < ships[ship_num].y +
            ↪ ships[ship_num].length && x == ships[ship_num].x)
            ship_out = 1;
    end
endmodule


`default_nettype wire
```

7.1.30 start_menu.sv

```
`default_nettype none

module start_menu (input wire clk_in, rst_in,
                   input wire [3:0] selected,
                   input wire [10:0] hcount_in,
                   input wire [9:0] vcount_in,
                   output logic [11:0] pixel_out);

    localparam HCOUNT_MAX = 1023;
    localparam VCOUNT_MAX = 767;

    logic [2:0] char_pixel;
    logic [63:0] cstring;
    logic [10:0] text_x;
    logic [9:0] text_y;

    logic [10:0] select_x;
    logic [9:0] select_y;

    logic [11:0] box1_pixel;
    logic [11:0] box2_pixel;
    logic [11:0] box3_pixel;
    logic [11:0] select_pixel;

    assign select_x = HCOUNT_MAX/4;
    assign select_y = (1'b1+selected) * VCOUNT_MAX/4 - 25;

    char_string_display text(.vclock(clk_in), .hcount(hcount_in),
       ↪ .vcount(vcount_in),
                             .cstring(cstring), .cx(text_x), .cy(text_y),
                                ↪ .pixel(char_pixel));

    blob #(.WIDTH(HCOUNT_MAX/2), .HEIGHT(50), .COLOR(12'h222)) box1
       ↪ (.x_in(HCOUNT_MAX/4), .y_in(VCOUNT_MAX/4-25),
                                                                     .vcount_in(vcount_in),
                                                                        ↪ .hcount_in(hcount_in
                                                                     .pixel_out(box1_pixel));
    blob #(.WIDTH(HCOUNT_MAX/2), .HEIGHT(50), .COLOR(12'h222)) box2
       ↪ (.x_in(HCOUNT_MAX/4), .y_in(VCOUNT_MAX/2-25),
                                                                     .vcount_in(vcount_in),
                                                                        ↪ .hcount_in(hcount_in
                                                                     .pixel_out(box2_pixel));
    blob #(.WIDTH(HCOUNT_MAX/2), .HEIGHT(50), .COLOR(12'h222)) box3
       ↪ (.x_in(HCOUNT_MAX/4), .y_in(3*VCOUNT_MAX/4-25),
                                                                     .vcount_in(vcount_in),
                                                                        ↪ .hcount_in(hcount_in
                                                                     .pixel_out(box3_pixel));
    blob #(.WIDTH(HCOUNT_MAX/2), .HEIGHT(50), .COLOR(12'hF00)) select_box
       ↪ (.x_in(select_x), .y_in(select_y),
                                                                           .vcount_in(vcount_in
                                                                              ↪ .hcount_in(hco
                                                                           .pixel_out(select_pix

    always_ff @(posedge clk_in) begin

        if(hcount_in == HCOUNT_MAX/2 - 32 && vcount_in == VCOUNT_MAX/4 - 16) begin
            cstring <= "HOST";
            text_x <= hcount_in;
            text_y <= vcount_in;
        end else if (hcount_in == HCOUNT_MAX/2 - 32 && vcount_in == VCOUNT_MAX/2
            ↪ - 16) begin
            cstring <= "JOIN";
            text_x <= hcount_in;
            text_y <= vcount_in;
        end else if (hcount_in == HCOUNT_MAX/2 - 32 && vcount_in ==
            ↪ 3*VCOUNT_MAX/4 - 16) begin
            cstring <= "SOLO";
            text_x <= hcount_in;
            text_y <= vcount_in;
        end

        pixel_out <= char_pixel[2] ? {12{char_pixel[2]}} :
                     select_pixel ? select_pixel :
                     box1_pixel ? box1_pixel :
                     box2_pixel ? box2_pixel :
                     box3_pixel ? box3_pixel :
                     12'h777;

    end

endmodule

`default_nettype wire
```

7.1.31 top_level.sv

```
`default_nettype none
import fsm_state::*;
//https://stackoverflow.com/questions/59041579/can-enum-be-made-an-output-in-systemverilog

