Jacob Linder

Charles Zhang

8 December 2021

Group 5

COM SCI M152A, Lab 5

Lab 4: Battleship Report

1 Introduction

In this lab, we developed a simplified version of the children's game Battleship on the Nexys 3 FPGA board, utilizing its buttons, switches, seven-segment displays, and VGA port. Our game allows 2 players to place 3 2x1 ships each on a 5x5 board displayed on a monitor using the VGA connection. Once they have placed their ships, they then take turns firing at each other's boards, attempting to hit all of their opponent's ships.

In placement mode, players can move the current ship being placed around the board by encoding their ship's location using binary instructions on the switches. The middle button can then be used to place each ship. Ships cannot be placed out-of-bounds or overlapping another ship, any attempts to do so will be rejected. Once player 1 has placed all 3 of their ships, player 2 will then be prompted to do the same.

Once ships are placed, the board will enter game mode. In this mode, players can aim their shots using the left 4 switches to decide the x-coordinate of their shot and the right 4 switches to decide the y-coordinate. Once the switches are set, the player can then shoot with the middle button. If they hit an opponent ship, they will be awarded with another turn, otherwise the other player will have an opportunity to shoot.

2 Design Description

We decided to model the overall design of our game on the following finite state machine:

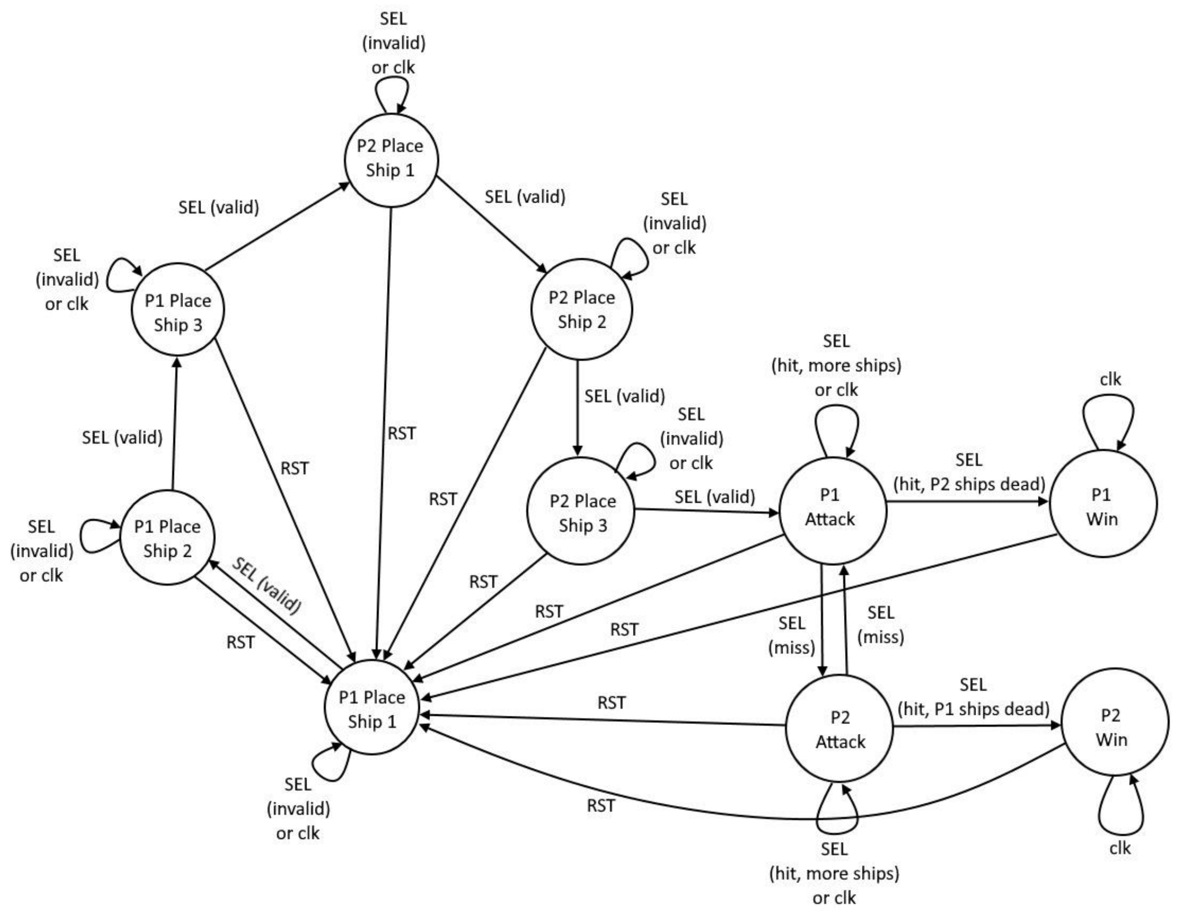


Figure 1: Finite state model of Battleship game

In this FSM, all of the `Place` states represent the placement mode, the `Attack` states represent the game mode, and the `Win` states represent the win mode. `SW0-5` and `ROT` determine whether `SEL` is valid or invalid in placement mode, and `SW0-5` determine whether `SEL` is a hit or a miss in game mode. Any time `RST` is pressed it returns to the initial game state, where player 1 chooses their first ship placement. `clk` never changes any of the game states. After a player wins the game, the only way to exit that state is through the `RST` button.

2.1 `battleship.v`

`battleship` is our game's top module. It takes in the master clock `clk`, the middle button `SEL`, the right button `RST`, the top button `ROT`, and switches `SW0-5`. It then outputs a

