VGA Battleship

Kyle Baran and Tae Lee
Engineering 155: Microprocessor-Based Systems
12/11/09

Abstract

In this project the team recreated the popular game "Battleship" using a keypad, pushbutton, PIC microcontroller, Xilinx FPGA, and VGA monitor. Two players first place their five ships on their board and then alternate turns guessing where their opponents' ships are. Both of these functions use a keypad polled by the PIC. The PIC keeps track of the ships' locations and their damage statuses and outputs information about the current guess to the FPGA in parallel. The FPGA is responsible for drawing on the VGA the game boards, showing which guesses were hits and misses, indicating which player is up, and displaying which ships have been destroyed. The game ends when one player sinks all of the other player's ships, and the FPGA indicates this by flashing which player has won.

Introduction

A traditional game of Battleship involves two players taking turns to guess the location of each other's ships on a 10x10 grid. Each player is given two such 10x10 boards and five different ships. One board is used to place his/her own ships and the other is used to mark the player's guesses of the opponent's ships. Due to the nature of the game, the players are not allowed to view each other's boards.

The game starts with the players' placing their ships such that no ships overlap or are off the board. Once all the ships have been placed, the players alternate making guesses about the location of the opponent's ships. Each ship has a specific length; the ships' names and their corresponding lengths are shown in ***Table 1***. Players mark an incorrect guess (miss) with a white pin and a correct guess (hit) with a red pin. A ship is sunk when every point along its length has been hit, and the player who lost his or her ship must say which ship was sunk. The game is over when all of one player's ships have been sunk, and the player with surviving ship(s) win.

Ship Name	**Length**
Destroyer	2
Submarine	3
Cruiser	3
Battleship	4
Aircraft Carrier	5

Table 1: Ship Name and Length

In this recreation of the game, a VGA screen is used to display the players' boards side by side, and each player is given only one board to both place the ships and hit opponent's ships. Since all that is really needed is that each player's guesses be indicated correctly (hit or miss), the players' ships placements are saved in a PIC microcontroller and are never shown on the screen. This may incur a little confusion during the placement stage, since the player cannot see the ships that he/she has placed, but it does not affect the overall game play.

The system takes input from a player via three main subsystems: a 4x4 keypad, a pushbutton, and a dip switch. The numbers from 0-9 are used to input a placement or a guess, depending on the stage of the game. The pushbutton is used to place a ship or guess a spot that's been input through the keypad. The DIP switch, which is used only during the ship placement stage, indicates the orientation (right or down) of the ship's placement. All of these buttons are polled by the PIC microcontroller, which stores the ships' placements and runs necessary game logic for the game play. The PIC outputs signals about the game conditions to the FPGA, which in turn outputs signals to run the VGA display. ***Figure 1*** shows basic a basic block diagram of this configuration.

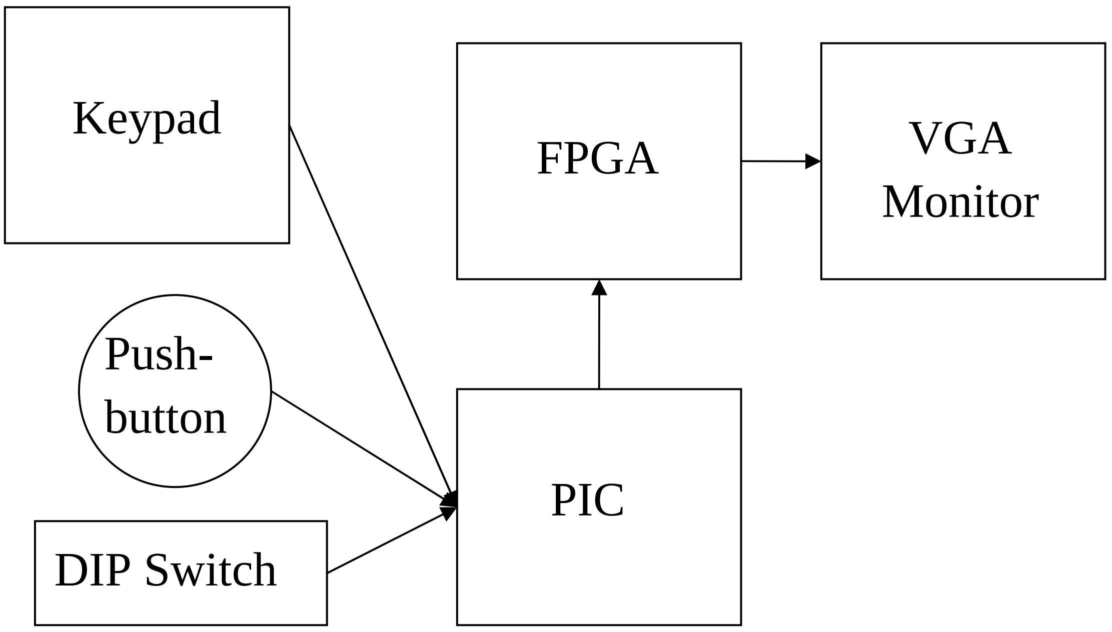
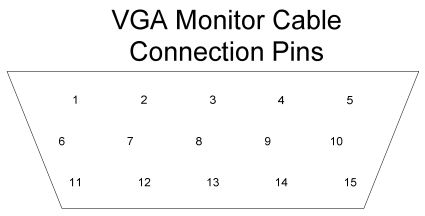