module top_level (input wire clk_100mhz,
                  input wire[15:0] sw,
                  input wire btnc, btnu, btnd, btnr, btnl,
                  input wire [7:0] ja,
                  inout wire ps2_clk, ps2_data, // data to/from PS/2 mouse
                  output logic [7:0] jb,
                  output logic ca,cb,cc,cd,ce,cf,cg,
                  output logic [3:0] vga_r, vga_g, vga_b,
                  output logic vga_hs, vga_vs,
                  output logic [7:0] an,
                  output logic aud_pwm, aud_sd);

    //button/switch inputs:
    //sw[0] = rotate ship
    //sw[2:1] + 9 = game size
    //sw[3] = singleplayer hard mode enable
    //sw[6:4] = number of rocks
    //sw[9] = mouse enable/disable
    //sw[10] = sounds on/off
    //sw[11] = bgm on/off
    //sw[14:12] = volume control
    //sw[15] = reset
    //btnc = main button
    //btnd, btnl, btnr, btnu for movement

    logic clk_65mhz;
    clk_wiz_0 clk65 (.clk_in1(clk_100mhz), .clk_out1(clk_65mhz));

    logic mouse_click;
    logic mouse_rightclick;
    logic mouse_rightclick_clean;
    logic btnc_clean, btnd_clean, btnu_clean, btnr_clean, btnl_clean, mouse_clean;
    debouncer mouse_debouncer (.reset(sw[15]), .clk_in(clk_65mhz),
       ↪ .noisy_in(mouse_click), .clean_out(mouse_clean));
    debouncer mouse_rightclick_debouncer (.reset(sw[15]), .clk_in(clk_65mhz),
       ↪ .noisy_in(mouse_rightclick), .clean_out(mouse_rightclick_clean));

    logic  [11:0] mx, my;   // current mouse position, 12 bits
    MouseCtl mymouse(
    .clk        (clk_65mhz)   , // in std_logic;
    .rst        (sw[15])            , // in std_logic;
    .xpos       (mx)            , // out std_logic_vector(11 downto 0);
    .ypos       (my)            , // out std_logic_vector(11 downto 0);
    .zpos       ()          , // out std_logic_vector(3 downto 0);
    .left       (mouse_click)       , // out std_logic;
    .middle     ()        , // out std_logic;
    .right      (mouse_rightclick)        , // out std_logic;
    .new_event  ()     , // out std_logic;
    .value      (12'b0)         , // in std_logic_vector(11 downto 0);
    .setx       (1'b0)          , // in std_logic;
    .sety       (1'b0)          , // in std_logic;
    .setmax_x   (1'b0)          , // in std_logic;
    .setmax_y   (1'b0)          , // in std_logic;
    .ps2_clk    (ps2_clk)       , // inout std_logic;
    .ps2_data   (ps2_data)        // inout std_logic
);

    debouncer btnc_debouncer (.reset(sw[15]), .clk_in(clk_65mhz),
       ↪ .noisy_in(btnc), .clean_out(btnc_clean));
    debouncer btnd_debouncer (.reset(sw[15]), .clk_in(clk_65mhz),
       ↪ .noisy_in(btnd), .clean_out(btnd_clean));
    debouncer btnu_debouncer (.reset(sw[15]), .clk_in(clk_65mhz),
       ↪ .noisy_in(btnu), .clean_out(btnu_clean));
    debouncer btnr_debouncer (.reset(sw[15]), .clk_in(clk_65mhz),
       ↪ .noisy_in(btnr), .clean_out(btnr_clean));
    debouncer btnl_debouncer (.reset(sw[15]), .clk_in(clk_65mhz),
       ↪ .noisy_in(btnl), .clean_out(btnl_clean));

