FPGA Battleship

Subtitle as needed

Jonathan Nguyen, Brian Wills
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
E-mails: jtnguyen2@oakland.edu, bdwills@oakland.edu

***Abstract*—The purpose of this project was to attempt to make a smaller rendition of the popular game "Battleship" out of VHDL and an old Nexyus-4 DDR board. With the use of components [1] such as registers, multiplexors, 7-segment displays, and conceptual circuits such as comparators, serializers, and a finite state machine (FSM), a digital version of the game could be built to store ship locations, register hits and misses, and keep track of player turns and coordinate position. This report covers the conception of the project design, methodology, and results that followed.**

I. Introduction

For the Final Project, we decided to recreate the classic game of Battleship using our field-programmable gate array (FPGA) boards and the VHDL programming language.

Battleship is a simple two-player game where players take turns taking "shots" on a coordinate grid in an attempt to sink the other player's fleet of ships. The game ends when a player's fleet is completely sunk. [2]

The motivation was to create a project of significance that would be easily understood by all group members. In a previous class, some group members had heard rumors of some students recreating classic DOOM for their final project, and this was born of that idea being discussed, with some requirements being relaxed to ensure that it can be made. Such requirement relaxing is related to the number of hits a ship can receive, how many rulesets the game has, how large the map and ships themselves are.

II. Methodology

A. Alterations to Original Game

The game will be based singularly upon the "classic" interpretation of Battleship, with one player choosing a single spot on the board per turn to attempt to sink the other player's ships. Normally the game board is a ten-by-ten grid[2], but for the purposes of simplifying the game for this project, the group decided to drop the size of the board to a seven-by-seven grid. This was done to shrink both the size of the amount of data needed to retain due to misses and to simplify the logic needed to ensure smooth play.

In addition to that, it was decided that each ship would only occupy a single tile of space rather than have different ships of different sizes in different orientations. Such logic was felt to be unnecessarily complicated for the intended scope of this project.

However, it was agreed that the players would need something to judge their positions by, and as such the ship position selected by six (three for each axis) board switches are displayed on one bank of four 7-Segment LED displays.

B. Gameplay States

A typical game was designed to consist of the following stages: Reset Screen (S0), Player Ship Position Selection (S1, S2), Player Turn (S3, S4), Player Selection Error (S6, S7) and End Game (S5).

As designed, Reset Screen is a resting state for the system and serves as a visual notice to the players that play has yet to commence. To exit the reset screen and begin play, they only need to engage the "load" switch.

Figure 1: State 0, Reset Low State

Figure 2: State 0, Reset High State

Player Position Select is when the players input the position of their ships, selecting them in ranks of five and engaging the "load" switch each time to confirm their selection and loading the data point into the system.

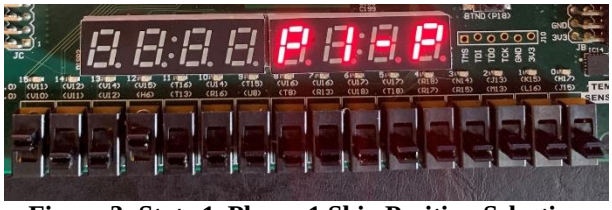


Figure 3: State 1, Player 1 Ship Position Selection

When Player One has successfully loaded data into their given memory registers, the finite state machine automatically switches over to Player Two, and displays such on the 7-Segment Display.

Figure 4: State 2: Player 2 Ship Position Selection

Once both player ship position registers are full, the state advances to the normal turns of play. Examples of which are provided below.

Figure 5: State 3, Example Player 1 Turn

Figure 6: State 4, Example Player 2 Turn

After Player Two has done the same, the game switches into the 'default' state, or the two states that the game will reside in for the longest period of time; turns being taken between both players. This time, rather than have the player selecting positions for the ship, it is what position to compare against that of the opponent.

Should either player choose a coordinate that had already been chosen, the game would enter an error state until the player entered another position. For this reason, the initial design called for a VGA visual display to show each player where they had missed or hit previously, but was deemed too complex following further study and research. Since the idea for a VGA display was dropped, the use of a 7-segment serializer was decided to be the main display resource.