Figure 1: High-Level Block Diagram of Subsystems

Extra Hardware: VGA Display

A VGA monitor was used to display the game in this project. A VGA monitor uses a 15-pin connection to receive signals to display on its screen. There are three analog color pins, one for each of the colors red, green, and blue. Although these pins are analog, for the purpose of this project they only use discrete values since only 5 colors are employed. There are 4 ground pins, one for each color and another for master ground. Furthermore, two more pins are used for horizontal and vertical sync signals. The rest of the pins are unused. All the pins were tied to discrete outputs from the FPGA. ***Figure 2*** shows a diagram of a typical VGA connection with pins and their usage.



VGA Adapter	Sync
1. Signal R	11. NC
2. Signal G	12. NC
3. Signal B	13. HSync
4. NC	14. VSync
5. NC	15. NC
6. GND R	
7. GND G	
8. GND B	
9. No pin	
10. GND	

Figure 2: VGA Connections

A standard 640x480 VGA screen takes in pixel data at 25.175MHz. The FPGA's 40MHz internal clock is scaled by 5/8 down to 25 MHz using a Digital Clock Manager (DCM) built into the FPGA. The FPGA outputs pixel data at 25MHz, which is within the acceptable range of data rate of a VGA monitor. HSync and VSync signals are used to tell the monitor when to begin a row of pixels and when to begin a new screen. Each time HSync pulses, a row of the screen is drawn. VSync is high for the entire time the screen is drawn, which is refreshed at approximately 60Hz. ***Table 2*** summarizes the VGA timing.

	25.175 MHz clock (VGA Standard)
[clock cycles] time	
Hsync period	[800] 31.778 us
H front porch	[8] 317.775 ns
Hsync pulse length	[96] 3.813 us
H back porch	[40] 1.589 us
H Border	[8] 317.775 ns
H active video	[640] 25.422 us
[scans] time	
Vsync period	[525] 16.683 ms
V front porch	[2] 63.555 us
Vsync pulse length	[2] 63.555 us
V back porch	[25] 794.439 us
V Border	[8] 254.220 us
V active video	[480] 15.253 ms
Vsync, Hsync polarity	-,-
H frequency	31.47 KHz
V frequency	59.94 Hz

Table 2: VGA Timing Information

The MicroToys VGA monitor documentation contains more information about VGA monitor with FPGA controls.[1]

Schematics

Figure 3 shows the breadboard schematic of our design.

