ECE 350 Final Project: Battleship

Harry Guo and Wei Tang

Design:

The design of our project was to make a simplified and digital version of battleship using the FPGA and VGA display. The concept was pretty simple. There are two players, each with their own half of the screen (representing the respective player's board). The game starts with each player placing the location of their ships on their board; the ship sizes are length 1, 3, and 5 (that can be oriented either horizontally or vertically) with the overall board size being 6x8 grid on each side. At the placing stage of the game, grid locations with a ship placed will become black for the player to keep track of where they are. However, while switching to the placing stage of Player B, Player A's ships will be hidden from the screen. After both players are finished placing their ships, the screen hides all the ships, and the players can begin attempting to "bomb" each other's ships. The grid locations will display different colors based on whether it is a 'hit' or a 'miss'. The game will continue to run until one player has completely eradicated the opposing player's ships.

Our design was relatively straight-forward. All visuals would be done via hardware. After the last lab, we used the base vga_controller module that was given to us and altered it to display the graphics we wanted to and the movement of the player's cursor on the screen. We decided to use hardware for visuals because it ensured that there would not be any major timing issues between the FPGA and the VGA display and also because we wanted to use the dynamic memory of our processor for keeping track of the gameplay. For the grid, we stored the status of the 96 grid squares (2*6*8) in the first 96 locations in our DMEM .

Our processor was used to keep track of ship locations, the status of each ship, and the stages of the game. To keep track of ship locations and the statuses of each ship, we used a doubly clocked memory (one clock to read, one clock to write). The addresses of the double clocked memory corresponded to the squares on the grid, and the values inside corresponded to their statuses. For the simplicity of our game, we used the following system: 0 = no ship; 1 = ship/part of ship; 2 = bombed this square, but contained no ship; 3 = bombed this square with a ship/part of ship present. We were constantly reading each address in sync with the VGA clock, so the display would always be updating. To keep track of the stages of the game, a simple register that incremented up by 1 (that was triggered by a stage change signal from the VGA) was implemented.

For the first phase of the game when the player's are setting ships, we pass a "place" input, a "size of ship" input (0 = length 1 ship, 1 = length 3 ship, 2 = length 5 ship), a

“rotate” input (0 = horizontal alignment, 1 = vertical alignment), and a “boxSelected” input into our processor. These inputs are processed by our processor to select the respective addresses within our dmem that correspond to the grid locations on the player’s board, and changes the value within that address. At this phase, whenever we placed a ship, the addresses corresponding to the ships would change from a 0 to 1, indicating a ship was placed here. This change would be reflected on the display, with each player able to see their own ship placements.

In the second phase, the boards visuals would be reset so that all ships are hidden, but the ship locations are still stored in our memory. Now, the players alternate bombing each other’s board. When a player selects to bomb a location, the processor will read the status at that location’s address and then update the status accordingly. It will change from 0 to 2 if there is no ship and 1 to 3 if there is a ship there. The VGA display will then be updated accordingly to this change. When one player’s ships have all been eradicated, the game is then concluded.

<u>I/O:</u>

1) 4 pushbuttons from the FPGA
    a) These are used to represent the up, down, left, right movement keys

2) Slide Switches from the FPGA
    a) Place ship
    b) Bomb ship
    c) Rotate/Orientation of ship
    d) Change stages of game

<u>Specifications:</u>

- Runs at 10 MHz clock
- 2 Doubly Clocked Memory Units
    - Read at VGA clock
    - Write at FPGA clock
- Unpipelined processor with fetch stage input from the I/O to decide next instruction to run in the imem.

<u>Processor Changes:</u>

There were not any “huge” changes to our processor since we would not be needing any custom instructions. The first thing was changing how our store word instruction

was implemented. Whenever we placed or bombed a ship, we would need to ensure that the correct value is written into the address(es) that needed to be affected. As a part of pre-processed assembly, we stored the value of 1 into register $1, 2 into register $2, and $3 into register $3. After combinational logic that would pick the correct number to write, we would jump to the respective part of our assembly code that would store those respective values into our memory. The little tweak that we made was using the immediate to modify the address / box that we were writing to. For example, if we wanted to place a vertical ship of length three in position 14. The instructions would be the following (with noops in between):

```
sw $1, 0(0)
sw $1, -12(0)
sw $1, 12(0)
```

This would ensure that when we are going through this part of the assembly code, we are writing the value of 1 into our current box, current box - 12, and current box + 12 to ensure proper placing.