7-bit `cathode` value and 4-bit `anode` for the seven-segment display and `hsync`, `vsync`, and a 8-bit `rgb` value for the VGA display.

First, this module sends each input into `debouncer` so that the rest of the modules can assume the inputs have metastability and are debounced. This results in the set of signals `sel`, `rst`, `rot`, and `sw0-5`.

We then passed the 100 MHz `clk` signal into the `clock_divider` module, dividing it into a 25 MHz `clk_ssd` for the SSD and a 763 Hz `clk_vga` for the VGA.

Next, we pass signals into the `game_logic` module, which outputs `mode`, `turn`, `active_ship`, `ships_locs`, `ship_bitmap`, and `miss_bitmap`. This module handles the general logic for the game's placement mode and game mode, generating outputs to be passed to the VGA to create the display. `mode` represents the current mode the game is in, `turn` represents the which player's turn it is, `active_ship` represents the coordinates of any active tiles, `ships_locs` represents the coordinates of each of the active player's ships, `ship_bitmap` represents the state of each of the current player's ships, and `miss_bitmap` represents coordinates where the current player has missed.

After the current turn is calculated, we pass `clk_ssd`, `turn`, and an `anode` and `cathode` into the `ssd` module to output the current player on the board's seven-segment display, either player 1 or player 2.

Finally, we output a graphic of our game by passing `clk_vga`, `rst`, `active_ship`, `ship_locs`, `ship_bitmap`, and `mode` into the `vga` module. This encodes the correct state of each of the board spots as a specified color, which is output through `hsync`, `vsync`, and `rgb`.

2.2 `clock_divider.v` and `debouncer.v`

For both the seven-segment display and VGA display, we need clock signals to multiplex their outputs. As a result, we implemented `clock_divider`, which translates the master clock signal into 2 signals of desired frequency. We did this by initializing a 17-bit counter, which counted up every clock cycle. We set the `clk_vga` signal to the second least significant bit of the counter, effectively dividing the period of the 100 MHz clk by 4, resulting in our 25 MHz clock. We set the clk_ssd signal to the most significant bit of the counter, dividing the master clock signal by 2^{17}, resulting in our 763 Hz clock.