Figure 7: State 5, End Game

Figure 8: State 6, Player 1 Error

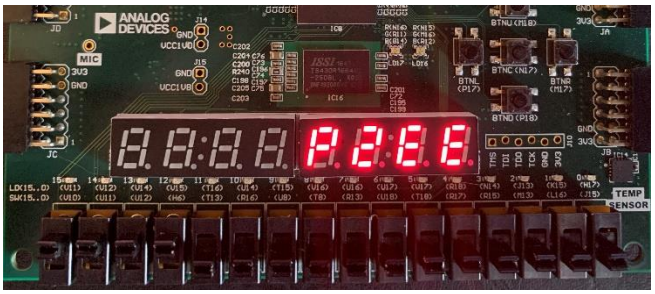


Figure 9: State 7, Player 2 Error

C. Controls

As designed, the X and Y coordinates for a given selection utilize the first six switches on the class board, iterating upwards in the fashion of a binary counter. The first three switches are mapped to the X coordinate, and the second three to the Y coordinate. Switch Fifteen (SW[15]) is mapped to the reset command, and Switch Fourteen (SW[14]) is the load command, which triggers the position to be loaded into a ship position register or compared to the opposite player's ship position registers.

D. Components

The following is a list of components that are core pieces to the final circuit and design of the game.

D1. Registers

Registers were used as storage spaces to store ship locations for each player as well as a copy of the field grid for comparison with regards to hits and misses. There were ten, six-bit registers used for storing ship coordinates split evenly between the two players. This was made possible due to the simplification of game rules mentioned earlier. There are two 49-bit registers that are used for both comparing (a comparator's function) against and storing (a register's

function) a 49-bit number that represents the game board, again one for each player.

D2. Comparator

The main comparator is, like the other two main circuit components, dual purpose. During the first two states of play (S1, S2), the comparator functions as a pass through that occasionally sets off an alarm when a player's input signal does not get cleared by a subsequent one soon enough. This function during these states is ignored by the FSM.

During the second portion of play, the comparator fulfills the role of determining if a given player has achieved a 'hit', or when their guess has matched with that of the opposing player. Once the player position registers have been filled, the FSM switches the 'fn' port to digital low, which changes the functionality of the main comparator.

This function outputs the information from the player into one of two states. In this next state first variant, the player's input is not equal to the opposing player's ship position, and is pushed into the demultiplexer between the two hybrid comparators and then into the appropriate one for further analysis.

In the second variant of the regular comparator function, the player's input is the same as the given value by the system register, and is recorded as a positive value on "comp" signal. This iterates a variable in the FSM (ship1 for Player 1, ship2 for Player 2). Once and only once either of these variables reaches five does the game shift into the end game, declaring the game to be over.

D2.1 Hybrid Comparator

The hybrid comparator is called such because it is not merely a comparator, nor does it compare values in the manner that would normally befit the name. The hybrid comparator has the job of comparing a given position on the board with the positions that are stored in the memory portion.

To compare the values and ease the computational burden on the system, it was determined that the best way to describe the board was with a single, 49-bit binary number as there are forty-nine places on the board, numbered in the same fashion as a binary number (0 to 48) to simplify the coding process. From there, each six-bit number is assigned a numeric value based upon the figure below, which was transformed into a large if-statement.

Once the numeric value was established, the module determined if it's resident 49-bit number contained a '1' at that location [miss(b) as it is called in the code]. If there is no positive logic, then it is swapped to positive and the turn is over.

If there is already a positive logic at that location within the number, the hybrid comparator notifies the FSM that there is a duplicate entry, and the system goes into an error state until the given player loads a new position into the system.