Challenges:

Some of the biggest challenges we had were with timing. At first, with our push buttons, we needed to ensure that when pressed, it would only cause our cursor to move one square at a time. This was done by using the “always @(posedge (button))” command. This worked for one button, but later found out that if we did this 4 times and for 4 different buttons and tried changing our x position and y position, we were faced with having multiple drivers error since we were attempting to change our positioning with multiple clauses. Luckily, after some trial and error, we were able to fix this by using extra register variables that only hold the value of 1 for one clock cycle each time a button is pressed.

Next was writing into our doubly clocked memory. We have two variables: (1) pixelBox and (2) boxSelect. PixelBox is a value received from the VGA, and this essentially runs through all the addresses from 0 to 95 at the frequency of the VGA clock. This is used to update our screen. BoxSelect is the current box that is selected with the cursor on the screen. At first, we were wondering why when we bombed, it was sometimes the wrong value. We realized that this was because pixelBox was always changing, so sometimes BoxSelect and PixelBox were two different values and thus we would be writing the wrong value into the box we selected. This was solved however, by adding a 2nd doubly clocked memory that always read from boxSelect. So essentially, our original

doubly clocked memory would be used to continuously update the display and the second one would update the value of BoxSelect based on the current value of boxSelect. Since we were writing the same value into both but just reading different values, there were no discrepancies between the values within those two different doubly clocked memory units. In addition to this, we needed to slow down our clock because the clock speed was too quick when we were writing with a bomb command, which required some combinational logic in the processor.

The timing was once again prevalent when implementing a counter for the game stages. Originally, we would have just liked a single switch that can alternate between game stages. This proved to not work as we put the counter into our skeleton, and found out that even though the “clock” of the counter would be a slide switch, the FPGA clock speed caused us to get some very strange values. Slowing the clock down helped, but was overall still erroneous. Thus, our last resort was to just use slide switches as logical highs and lows in binary to represent the stages of our game.

Testing:

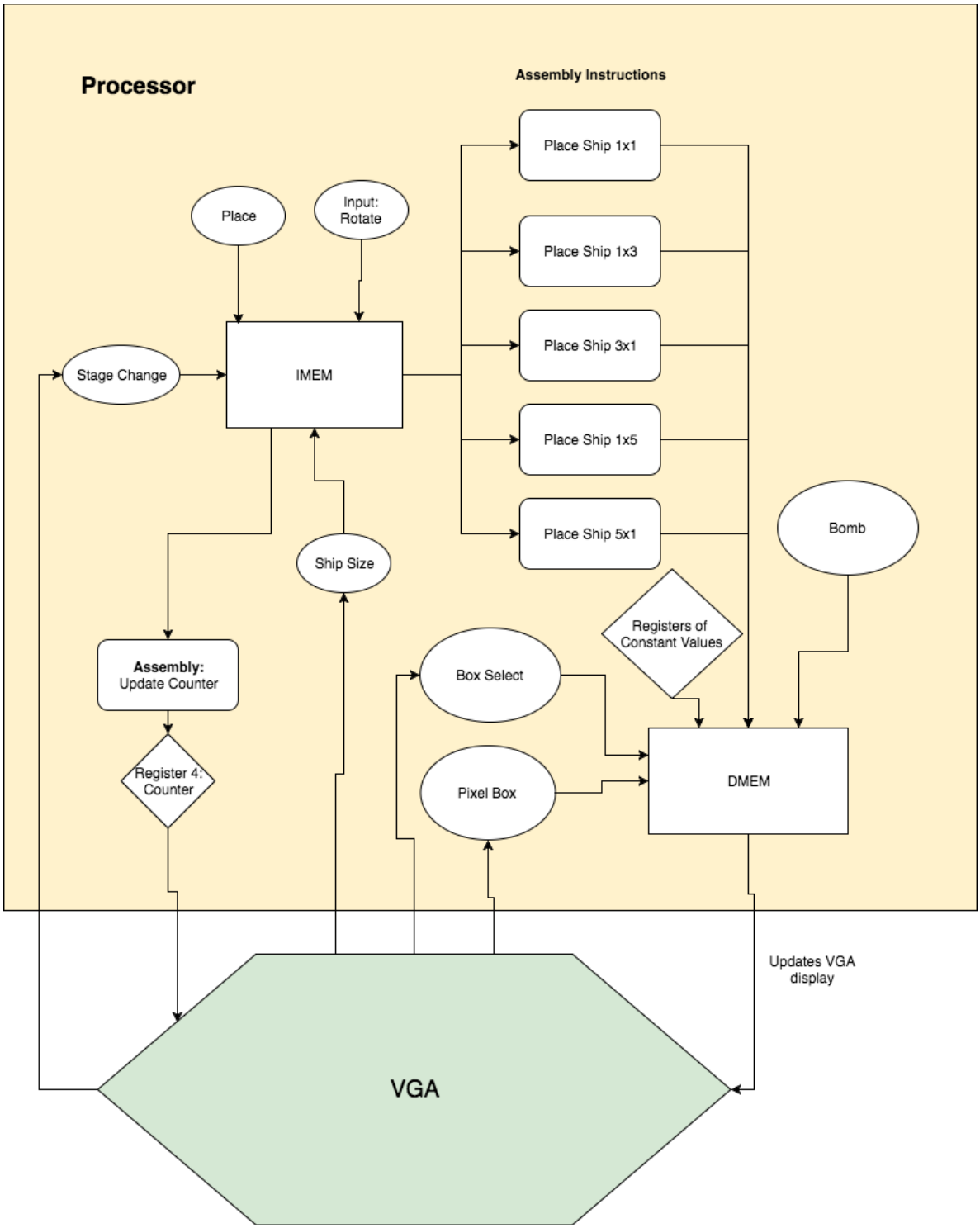
In regards to testing, a lot of it was just blasting it onto the FPGA board and testing if any of our changes worked. When we were working on the processor, we did use testbenches to test the response to various inputs before merging it with the VGA module in our head skeleton. The seven segment displays and LED’s were also very essential to help debug. We used the seven segment display the box number that our cursor was currently in and also for when we jumped addresses to ensure we were jumping to the correct one. We also used the LED’s to display the current state of the ships when we were on that square.