In order to implement input signals from the FPGA board, we have to deal with the issues of debouncing and metastability. Debouncing is required due to the instability of buttons and switches on the board, which leads to considerable noise when pressed or switched. Without any filtering of this noise, it would cause many unintentional input signals to come through, resulting in a jittery implementation. On a similar note, the asynchronous nature of these input signals forces us to deal with metastability in our implementation. This means that we must ensure that input signals are held constant for a period of time before and after a clock edge at which they will be processed. Otherwise, this may create variance in our outputs due to input signals changing when they are not supposed to.

To ensure metastability in `debouncer`, we start by reading any asynchronous input into a register `store`. The rest of this module will then read from `store` on the posedge of `clk`, synchronizing the input signal with the clock cycle. This signal will be read into another register temp_signal, which represents the current signal that the module is reading in. In order for this signal to be considered valid, it must be held constant for 1 ms. We track this with a counter that resets to `0` if the signal is lost. If the signal is held constant for 1 ms, then it is output as `signal_f`. This effectively strips out any noise between button presses.

2.3 `game_logic.v`

Placement mode

The game is initialized with `mode` as `0` and `turn` as `0`, representing player 1's turn in `PLACEMENT` mode (`mode = 00`). Player 1 then has to place all three of their 2x1 ships in the 5x5 grid, such that no ship is overlapping or is out of bounds. We store each ship location as a pair of 6-bit registers, `p1_ship_x_0` and `p1_ship_x_1` for each of the three ships (`x` = 0, `x` = 1, and `x` = 2). The first three bits in each register represent the row (from range 0 to 4), and the last three bits in each register represent the column (also from range 0 to 4). To choose the location of the ship, the player uses the binary representation of { `sw5`, `sw4`, `sw3` } to choose the column, and { `sw2`, `sw1`, `sw0` } to choose the row. The player can also toggle the orientation of the ship (horizontal or vertical) with `rot`. The representation of the ship's location as two

3-bit coordinates corresponds to the top-left most segment of the ship, stored in `p1_ship_x_0`, and the value of `p1_ship_x_1` depends on the on whether the ship is horizontal (the column value is be the same, but the row value is added to by 1), or vertical (the row value is the same, but the column value is added to by 1). If any of the coordinates for the current ship are out of bounds, the ship is placed entirely out of bounds so that there is not just a single coordinate showing. A ship can be placed if both of its coordinates are in bounds and `sel` is pressed. When a ship is placed, the `x` value is increased for `p1_ship_x_0` and `p1_ship_x_1`. Once all three ships are placed for player 1, `turn` is changed to `1` and player 2 repeats the same process and places their three ships. After all six ships are placed, the game's `mode` switches to `01`, representing game mode.

The output from this module is the `active_ship`, the two 6-bit coordinates representing where the currently selected ship is. It also outputs `ship_locs`, which is a 36-bit array that holds all of the coordinates for the ships placed so far. This is muxed based on the current player turn. It also outputs the 6-bit `ship_bitmap`, which is initialized as all `1`'s in placement mode, and the 25-bit `miss_bitmap`, which is initialized to all `0`s.

Game mode

Once each player places their three ships and the game is in `GAME` mode (`mode = 01`), player 1 will begin by selecting a coordinate to fire at on player 2's board. All of the ships are hidden, so player 1 has to make a good guess. The coordinate selection is the same mechanism as in placement mode, except that this time it is controlling a single coordinate rather than two adjacent coordinates. Player 1 presses `sel` to confirm their decision. If the shot is a hit, the corresponding bit to that ship's segment in `standing_ships` turns to `0`, and the player goes again. If the shot is a miss, the coordinate in `p2_miss` (the 25-bit miss bitmap for player 2's board) is changed to `1` and `turn` is changed to allow player 2 to play. Once all of the ship segments for a player are hit, detected by checking whether they have any undestroyed ship segments in `standing_ships`, then `mode` is changed to `WIN` (`mode = 10`) and the winner is displayed on the seven-segment display.