	1	2	3	4	5	6	7
A	0	1	2	3	4	5	6
B	7	8	9	10	11	12	13
C	14	15	16	17	18	19	20
D	21	22	23	24	25	26	27
E	28	29	30	31	32	33	34
F	35	36	37	38	39	40	41
G	42	43	44	45	46	47	48

Figure 10: Board Layout with Enumeration

D3. Serializer

The serializer component is based off of the VHDL code given from ECE 2700 notes/website [1]. The main purpose of the serializer is to cycle through the bank of four 7-segment displays since the FGPA can only turn on one display at a given time. The main code modifications were that the displays are tied to the states of the main FSM (not the small internal one that cycles through the displays) and the hex-to-7-segment decoder has been modified to display A-G and 1-7 for the coordinates (ex: A7, G1, F6, etc.) as well as a "P1" or "P2" set to show which player's turn it is. As stated earlier, since the idea of using a VGA display was dropped, the initial goal was to use all eight 7-segment displays on the FGPA, but after consideration of what inputs were being used and what was to be displayed, it was settled on using only four displays to show player turn and cursor position for the majority of the gameplay.

D4. Demultiplexer

Used to simplify data flow from major modules. Each player has a dedicated demultiplexer to enable data flow from the comparator into each position register. Around the main comparator there are two demultiplexers, one to direct data to the player appropriate bank of registers and one to direct data to the appropriate hybrid comparator/register.

D5. Multiplexers

There are three multiplexers in the circuit. Two are five-input multiplexers and are used to feed an opposing player's ship position data into the main comparator. However, due to an initial desire to simplify the construction of each circuit component, a third multiplexer was added between these five-input MUXs and the comparator to only allow the appropriate player's data into the main comparator.

D6. Integrators

There is one integrator in the circuit, right at the beginning of the data path. This integrator turns the player input into the corresponding 6-bit vector that is used for the coordinate position that is used throughout the rest of the circuit.

D7. Multi-Input OR Gate

This module served as the key input for determining if the player's ship registers had been filled up. Each register has a signal that goes low if the held value is anything but "000000". Once each signal has gone low, the signal output for this module goes low, which informs the FSM to move along to the next state of play.

D8. FSM (Finite State Machine)

The FSM controlling this project was envisioned and built as the central authority on what happens and when in the system. For this reason, it has numerous conditions for state changes, and each state comes with a slew of different values for each of its outputs.

One crucial part of the FSM was how to iterate the mux controller (rmuxc) between two different players, clear it, and have it ready to go as soon as the state changes without causing a wrong value to be shunted to the wrong portion of the system. To solve this, two separate mux values were created, iterated and cleared asynchronously (muxcs, muxcs1). From here, the two mux values were given a series of when statements that culminated in the output of (rmuxc).

Additionally, an issue of having the FSM change state early arose, leading to the adoption of a process to detect when a new value was put into the system (pi), and to change only when such a thing occurred. To this end, the (fsms) signal was tied into the state change statements to ensure that only when it was proper time did the FSM allow the system to change states.

Of course, this was not the only version of this issue that arose; it was determined after trial and error that having the mux controller variables go up one value past the number of ship registers ensured proper timing, for their part. Should the value be lower, then the state would change well before the player finished flipping the switch on his or her turn, and result in a cascade of unauthorized input and calculations upon entering subsequent states.

In the figure below is a simplified state machine diagram for the project. Due to the large number of output control signals in any given state, it was determined to be best to leave them out of the drawing.