    logic [31:0] debug;
    game_state_controller gsc (.reset(sw[15]), .clk(clk_65mhz),
                              .btnc(btnc_clean | mouse_clean), .btnd(btnd_clean),
                                 ↪ .btnu(btnu_clean), .btnl(btnl_clean),
                                 ↪ .btnr(btnr_clean), //button inputs
                              .rotate_ship(sw[0] ^ (mouse_rightclick_clean & sw[9])),
                                 ↪ .game_size_switch({2'b00, sw[2:1]}), //ship
                                 ↪ placement / size options
                              .vga_r(vga_r), .vga_g(vga_g), .vga_b(vga_b),
                                 ↪ .vga_hs(vga_hs), .vga_vs(vga_vs),
                              .num_rocks(sw[6:4]),
                              .rx_in(ja[0]), .tx_out(jb[0]), .debug(debug),
                                 ↪ .mousex(mx), .mousey(my),
                              .volume_control(sw[14:12]), .bgm_on(sw[11]),
                                 ↪ .sounds_on(sw[10]),
                              .aud_pwm(aud_pwm), .aud_sd(aud_sd), .bgm_test(sw[5]),
                              .mouse_enable(sw[9]), .hard_mode(sw[3])); //board state

    logic [6:0] segments;
    assign {cg, cf, ce, cd, cc, cb, ca} = segments[6:0];
    display_8hex display(.clk_in(clk_65mhz), .data_in(debug), .seg_out(segments),
       ↪ .strobe_out(an)); //for debugging

endmodule

module display_8hex(
    input wire clk_in,                  // system clock
    input wire [31:0] data_in,          // 8 hex numbers, msb first
    output logic [6:0] seg_out,      // seven segment display output
    output logic [7:0] strobe_out    // digit strobe
    );

    localparam bits = 13;

    logic [bits:0] counter = 0;  // clear on power up

    logic [6:0] segments[15:0]; // 16 7 bit memorys
    assign segments[0]  = 7'b100_0000;  // inverted logic
    assign segments[1]  = 7'b111_1001;  // gfedcba
    assign segments[2]  = 7'b010_0100;
    assign segments[3]  = 7'b011_0000;
    assign segments[4]  = 7'b001_1001;
    assign segments[5]  = 7'b001_0010;
    assign segments[6]  = 7'b000_0010;
    assign segments[7]  = 7'b111_1000;
    assign segments[8]  = 7'b000_0000;
    assign segments[9]  = 7'b001_1000;
    assign segments[10] = 7'b000_1000;
    assign segments[11] = 7'b000_0011;
    assign segments[12] = 7'b010_0111;
    assign segments[13] = 7'b010_0001;
    assign segments[14] = 7'b000_0110;
    assign segments[15] = 7'b000_1110;

    always_ff @(posedge clk_in) begin
      // Here I am using a counter and select 3 bits which provides
      // a reasonable refresh rate starting the left most digit
      // and moving left.
      counter <= counter + 1;
      case (counter[bits:bits-2])
          3'b000: begin  // use the MSB 4 bits
                   seg_out <= segments[data_in[31:28]];
                   strobe_out <= 8'b0111_1111 ;
                  end

          3'b001: begin
                   seg_out <= segments[data_in[27:24]];
                   strobe_out <= 8'b1011_1111 ;
                  end

          3'b010: begin
                    seg_out <= segments[data_in[23:20]];
                    strobe_out <= 8'b1101_1111 ;
                   end
          3'b011: begin
                   seg_out <= segments[data_in[19:16]];
                   strobe_out <= 8'b1110_1111;
                  end
          3'b100: begin
                   seg_out <= segments[data_in[15:12]];
                   strobe_out <= 8'b1111_0111;
                  end

          3'b101: begin
                   seg_out <= segments[data_in[11:8]];
                   strobe_out <= 8'b1111_1011;
                  end