2.4 `ssd.v`

To handle our display, we had to understand the functionality of the seven-segment display on the FPGA. Each seven-segment display is encoded by a cathode value, which tells the board which of the seven segments should be illuminated. Each segment is encoded as follows:

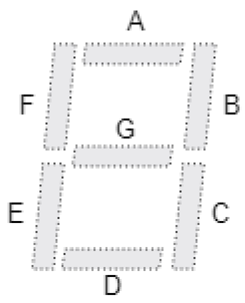


Figure 2: Seven-segment display encoding

Our game design requires us to encode the middle 2 displays on the board, however, the board only provides access to a single display at a time, determined by the anode value it is passed. As a result, our seven-segment displays have to be encoded one at a time, quickly changing the cathode and anode values so that the display looks consistent to the human eye.

To accomplish this, we quickly multiplex through the `anode` values using `clk_ssd`, changing the `cathode` values depending on the position of the digit. If the `anode` value encoded the outermost 2 digits, we would set `cathode` to a blank representation. If the `anode` value encoded the second digit, we had the `cathode` encode a letter P. Finally, if the `anode` value encoded the third digit, we had the `cathode` encode either a 1 or a 2, depending on the value of `turn`.

2.5 `vga.v`

In order to implement our VGA display in `vga`, we incorporated demo code from the following link: *Draw VGA color bars with FPGA in Verilog - Forum - FPGA*. This code initializes 2 counters, `h_counter` and `v_counter`, which determine the current pixel that is being encoded, resetting when they hit dimensional bounds. They are also used to generate the sync pulses `hsync` and `vsync` that are output from the module.

Our module builds upon this demo code by dividing the screen into 5 rows and 5 columns, using the values of `h_counter` and `v_counter` to determine the current row and column. This process pays special attention to if the counter is currently out of bounds. We then check to see if the current row and column are out of bounds, and output the RGB encoding of black if they are. Next, we check to see if any player has won by checking the value of `mode`, setting the entire screen to green if so. Then, we check to see if `h_counter` and `v_counter` are at the borders of a board tile, outputting the RGB encoding of black to generate grid lines. Finally, if all prior checks were passed, we output the color of the tile at the designated row and column.

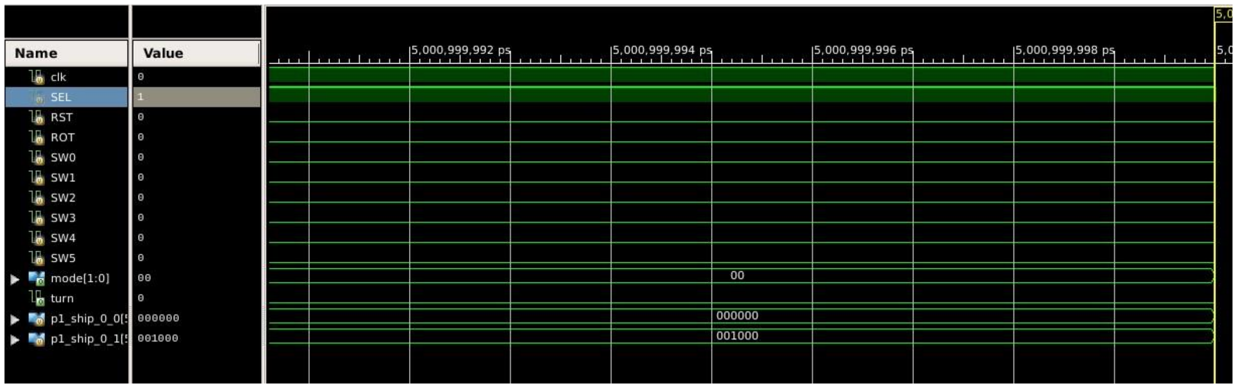
First, we check if the row and column are the location of a ship in `ship_locs`. If they are, we then start by checking if the game is in placement mode and if the active cursor defined by `active_ships` is on the current row and column. If it is, we set the tile to red, representing an overlapping ship. Next, we check to see if the corresponding bit in `ship_bitmap` is `0`, which marks the current ship as "hit". In this case, we once again set the tile to red, representing a hit ship in game mode. Then, we check to see if the current position is active, in which case we set it to green. Finally, we check if the corresponding bit in `ship_bitmap` is `1`, which marks the current ship as "alive". In this case, we set the tile equal to purple if the game is in placement mode, representing one of the current player's ships, or blue if the game is in game mode, hiding an opponent player's ship.