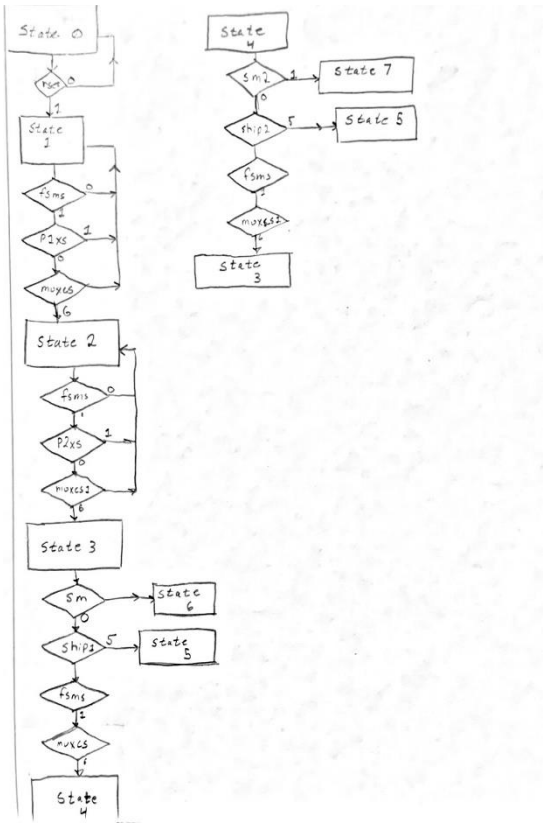


Figure 11: Sketch of System Finite State Machine

III. Experimental Setup

For the final project, the game was to be implemented on a Nexys-4 DDR or Nexys A7-50T FGPA board using Vivado software for coding and simulation purposes. All code was either created or modified from provided code from the class website.

For simulation conditions, the time spent simulating was expanded from the default one thousand nanoseconds to one-hundred-sixty milliseconds. The time was lengthened significantly versus normal simulation standards to allow for more realistic testing of the system. An early solution to the timing issue between the hybrid comparator and the multiplexer associated with them was to provide one or both with a delay in sending or receiving the data.

In the nanoscale environment, such a delay of anywhere from one to four clock cycles was sufficient enough to provide stable results. However, once scaled up to a more realistic time scale, such minor changes were ineffective at solving the issue, thus necessitating a redesign of the system.

Additionally, due to the large number of digital signals being tracked at any one-time, custom wave-configuration files were saved to provide easier access to the data and are included in the report

IV. Results

The final outcome is best classified as mixed results. While the code could be simulated, implementation proved

problematic, and therefore was not fully realized. One of the problems encountered is that there was an error which was causing signals synchronizing with the wrong player for the 49-bit registers during simulation.

Another issue that came up is that during actual attempts at implementing the circuit on the FGPA, the circuit would be stuck on the opening state and would not proceed past the initial state, even though the circuit would not show such a thing during simulations. Root cause analysis revealed this was an error in synchronizing three elements: the button that enabled the reset (initially tied to CPU RESET pin on the board), the reset signal value in the individual components, and the method of iterating the serializer.

On the other hand, most of the components successfully worked in stand-alone testing and showed correct values when simulated using a testbench file outside of the noted above issues.

In addition, one issue that came up quite frequently in encoding some of the more complex items (FSM, hybrid comparator) was the fact that certain statements and case statements could not be used within one another in the given version of VHDL that Vivado uses by default, the 1995 release. However, such statements were widely supported in the 2008 release version of VHDL, but the simulation environment for Vivado does not support the 2008 version.

This limited functionality hampered development, most clearly seen in the case of the hybrid comparator's massive if-elseif statement chain. Something of this magnitude would normally use a when-statement, but because it was tied into a clock process, the '95 release did not support it.