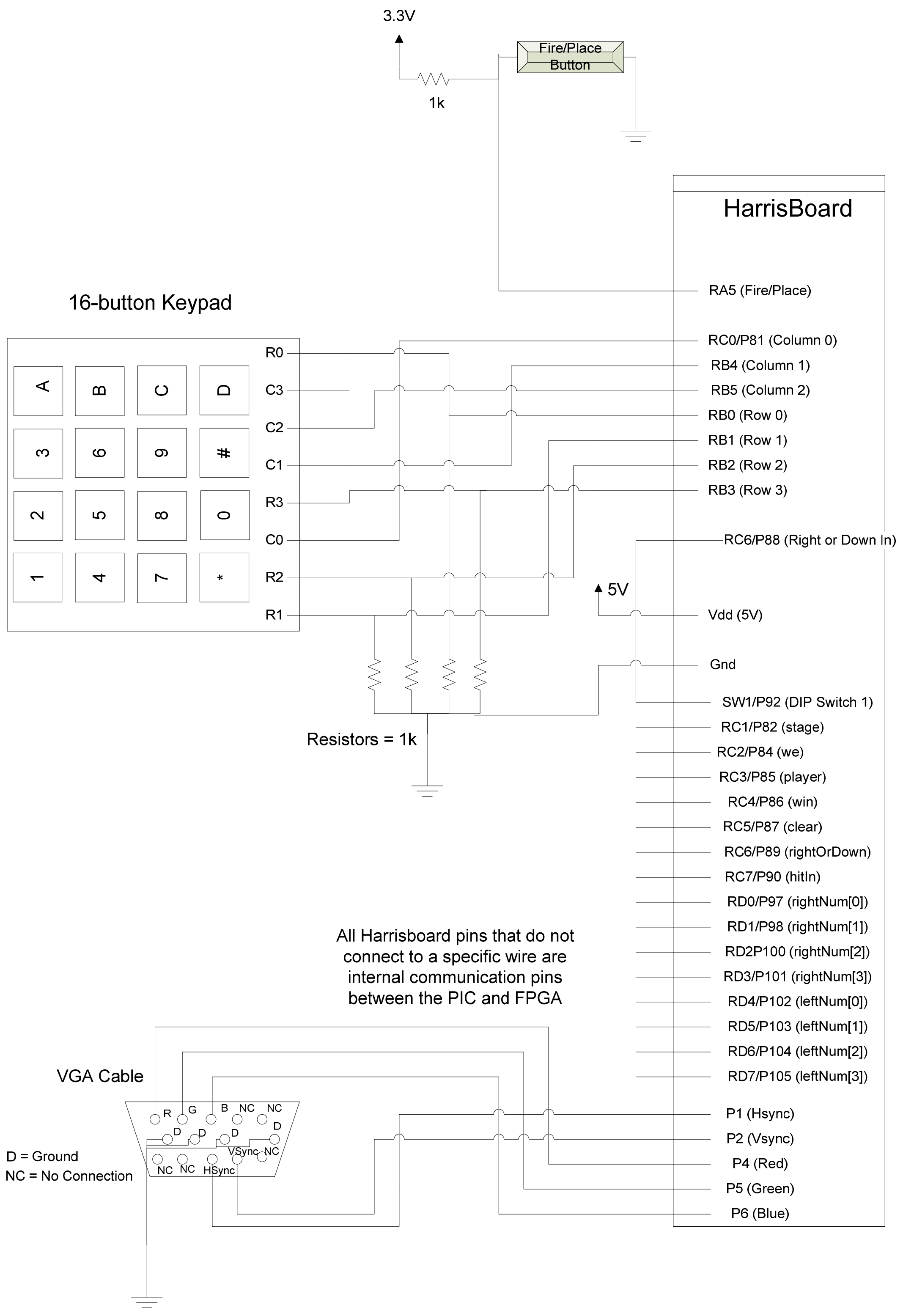
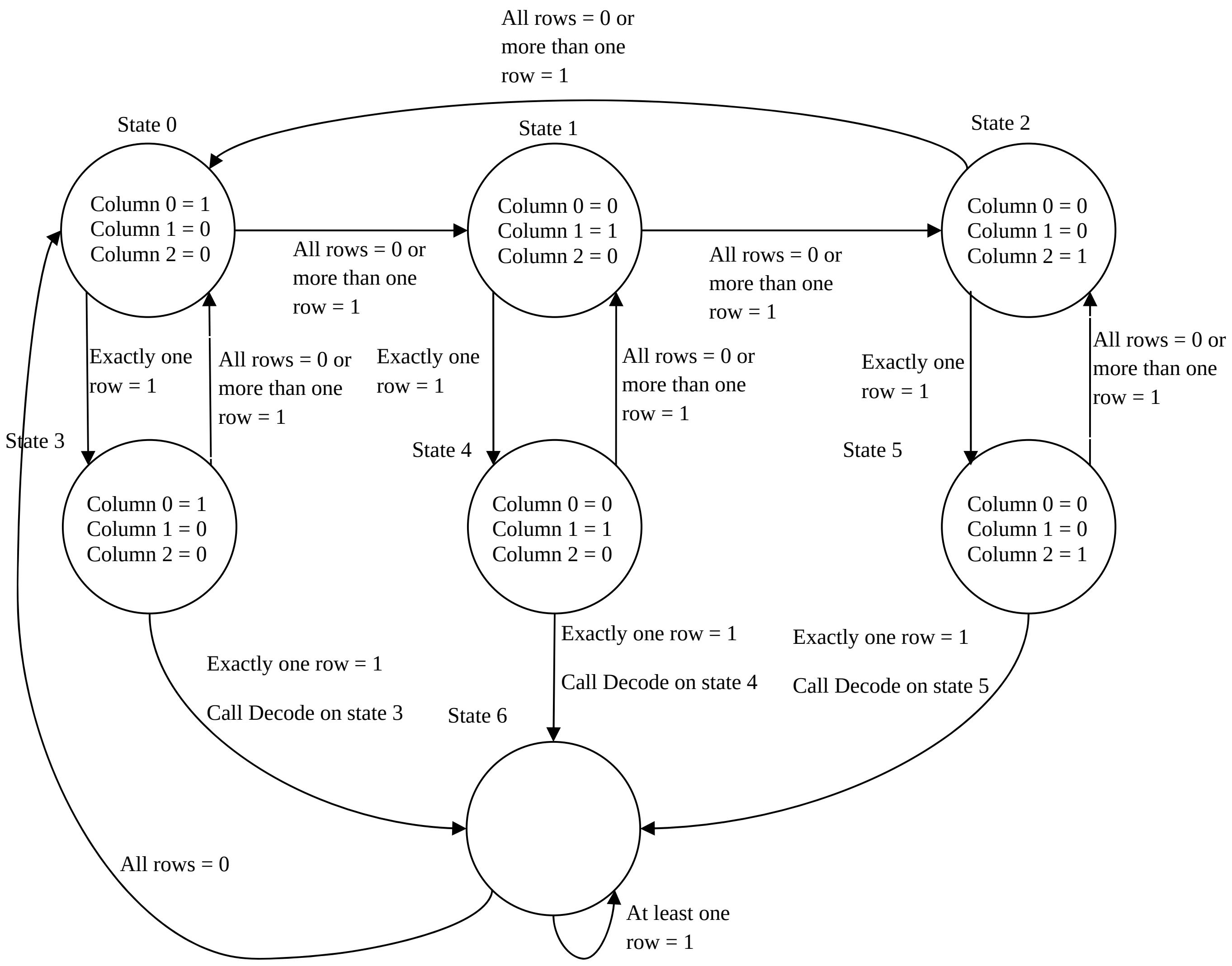