          3'b110: begin
                    seg_out <= segments[data_in[7:4]];
                    strobe_out <= 8'b1111_1101;
                   end
          3'b111: begin
                   seg_out <= segments[data_in[3:0]];
                   strobe_out <= 8'b1111_1110;
                  end

       endcase
      end
endmodule

`default_nettype wire
```

7.1.32 tx.sv

```
`default_nettype none

module tx
        #(parameter BUFFER_SIZE = 16,
           CLOCK = 65000000,
           BAUD = 9600)
          (input wire clk_in, rst_in, valid_in,
           input wire [BUFFER_SIZE-1:0] data_in,
           output logic tx_out, ready_out
    );

    localparam DIVISOR = CLOCK/BAUD;
    localparam DATA_FRAME = 8;

    typedef enum {IDLE, SEND_DATA, END_SEND} T_state;

    logic [BUFFER_SIZE-1:0] data;
    logic [$clog2(BUFFER_SIZE):0] data_ptr;
    logic [$clog2(DIVISOR)-1:0] baud_count;
    logic [9:0] frame;
    logic [3:0] frame_count;

    T_state state;

    always_ff @(posedge clk_in) begin
        if (rst_in) begin
            state <= IDLE;
            baud_count <= 0;
            frame <= 10'b11_1111_1111;
            frame_count <= 0;
            data_ptr <= 0;
            ready_out <= 1'b0;
            tx_out <= 1'b1;
        end else begin
            case (state)
                IDLE:
                    begin
                        if(valid_in) begin
                            state <= SEND_DATA;
                            frame_count <= 0;
                            frame <= {1'b1, data_in[data_ptr +: DATA_FRAME],
                                ↪ 1'b0};
                            data <= data_in;
                            ready_out <= 1'b0;
                            baud_count <= 0;
                        end
                    end

                SEND_DATA: begin
                    if (baud_count == DIVISOR - 1) begin
                        baud_count <= 0;
                        tx_out <= frame[0];
                        if (frame_count == DATA_FRAME + 1) begin
                            data_ptr <= data_ptr + DATA_FRAME;
                            if (data_ptr == BUFFER_SIZE - DATA_FRAME) begin
                                state <= END_SEND;
                            end else begin
```

```
                                frame <= {1'b1, data[data_ptr + DATA_FRAME +:
                                    ↪ DATA_FRAME], 1'b0};
                                frame_count <= 0;
                            end
                        end else begin
                            frame_count <= frame_count + 1'b1;
                            frame <= frame >> 1;
                        end
                    end
                    else baud_count <= baud_count + 1'b1;
                end

                END_SEND: begin
                    state <= IDLE;
                    ready_out <= 1'b1;
                    tx_out <= 1'b1;
                    data_ptr <= 0;
                    frame_count <= 0;
                    baud_count <= 0;
                end
            endcase
        end
    end

endmodule

`default_nettype wire
```

7.1.33 user_input.sv

```
`default_nettype none
import fsm_state::*;

module user_input(input wire reset, btnd, btnu, btnr, btnl, clk,
                  input game_state state,
                  input wire[3:0] size,
                  input wire[2:0] ship_num,
                  input wire rotate,
                  input wire mouse_enable,
                  input wire[11:0] mousex, mousey,
                  input ship_array ships,
                  output logic [3:0] x, y);

    localparam xsize = 1024;
    localparam ysize = 768;
    localparam spacing = 62;
    localparam [5:0] sq_sz = 40;
    logic old_btnd, old_btnu, old_btnr, old_btnl;
    logic [10:0] x_offset;
    logic [9:0] y_offset;

    assign x_offset = (size ==  4'd9) ? 10'd123 :
                      (size == 4'd10) ? 10'd082 :
                      (size == 4'd11) ? 10'd041 :
                      (size == 4'd12) ? 10'd000 :
                      10'b0;

    assign y_offset = (size == 4'd12) ? 9'd137 :
                      (size == 4'd11) ? 9'd158 :
                      (size == 4'd10) ? 9'd179 :
                      (size == 4'd9) ? 9'd199 :
                      10'b0;