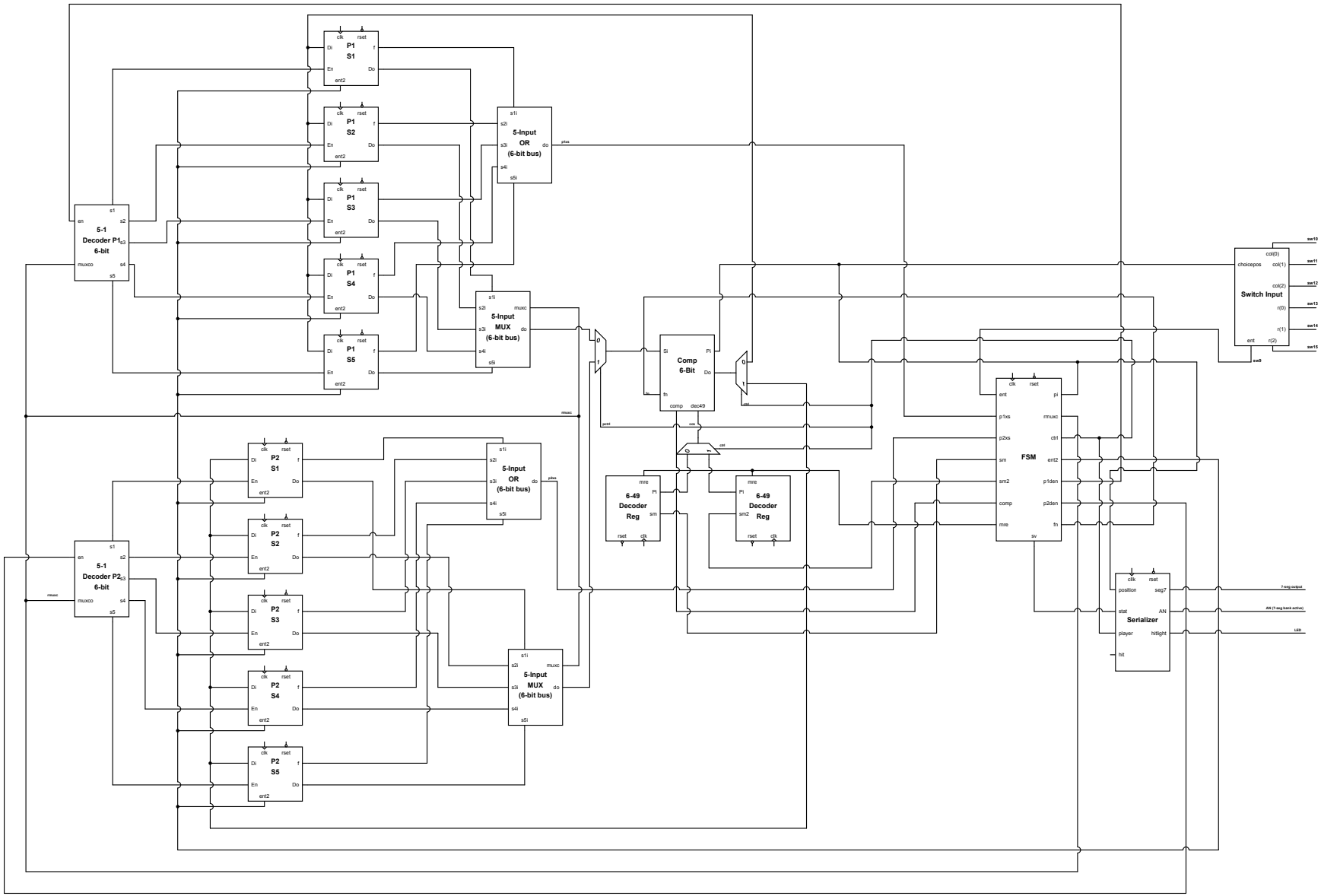
Conclusions

In conclusion, while the project was not fully implemented, it was not a wasted effort. While the implementation was not perfect, the project was able to display various game states, display correct coordinates, input and store ship locations, and simulations showed the circuit was able to handle correct comparisons for hits and misses. Our group was able to gain valuable insight into how VHDL code works and crucial experience with reading and debugging digital circuits from timing diagrams. Some of the main issues still left to be fixed are the delayed synchronizing signals between the hybrid comparator and the FSM as well as sorting out the FSM internal issues currently plaguing the current build, as shown below. Once those are fixed and provided the solutions do not break anything else the overall project could then be considered complete. For improvements, one of them would be implementing a VGA display for a more visually appealing and ease of understanding as well as providing a better player interface.

References

[1] "RECRLab", Electrical and Computer Engineering Department, Oakland University
https://www.secs.oakland.edu/~llamocca/VHDLforFPGAs.html

[2] "Battleship (Game)", Wikipedia.org
https://en.wikipedia.org/wiki/Battleship_(game)
.


P1 S1
P1 S2
P1 S3
P1 S4
P1 S5
5-Input OR (6-bit bus)
5-Input MUX (6-bit bus)
5-1 Decoder P1 6-bit
P2 S1
P2 S2
P2 S3
P2 S4
P2 S5
5-Input OR (6-bit bus)
5-Input MUX (6-bit bus)
5-1 Decoder P2 6-bit
Comp 6-Bit
6-49 Decoder Reg
6-49 Decoder Reg
FSM
Switch Input
Serializer