Figure 3: Breadboard Schematic

Microcontroller Design

Input Functionality

All of the PIC's input functionality is governed by a finite state machine as shown in ***Figure 4***.



All rows = 0 or more than one row = 1
State 0
State 1
State 2
Column 0 = 1 Column 1 = 0 Column 2 = 0
Column 0 = 0 Column 1 = 1 Column 2 = 0
Column 0 = 0 Column 1 = 0 Column 2 = 1
All rows = 0 or more than one row = 1
All rows = 0 or more than one row = 1
Exactly one row = 1
All rows = 0 or more than one row = 1
Exactly one row = 1
All rows = 0 or more than one row = 1
Exactly one row = 1
All rows = 0 or more than one row = 1
State 3
State 4
State 5
Column 0 = 1 Column 1 = 0 Column 2 = 0
Column 0 = 0 Column 1 = 1 Column 2 = 0
Column 0 = 0 Column 1 = 0 Column 2 = 1
Exactly one row = 1
Call Decode on state 3
Exactly one row = 1
Call Decode on state 4
Exactly one row = 1
Call Decode on state 5
State 6
All rows = 0
At least one row = 1

Figure 4: Diagram of keypad polling Finite State Machine

The PIC accepts input from a 16-segment keypad (shown in ***Figure 3***) through polling. The PIC drives the voltage of each column of the keypad high one at a time and checks to see if any of the buttons have been pressed, which is indicated by one or more of the rows having a high voltage. The PIC goes to the next state every 13.1 ms to ignore bounce from the keypad. The columns are connected to PORTB<5:4> and PORTC<0> while the rows are connected to PORTB<3:0>. If exactly one of the rows is high, and stays that way for another 13 ms, then only one button has been pressed, and the PIC records which button has been pressed based on the high row and column information. Since the standard Battleship board is a 10x10 grid, all 16 keys are not necessary, so the rightmost column, which contains the letters A through D, is not polled at all.

The star and pound keys are ignored by the decoding logic, meaning that only pressing the 0 through 9 buttons will be accepted as a valid entry by the PIC.

The PIC stores the potential guesses as a two-digit number, with the left digit representing the row to be guessed and the right number representing the column. The PIC continually outputs the row/column information over PORTD to the FPGA, with PORTD<7:4> holding the row value and PORTD<3:0> holding the column value. The rows and columns are 0-indexed, with the top left square having a location of (0,0) and the bottom right square having a location of (9,9). Each value is initialized as 10, which the game logic reads as an invalid value. When a valid button is pressed, the PIC shifts the guess digits left, so the old column value becomes the new row value and the most recent button press becomes the new column value. The user can press the keypad as many times as he or she likes without the PIC registering a guess, since a separate press of a pushbutton (hereafter referred to as the "fire" button) is necessary for the current number to be entered as a guess. A guess will only be registered as valid if both the digits are not 10.

Miscellaneous Settings

The PIC uses Timer0 to control all time-based aspects of the game. INCTON is set to 0xA0 so that interrupts are enabled and the Timer0 overflow is enabled. T0CON is set to 0x81 so that Timr0 is enabled and has a prescalar of 2; this prescalar means that Timer0 increments once every 0.2 μs and overflows every 13.1 ms. ADCON1 is set to 0x0F so that PORTB can function as digital I/O instead of analog output.

Placement Stage

The ships are initialized as arrays of chars with lengths matching the physical length of each ship. Each ship has an array for the row locations, the column locations, and the damage status. The row and column arrays are not set to any values since those will be determined by the user. The damage array is initialized to all zeroes, which is read by the game logic as not hit.