    logic [11:0] px, py, ox, oy;
    hv_to_xy #(.SQ_SIZE(sq_sz)) player (.hcount_in(mousex),
        ↪ .start_hcount(x_offset), .vcount_in(mousey), .start_vcount(y_offset),
        ↪ .x_out(px), .y_out(py));
    hv_to_xy #(.SQ_SIZE(sq_sz)) opponent (.hcount_in(mousex),
        ↪ .start_hcount(512+spacing/2), .vcount_in(mousey),
        ↪ .start_vcount(y_offset), .x_out(ox), .y_out(oy));

    always_ff @(posedge clk) begin
        if(reset) begin
            x <= 0;
            y <= 0;
            old_btnd <= 1;
            old_btnu <= 1;
            old_btnr <= 1;
            old_btnl <= 1;
        end
        else begin
            old_btnd <= btnd;
            old_btnu <= btnu;
            old_btnr <= btnr;
            old_btnl <= btnl;

            if(state == START_MENU) begin
```

```verilog
                //only vertical menu scroll
                //here is menu with 3 options (x,y) = (0, 0) (0, 1) (0, 2)
                if(~old_btnd & btnd) y <= y == 2 ? y : y + 1;
                else if(~old_btnu & btnu) y <= y == 0 ? y : y - 1;
                else if(mouse_enable) begin
                    if(xsize/4 <= mousex && mousex <= xsize * 3/4) begin
                        if (mousey >= ysize/4 - 25 && mousey <= ysize/4 + 25) y <= 0;
                        else if (mousey >= ysize/2 - 25 && mousey <= ysize/2 + 25) y <= 1;
                        else if (mousey >= 3*ysize/4 - 25 && mousey <= 3*ysize/4 + 25) y <= 2;
                    end
                end
            end
            else if(state == CONNECTION_WAITING || state == CONNECTION_SENDING
                 || state == SETUP_WAITING || state == WAIT_MOVE_WAITING
                 || state == SINGLEPLAYER_PLACE_CPU_SHIPS
                 || state == SINGLEPLAYER_PLAY_MOVE_STALL) begin
                //reset all x, y pos
                x <= 0;
                y <= 0;
            end
            else if(state == SETUP_BOARD || state == SINGLEPLAYER_PLACE_SHIPS) begin
                // (x, y) -> {0, 1, ..., size - 1}^2
                // if rotate = 0, then the ship is horizontal. y should have no additional bounds, while x must be reduced by the ship length
                // similarly, if rotate = 1, then the ship is vertical, and x should have the additional bound
                if(~old_btnd & btnd) y <= y == size - 1 - (rotate ? ships[ship_num].length - 1: 0) ? y : y + 1;
                else if(~old_btnu & btnu) y <= y == 0 ? y : y - 1;
                else if(~old_btnr & btnr) x <= x == size - 1 - (rotate ? 0 : ships[ship_num].length - 1) ? x : x + 1;
                else if(~old_btnl & btnl) x <= x == 0 ? x : x - 1;
                else if(mouse_enable && (px != 4'hF) && (py != 4'hF)) begin
                    x <= px > size - 1 - (rotate ? 0 : ships[ship_num].length - 1) ? size - 1 - (rotate ? 0 : ships[ship_num].length - 1) : px;
                    y <= py > size - 1 - (rotate ? ships[ship_num].length - 1 : 0) ? size - 1 - (rotate ? ships[ship_num].length - 1 : 0) : py;
                end
                else begin
                    //if (x, y) coordinates out of bounds, then move to closest valid boundary
                    x <= x > size - 1 - (rotate ? 0 : ships[ship_num].length - 1) ? size - 1 - (rotate ? 0 : ships[ship_num].length - 1) : x;
                    y <= y > size - 1 - (rotate ? ships[ship_num].length - 1 : 0) ? size - 1 - (rotate ? ships[ship_num].length - 1 : 0) : y;
                end

            end
            else if(state == PLAY_MOVE || state == SINGLEPLAYER_PLAY_MOVE) begin
                // (x, y) -> {0, 1, ..., size - 1}^2
                // no further restrictions on shape due to ship size
                if(~old_btnd & btnd) y <= y == size - 1 ? y : y + 1;
                else if(~old_btnu & btnu) y <= y == 0 ? y : y - 1;
                else if(~old_btnr & btnr) x <= x == size - 1 ? x : x + 1;
                else if(~old_btnl & btnl) x <= x == 0 ? x : x - 1;
                else if(mouse_enable && (ox != 4'hF) && (oy != 4'hF) ) begin
                    x <= ox > size - 1 ? size - 1 : ox;
                    y <= oy > size - 1 ? size - 1 : oy;
                end
            end