Assembly Program:

In response to our processor changes above, our assembly program was not terribly complicated. We did the majority of our visuals on hardware so that was handled by the VGA. We did attempt to use memory for visuals, but found it slightly difficult with clock timing issues and decided to use the memory that we had for something else. That being said, the main instructions that we used were add immediate to get the values into $1, $2, $3, and our counter into register $4; and a modified store word that changed the values at certain addresses with the help of an immediate. Instead of adding the immediate to the $rs, we just directly added it to our boxSelected since that was input directly passed to the processor from the VGA.

Our assembly code is attached separately.

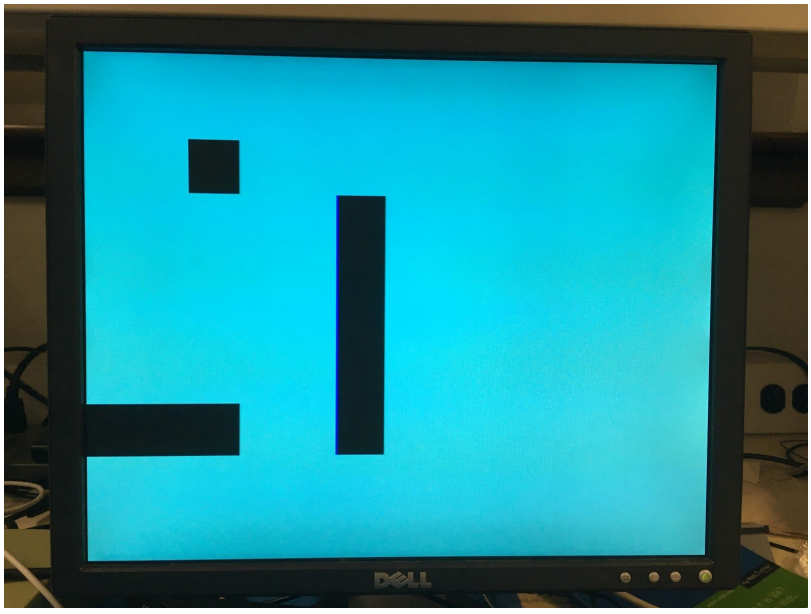
Flowchart of Assembly Code:



Pictures of Project:

Splash Screen:

Placement of Ships for Player 1:



Gameplay:

Yellow = miss
Red = Hit