Upon resetting the PIC, the game begins in placement stage. The keypad is used to select a square where a ship will start, and DIP switch 1 controls whether the rest of the ship will be placed to the right of that square or down from that square. Which ship is currently being placed is communicated to the FPGA by setting the overall value of PORTE; a PORTE<2:0> value of 1 means the destroyer is due to be placed, 2 the submarine, 3 the cruiser, 4 the battleship, and 5 the aircraft carrier. The ship orientation is given to the FPGA over PORTC<6>, with 0 indicating "right" and 1 indicating "down." Once the "fire" button is pressed with valid values for the row and column digits, the game attempts to place the current ship.

If the player tries to place a ship such that part of it would either be off the game board or overlapping an already-placed ship, the PIC does not save the location, resets the row/column values, and waits for the user to make a new attempt. Once a valid placement is given, the PIC stores the ship's location and tells the FPGA to display the next ship by changing the value of PORTE. The ships are always placed in the order of destroyer, submarine, cruiser, battleship, and aircraft carrier. Player 1 places all of his or her ships in succession, followed by player 2 doing the same. The PIC continually tells the FPGA which player is up by controlling the value of PORTC<3>; a value of 0 indicates that Player 1 is up while a value of 1 indicates that Player 2 is up.

Guess Stage

Once all ten ships have been placed, the game enters guess stage. The players take turns guessing a square to see if one of their opponent's ships is at that location by entering a row/column guess and pressing the “fire” button. The PIC checks to see if that row and column match any of the other player’s ships’ locations, and if so, sets that ship’s damage status to 1 at the index of the current guess, i.e. if (1,5) occurred at row[2] and colum[2] for a battleship, then damage[2] would be set to 1. If the guess is a hit, the PIC tells the FPGA that information by setting PORTC<7> high; otherwise, it remains low, indicating a miss. If the most recent guess caused a ship to be sunk, the PIC tells the FPGA that information by setting PORTE to something nonzero; the values and corresponding ship are given above in the “Placement Stage” section. The PIC concludes the hit logic by setting PORTC<2> high, telling the FPGA that the guess is official.

The PIC then pauses for approximately a quarter of a second to ensure smoothness of gameplay. After the pause is over, the PIC sets PORTC<2> low to inform the FPGA that the guess has been entered and a new turn is upcoming. If neither player has won yet, the PIC switches the value of PORTC<3> to change players, resets the row/column values and PORTD to (10, 10), and sets PORTE to 0.

Once all of the ships of one player are destroyed, the PIC sets PORTC<4> high, informing the FPGA that the game is over. The FPGA turns the player indicator green and begins to flash the player indicator and all the ship status indicators on the winning player’s board. The PIC and FPGA enter infinite loops and must be manually reset to play again.

Output Functionality

The FPGA continually draws the game boards, player indicator, and ship statuses on the VGA monitor. The game boards are 10x10 grids with each square having 30 pixel sides. The player indicator is 300 pixels wide and 60 pixels tall and is drawn above the game board of the player who is currently up. The ship status indicators are small representations of the ships, one set for each player, that are drawn below the game boards and have lengths relative to the size of the represented ships.

The FPGA uses that information to highlight the proper square, showing the user where they would be guessing if he or she pressed the “fire” button. The FPGA does not write any data unless PORTC<2> is driven high by the FPGA, which indicates that a valid guess has been entered. After each guess the FPGA resets the highlighted square to the location (0,0), although the row/column information is reset to its original invalid state, and two buttons must be pressed before a valid guess can be made.

FPGA Design

The FPGA is used to generate the necessary signals to drive a VGA display. The Verilog code from the MicroToys VGA project [1] was heavily modified and augmented to fulfill the requirements of this project’s specific game play. The FPGA takes a 40 MHz internal clock and 16 bits of parallel inputs from the PIC microcontroller and generates HSync, VSync, and the color (RGB) signals.

Digital Clock Manager

The first thing the FPGA does is set up a new clock at 25 MHz to be able to drive the VGA. This is done using a built-in Digital Clock Manager. Using Xilinx tools CoreGen and Architecture Wizard’s Single DCM option, the FPGA scales the internal clock down by 5/8 to

25MHz. In order to set the scaling factor to 5/8, CLKDV and CLKFX has to be set to 5 and 8 respectively. Then a module with output 25MHz sclk is instantiated for use.

HSync and VSync Generation

Module *GenSyncsFullVGA* was written to generate horizontal and vertical sync signals that are used by the VGA to scan effectively through the display. The module takes in the 25MHz clock and employs several counters to generate these signals. It also takes into account the front and back porches of each signal. Note that the porch timing and the wire names are slightly modified from the original MicroToys code, but only for the sake of the writer of the code. Nothing is effectively changed. This module also outputs a *DataValid* bit that is asserted when the pixel data is valid.