            //for further functionality if needed
        end
    end

endmodule
`default_nettype wire
```

7.1.34 water.sv

```
////////////////////////////////////////////////////////////////////////////////
//
////////////////////////////////////////////////////////////////////////////////
`default_nettype none

module water(   input wire clk_in, rst_in,
                input wire [10:0] hcount_in,
                input wire [9:0] vcount_in,
                output logic [11:0] pixel_out
    );

    logic [9:0] sr;

    always_ff @(posedge clk_in) begin
        if (rst_in) begin
            sr <= 10'b00_0000_0000;
            pixel_out <= 12'hFFF;
        end else begin
            if ((hcount_in == 0 || hcount_in == 50 || hcount_in == 100 || hcount_in == 150 || hcount_in == 200 ||
                  hcount_in == 250 || hcount_in == 350 || hcount_in == 400 || hcount_in == 450 || hcount_in == 500 ||
                  hcount_in == 550 || hcount_in == 600 || hcount_in == 650 || hcount_in == 700 || hcount_in == 750 ||
                  hcount_in == 800 || hcount_in == 850 || hcount_in == 900 || hcount_in == 950 || hcount_in == 1000) && (
                  vcount_in == 0 || vcount_in == 50 || vcount_in == 100 || vcount_in == 150 || vcount_in == 200 ||
                  vcount_in == 250 || vcount_in == 300 || vcount_in == 350 || vcount_in == 400 || vcount_in == 450 ||
                  vcount_in == 500))
            begin
                sr <= vcount_in;
                pixel_out <= (sr > 768) ? 12'hFFF : 12'h07F;
            end else begin
                sr <= {sr[8:0], ~(sr[9] ^ sr[6])};
                pixel_out <= (sr > 768) ? 12'hFFF : 12'h07F;
            end
        end
    end