If the current row and column are not the location of a ship in `ship_locs`, we start by checking if the current tile is active. If it is, we set it to green. Next, we check through miss_bitmap, checking if the corresponding tile should represent a miss. If it does, we set the tile to yellow. Finally, if none of the prior checks succeeded, we set the tile to blue, representing an empty tile.

3 Simulation Documentation

To test our game, we developed a testbench that was designed to test basic game states. It starts by placing player 1's ships horizontally, followed by player 2's ships vertically. It then plays through the game mode as player 1, showcasing a hit and a miss. Finally, it sinks all of player 2's ships, demonstrating the game's win condition.

The following waveform shows a simple horizontal placement by player 1:

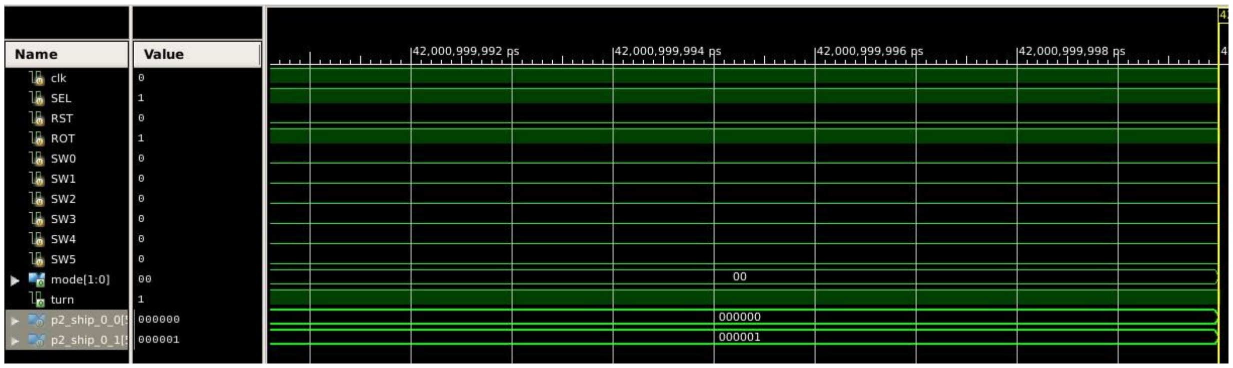


Name	Value
clk	0
SEL	1
RST	0
ROT	0
SW0	0
SW1	0
SW2	0
SW3	0
SW4	0
SW5	0
mode[1:0]	00
turn	0
p1_ship_0_0[5	000000
p1_ship_0_1[5	001000

Figure 3: Waveform diagram of player 1’s placement

Here, we can see that `mode` and `turn` are both `0`, indicating the game is in placement mode and that it is player 1’s turn. We can see that `SEL` has been set to high, and that player 1’s first ship has been placed at `000000` and `001000`, which represent the coordinates (0, 0) and (1, 0), respectively. This tells us that players can successfully place ships horizontally in valid locations.

After placing the rest of player 1’s ships, we can see the following waveform for player 2’s placement mode:

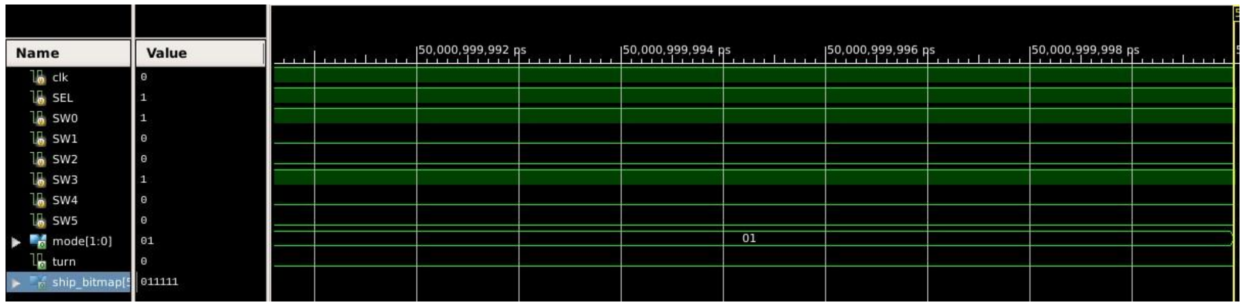


Name	Value
clk	0
SEL	1
RST	0
ROT	1
SW0	0
SW1	0
SW2	0
SW3	0
SW4	0
SW5	0
mode[1:0]	00
turn	1
p2_ship_0_0[5	000000
p2_ship_0_1[5	000001

Figure 4: Waveform diagram of player 2’s placement

In these diagrams, we can see that `mode` is still `0`, but `turn` is now `1`. This means that it is now player 2’s turn in placement mode. We can see that `SEL` and `ROT` have both been set to `1`, indicating that a vertical ship should have been placed at (0, 0). Looking at `p2_ship_0_0` and `p2_ship_0_1`, we can see that there are now ships at `000000` and `000001`, which encode the coordinates (0, 0) and (0, 1), respectively. This tells us that players can successfully place ships vertically in valid locations.

We can then place the rest of player 2's ships and observe the following waveform:

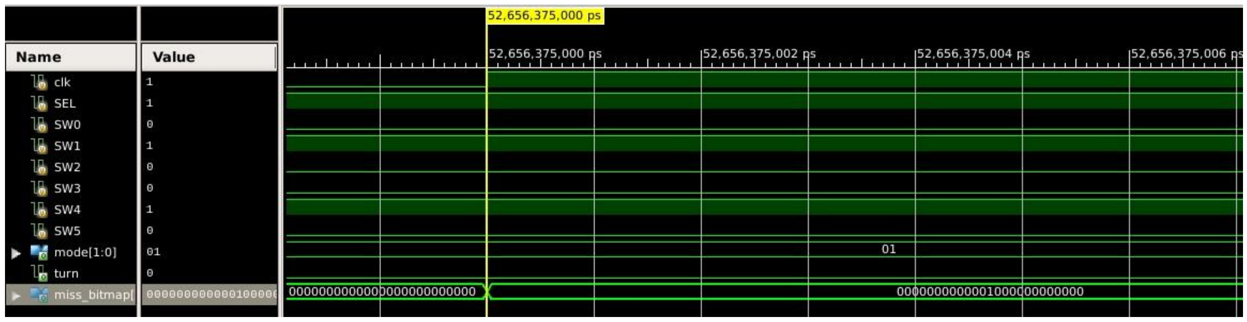


Name	Value
clk	0
SEL	1
SW0	1
SW1	0
SW2	0
SW3	1
SW4	0
SW5	0
mode[1:0]	01
turn	0
ship_bitmap[5	011111

Figure 5: Waveform diagram of a player 1 hit

In this diagram, we can see that `mode` is `1` and `turn` is `0`, representing player 1's turn in game mode. We can also see that `SEL` has been set to `1`, with `SW0` and `SW3` being set to `1` as well. This indicates that player 1 fired at row `001` and column `001`, or (0, 0). Since we know player 2 placed a ship there, we can expect that this would hit a ship. We can see this in `ship_bitmap`, which now has the first bit set to `0`, indicating that the first ship has been hit, showing that our hit logic is successful.

We can repeat the same process, instead shooting at a tile that doesn't contain a ship to get the following waveform:



Name	Value
clk	1
SEL	1
SW0	0
SW1	1
SW2	0
SW3	0
SW4	1
SW5	0
mode[1:0]	01
turn	0
miss_bitmap[	0000000000000100000

Figure 6: Waveform diagram of a player 1 miss

Here, we see that player 1 has aimed at row `010` and col `010`, or (2, 2). Player 2 did not place a ship here, so we know that this shot must miss. We can see that this is true in `miss_bitmap`, where the middle bit has been set to `1`, indicating a miss at (2, 2). This trial shows that our miss logic was successfully implemented.