Board Display

A major part of the FPGA's job is to draw the game boards on the screen. This proved tricky, however, since in this project the PIC does not constantly update the FPGA with locations of everything that needs to be drawn on the board. Because the PIC outputs the current grid number that has been pressed and updates hit or miss only after the fire button is pressed, it is the FPGA's job to save this information to display it on the screen. In order to do this, module *GenBlockNum*, which was heavily modified from the MicroToys project, was written. This module uses *HSync*, *VSync*, and *DataValid* from *GenSyncsFullVGA* to index the rows and columns of every pixel on the 640x480 display. Furthermore, using the row and column indices, the board is divided into two adjacent 10x10 grids (used as boards), where each block in the grid is also index. The left grid (player one's board) is indexed 0-99, and the right grid is index 100-199. These are called block numbers, and they are effectively encoded addresses for the RAM modules.

When a player pushes the keypad to choose a square to guess, this information is sent to the FPGA, which uses a module called *getBlockNumWAddr* to encode these numbers to write addresses for the board information RAMs. When the player finally fires, the FPGA gets the write address and the hit bit from the PIC and stores this information in the RAM. Then when the display is scanned using *HSync* and *VSync*, the *GenSignals* module generates appropriate color signals for each pixel within the boards based on the contents of the RAM addressed by the block number.

Other Game Information Display

GenSignals module also outputs other necessary game information to make the game more playable. It indicates the block that is currently being guessed (before fire button is pressed). It produces signals to draw a white bar above the board that is in play. It also calls other modules to draw the five ships beneath each board, and color them red or green to indicate if they have been sunk or not. This is done by employing similar RAM logic as is done when displaying the boards. It calls *inWhichShip* module to determine where on the board the pixel is being currently drawn and colors the ship according to RAM data that's been written with signals from the PIC. It also uses *InBox* module that was directly taken from the MicroToys project to set boundary conditions for each ship displayed. During the placement stage, the ships are displayed one at a time to show which ship the player is placing on the board.

Win

When a player wins, a 1Hz pulse is generated using the *counter* module. It is used to flash the sunk ships and the white bar of the player's board to indicate that the player has won and sunk all of the opponent's ships.

Results

The team created a fully functional game that met or exceeded all of the original goals. In placement stage it is impossible to place a ship in an invalid or already-occupied location, and no mistakes are made in storing the ship's location to memory. Guess stage works perfectly, always recording the correct hit logic and outputting the correct information to the FPGA. The FPGA draws the board almost perfectly, with the only mistakes occasional faint random lines on the game boards.

The most difficult part of the PIC design was the ship placement. It took a good deal of thinking to figure out the best way to check the current ship's potential position against the positions of all of the previously placed ships. On the other hand, the most difficult part of the FPGA design was in implementing the RAM logic to draw the board. Examples from previous projects had the PIC updating the FPGA the locations of everything in real time. In this project we needed a way to store the information on the FPGA. It took a long amount of time to decide on using RAMs and actually implement it by indexing the blocks within the boards on the screen to use as addresses.

A few aspects of the game are different from the original proposal but no less effective. The proposal called for the ships' locations to be fixed, but the placement functionality was added to make the game more involving and enables players to change the ships' locations without changing code and reprogramming the PIC. This led to added functionality and logic for both the PIC and FPGA, as the placement requires somewhat different procedures from the normal game stage. The proposal also stated that the VGA would be drawing the row and column digits on the screen, but the team decided to highlight the indicated square on the board instead. The team feels that this makes it easier to determine where the guess will be since the player can see exactly where that square is in relation to where he or she has already guessed.

Reference

[1] MicroToys VGA Project

http://www4.hmc.edu:8001/Engineering/microtoys/VGA/MicroToys%20VGA.pdf

Materials

The only materials used were the VGA monitor, keypad, and pushbutton. All of these were obtained from the MicroP's lab.

Appendix A: Verilog Code

battle_ship_vga.v

```
`timescale 1ns / 1ps
// 11-24-09
// Samuel Taeyoung Lee
// tlee@hmc.edu
// This module handles the inputs and outputs as well as calls the instances
of
//     modules that produce VGA signals

module battle_ship_vga( input clk, //40MHz clock
                        input             reset,
                        input             we,
                        input             player,
                        input             hitIn,
                        input             stage,
                        input             rightOrDown,
                        input             win,
                        input  [2:0] shipSunkIn,
                        input  [3:0] rightNum,
                        input  [3:0] leftNum,
                        output            HSync, VSync,
                        output [2:0] Signal);

      wire  clkdv, clkfx, clklock, sclk; //sclk = 25MHz
      wire  DataValid;
      wire  [7:0] blockNumWAddr;

      // Use DCM to create a 25MHz clock
      DCMclk dcm1(clk, reset, clkdv, clkfx, sclk, clklock);

      getBlockNumWAddr getInputBlockNum(clk, rightNum, leftNum,
                                        blockNumWAddr);

      // Generate monitor timing signals
      GenSyncsFullVGA GenSyncs(sclk, reset, DataValid, HSync, VSync);

      counter getSecond(HSync, second);

      // Generate signal to monitor
      GenSignals GenSignal( sclk, VSync, DataValid, we, player, hitIn,
                            blockNumWAddr, shipSunkIn, stage, rightOrDown,
                            win, second, Signal);

endmodule
```