endmodule

`default_nettype wire
```

7.1.35 xvga.sv

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Update: 8/8/2019 GH
// Create Date: 10/02/2015 02:05:19 AM
// Module Name: xvga
//
// xvga: Generate VGA display signals (1024 x 768 @ 60Hz)
//
//                              ---- HORIZONTAL -----     ------VERTICAL -----
//                              Active                    Active
//                    Freq      Video   FP  Sync   BP      Video   FP  Sync  BP
//   640x480, 60Hz    25.175    640     16    96   48       480    11    2   31
//   800x600, 60Hz    40.000    800     40   128   88       600     1    4   23
//   1024x768, 60Hz   65.000    1024    24   136  160       768     3    6   29
//   1280x1024, 60Hz  108.00    1280    48   112  248       768     1    3   38
//   1280x720p 60Hz   75.25     1280    72    80  216       720     3    5   30
//   1920x1080 60Hz   148.5     1920    88    44  148      1080     4    5   36
//
// change the clock frequency, front porches, sync's, and back porches to create
// other screen resolutions
////////////////////////////////////////////////////////////////////////////////
`default_nettype none

module xvga(input wire vclock_in,
            output logic [10:0] hcount_out,    // pixel number on current line
            output logic [9:0] vcount_out,     // line number
            output logic vsync_out, hsync_out,
            output logic blank_out);

   parameter DISPLAY_WIDTH  = 1024;      // display width
   parameter DISPLAY_HEIGHT = 768;       // number of lines

   parameter  H_FP = 24;                 // horizontal front porch
   parameter  H_SYNC_PULSE = 136;        // horizontal sync
   parameter  H_BP = 160;                // horizontal back porch

   parameter  V_FP = 3;                  // vertical front porch
   parameter  V_SYNC_PULSE = 6;          // vertical sync
   parameter  V_BP = 29;                 // vertical back porch

   // horizontal: 1344 pixels total
   // display 1024 pixels per line
   logic hblank,vblank;
   logic hsyncon,hsyncoff,hreset,hblankon;
   assign hblankon = (hcount_out == (DISPLAY_WIDTH -1));
   assign hsyncon = (hcount_out == (DISPLAY_WIDTH + H_FP - 1));  //1047
   assign hsyncoff = (hcount_out == (DISPLAY_WIDTH + H_FP + H_SYNC_PULSE - 1));
       ↪ // 1183
   assign hreset = (hcount_out == (DISPLAY_WIDTH + H_FP + H_SYNC_PULSE + H_BP -
       ↪ 1));  //1343

   // vertical: 806 lines total
   // display 768 lines
   logic vsyncon,vsyncoff,vreset,vblankon;
   assign vblankon = hreset & (vcount_out == (DISPLAY_HEIGHT - 1));   // 767
   assign vsyncon = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP - 1));  // 771
   assign vsyncoff = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP +
       ↪ V_SYNC_PULSE - 1));  // 777
   assign vreset = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP + V_SYNC_PULSE
       ↪ + V_BP - 1)); // 805

   // sync and blanking
   logic next_hblank,next_vblank;
   assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
   assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
   always_ff @(posedge vclock_in) begin
      hcount_out <= hreset ? 0 : hcount_out + 1;
      hblank <= next_hblank;
      hsync_out <= hsyncon ? 0 : hsyncoff ? 1 : hsync_out;  // active low

      vcount_out <= hreset ? (vreset ? 0 : vcount_out + 1) : vcount_out;
      vblank <= next_vblank;
      vsync_out <= vsyncon ? 0 : vsyncoff ? 1 : vsync_out;  // active low

      blank_out <= next_vblank | (next_hblank & ~hreset);
   end
endmodule

`default_nettype wire
```

7.2 Python Scripts

7.2.1 Audio Generation Script

```python
from lib6003.fft import fft,ifft
from lib6003.audio import wav_read,wav_write
from math import pi
import matplotlib.pyplot as plt
import numpy as np


def filter(wav, fs, interval):
    a, b = interval[0], interval[1]
    band_filter = list(map(lambda x: 1 if 2*pi/fs*a <= min(abs(x-pi), abs(x+pi)) <= 2*pi/fs*b else 0, np.linspace(-pi, pi, len(wav) + 1)))
    output = ifft([i*j for i, j in zip(fft(wav), band_filter)])
    print(output[:20]) #verify that there are no imaginary parts
    #take real part of output only
    output = np.array(list(map(lambda x:x.real, output)))
    #normalize to maximize volume
    output = output / np.max(abs(output))
    return output

nr = 6000 #desired sampling frequency

filepath = 'FILEPATH'
wav,fs = wav_read(filepath)
output = filter(wav, fs, (0, nr / 2))[::fs//nr] #filter out all frequencies above Nyquist frequency
#print(output[:20])
f = lambda x: round(x.real*2**7) / 2**7 #to change to 8-bit file
no = list(map(f, output))
print(no[:20])
wav_write(no, nr, 'out.wav')
with open('out.txt', 'w') as f:
    f.write("""memory_initialization_radix=2;\nmemory_initialization_vector=\n""")
    for i, l in enumerate(no):
        val = bin(round(l * 2 ** 7 + 2**7))[2:] # offset binary
        f.write(val.zfill(8) + ',\n')
print(len(no))
```

7.2.2 Random Walk Verification

```python
import numpy as np

def generate_walk_matrix(n):
    M = [1/2 if (i == 0 and j == 0)
              or (i == n-1 and j == n-1)
              or abs(i - j) == 1
              else 0 for i in range(n) for j in range(n)]
    return np.array(M).reshape(n, n)

game_sizes = [9, 10, 11, 12]
for n in game_sizes:
    print(np.linalg.matrix_power(generate_walk_matrix(n), 500))
```