Finally, we return to player 1 and hit the rest of player 2's ships. In doing so, we arrive at the following waveform:

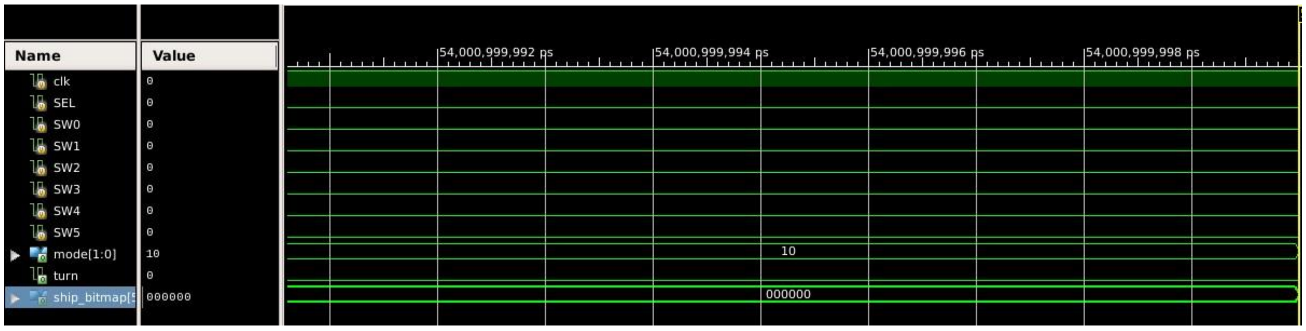


Figure 7: Waveform diagram of player 1 win state

In this diagram, we can see that `ship_bitmap` is `000000`, indicating all of player 2's ships have been hit. We can then see that the mode has been set to `10`, which indicates that the current player has won. Noting that the current player is player 1, we know that our win condition logic is also correct.

These waveforms tell us that the basic logic of our game is functional. Edge cases, such as out-of-bounds placements and turn switching logic were tested using the VGA display to visualize the results of each test.

4 Conclusion

In this lab, we developed a fully functioning game of Battleship that is controlled with switches and buttons on the Nexys 3 FPGA board, and was displayed on a seven-segment display and on a computer monitor through use of a VGA. Our design incorporated submodules that built up our top module, `battleship`. `clock_divider` was used to transform the master clock into a set of clock signals that were needed to correctly output the seven-segment display and the VGA signals. `debouncer` was used to debounce and ensure metastability on our input signals. `game_logic` is used to store and update the current state of the game based on the input decisions of the players. `ssd` was used to output the current player onto the seven-segment display. Finally, `vga` was used to translate the game logic into a grid of colors corresponding to the state of the board, and was outputted for display on a monitor.

The biggest difficulties we encountered were space location on the board. We originally built the entire game on a 10x10 board with variable length ships: one of length 5, one of length 4, two of length 3, and one of length 2. We stored the game state in a 300-bit array for each player, which represented the concatenated rows of the 10x10 board where each element was a 3-bit value that determined the value of the grid -- `HIT`, `BLANK`, `SHIP`, etc. To calculate the current position, we would calculate `row * 30 + col * 3` and take the following 3 bits. This requires the board to create a mux for every one of the possible 3-bit positions in this array, which took up a massive amount of space, especially since we needed to have this logic every time we did a board check or set. We therefore continually ran out of memory of the FPGA, so we had to scale down to a 5x5 board with three ships each of length 2.

We also had a significant number of issues setting up the VGA display using the demo code. We originally attempted to modify the dimensions in the module, but it resulted in an incoherent set of colors on the monitor. After a few sessions of troubleshooting, we eventually decided to revert back to the original demo code and simply build on top of it, leading to our final design.

Overall, I think that we both learned a lot by doing this lab and ended up with an enjoyable game to play, whereupon we could look with pride.