DCMclk.v

```
////////////////////////////////////////////////////////////////////////////////
///
// Copyright (c) 1995-2008 Xilinx, Inc.  All rights reserved.
////////////////////////////////////////////////////////////////////////////////
///
//   ____  ____
//  /   /\/   /
// /___/  \  /    Vendor: Xilinx
// \   \   \/     Version : 10.1.03
//  \   \         Application : xaw2verilog
//  /   /         Filename : DCMclk.v
// /___/   /\     Timestamp : 12/06/2009 23:05:46
// \   \  /  \
//  \___\/\___\
//
//Command: xaw2verilog -intstyle C:/Battle Ship/battleship/DCMclk.xaw -st
DCMclk.v
//Design Name: DCMclk
//Device: xc3s400-4tq144
//
// Module DCMclk
// Generated by Xilinx Architecture Wizard
// Written for synthesis tool: SynplifyPro
// Period Jitter (unit interval) for block DCM_INST = 0.03 UI
// Period Jitter (Peak-to-Peak) for block DCM_INST = 1.23 ns
`timescale 1ns / 1ps

module DCMclk(CLKIN_IN,
              RST_IN,
              CLKDV_OUT,
              CLKFX_OUT,
              CLK0_OUT,
              LOCKED_OUT);

   input CLKIN_IN;
   input RST_IN;
   output CLKDV_OUT;
   output CLKFX_OUT;
   output CLK0_OUT;
   output LOCKED_OUT;

   wire CLKDV_BUF;
   wire CLKFB_IN;
   wire CLKFX_BUF;
   wire CLK0_BUF;
   wire GND_BIT;

   assign GND_BIT = 0;
   assign CLK0_OUT = CLKFB_IN;
   BUFG CLKDV_BUFG_INST (.I(CLKDV_BUF),
                        .O(CLKDV_OUT));
   BUFG CLKFX_BUFG_INST (.I(CLKFX_BUF),
                        .O(CLKFX_OUT));
   BUFG CLK0_BUFG_INST (.I(CLK0_BUF),
                        .O(CLKFB_IN));
   DCM DCM_INST (.CLKFB(CLKFB_IN),
                 .CLKIN(CLKIN_IN),
                 .DSSEN(GND_BIT),
                 .PSCLK(GND_BIT),
                 .PSEN(GND_BIT),
                 .PSINCDEC(GND_BIT),
                 .RST(RST_IN),
                 .CLKDV(CLKDV_BUF),
                 .CLKFX(CLKFX_BUF),
                 .CLKFX180(),
                 .CLK0(CLK0_BUF),
                 .CLK2X(),
                 .CLK2X180(),
                 .CLK90(),
                 .CLK180(),
                 .CLK270(),
                 .LOCKED(LOCKED_OUT),
                 .PSDONE(),
                 .STATUS());
   defparam DCM_INST.CLK_FEEDBACK = "1X";
   defparam DCM_INST.CLKDV_DIVIDE = 2.0;
   defparam DCM_INST.CLKFX_DIVIDE = 8;
   defparam DCM_INST.CLKFX_MULTIPLY = 5;
   defparam DCM_INST.CLKIN_DIVIDE_BY_2 = "FALSE";
   defparam DCM_INST.CLKIN_PERIOD = 25.000;
   defparam DCM_INST.CLKOUT_PHASE_SHIFT = "NONE";
   defparam DCM_INST.DESKEW_ADJUST = "SYSTEM_SYNCHRONOUS";
   defparam DCM_INST.DFS_FREQUENCY_MODE = "LOW";
   defparam DCM_INST.DLL_FREQUENCY_MODE = "LOW";
   defparam DCM_INST.DUTY_CYCLE_CORRECTION = "TRUE";
   defparam DCM_INST.FACTORY_JF = 16'h8080;
   defparam DCM_INST.PHASE_SHIFT = 0;
   defparam DCM_INST.STARTUP_WAIT = "FALSE";
endmodule
```

sumbmodules.v

GenSyncsFullVGA

```
`timescale 1ns / 1ps
// 11-24-09
//Originally written by Michael Cope & Philip Johnson 1999
//modified by Dan Chan, Nate Pinckney & Dan Rinzler Spring 2005
//Further modified by Samuel Taeyoung Lee Fall 2009 (tlee@hmc.edu)
//This module takes the 25Mhz clock and steps it down to turn on
//HSync and VSync at the correct frequencies. It also determines when
//it is possible to send data for each pixel.
module GenSyncsFullVGA( input      clk,
                        input      reset,
                        output     DataValid,
                        output reg HSync, VSync);

      // DataValid is high when data is ready to flow
      //Hsync = 31470Hz Vsync = 59.94Hz
      // counters to slow down for HSync and VSync
```

```
    reg [9:0] Hcount, Vcount;
    reg          HData;           // high when HSync data is ready to flow
    reg          VData;           // high when VSync data is ready to flow

    always @ (posedge clk)
          begin
                Hcount <= Hcount + 1;
                if ((Hcount == 10'd800) || (reset == 1))
                      Hcount <= 0;
                if ((Hcount >= 10'd0) && (Hcount < 10'd96))
                      HSync <= 0;
                else
                      HSync <= 1;
                if ((Hcount >= 10'd144) && (Hcount < 10'd784))
                      HData <= 1;
                else
                      HData <= 0;
          end

    //this always block determines when VSync should be driven low,
    //indicating the start of a new screen
    always @ (negedge HSync)
          begin
                Vcount <= Vcount + 1;
                if ((Vcount == 10'd525) || (reset == 1))
                      Vcount <= 0;
                if ((Vcount >= 10'd0) && (Vcount < 10'd2))
                      VSync <= 0;
                else
                      VSync <= 1;
                if ((Vcount >= 10'd35) && (Vcount < 10'd515))
                      VData <= 1;
                else
                      VData <= 0;
          end

    assign DataValid = HData && VData;
endmodule
```

InBox

```
// Samuel Lee (tlee@hmc.edu)
// Code taken from... 2006 MicroP's final project Pong
// Based on code by Michael Cope and Philip Johnson 1999
// Modified by Dan Chan, Nate Pinckney and Dan Rinzler Spring 2005
// Further modified by Jonathan Beall and Austin Katzin, Fall 2006
// Given coordinates, the upper left corner of the box, and the size,
// are the coordinates in the box?
module InBox(x, y, x1, y1, width, height, in);
    input [9:0] x, y;
    input [9:0] x1, y1;
    input [9:0] width, height;
    output in;

    wire [9:0] x2, y2;
    assign x2 = x1 + width;
    assign y2 = y1 + height;

    assign in = x >= x1 && x < x2 && y >= y1 && y < y2;
endmodule
```

counter

```
// Samuel Taeyoung Lee Fall 2009 (tlee@hmc.edu)
// This module
// counts using HSync and outputs a signal second
// that flashes every second
module counter( input      HSync,
                output reg second);

      reg [14:0] count;

      always @ (posedge HSync)
            if (count == 15'd31470)
                  begin
                        count <= 0;
                        second <= ~second;
                  end
            else
                  count <= count+15'd1;

endmodule
```

mux2_10bit

```
// Samuel Taeyoung Lee Fall 2009 (tlee@hmc.edu)
a simple 2 10-bit input mux
module mux2_10bit( input  [9:0] d0, d1,
                   input        sel,
                   output [9:0] y);

      assign y = sel ? d1 : d0;

endmodule
```

getBlockNumWAddr

```
// Samuel Taeyoung Lee Fall 2009 (tlee@hmc.edu)
// This module gets playe's guessed rightNum and leftNum
// and produces blockNum for the write address of the blockNum RAM
module getBlockNumWAddr( input            clk,
                         input      [3:0] rightNum, leftNum,
                         output reg [7:0] blockNumWAddr);


      always @ ( posedge clk )
            if ((leftNum != 4'd10) && (rightNum != 4'd10))
                  blockNumWAddr <= {4'd0,leftNum}*10 + {4'd0,rightNum};
            else if ((leftNum == 4'd10) || (rightNum == 4'd10))
                  blockNumWAddr <= 8'd0;

endmodule
```

inWhichShip

```
//12-7-09
//Written by Samuel Taeyoung Lee Fall 2009 (tlee@hmc.edu)
//This module takes different in"ship" which means the coordinates
//of scanner are in the corresponding ship
//and outputs a decoded address for the ship for the RAM
module inWhichShip( input           inDest1,inSub1,inCruis1,inBatt1,inCarr1,
                    input           inDest2,inSub2,inCruis2,inBatt2,inCarr2,
                    output reg [3:0] ship);

      wire [9:0] ships;

      assign ships = {inCarr2, inBatt2, inCruis2, inSub2, inDest2,
                      inCarr1, inBatt1, inCruis1, inSub1, inDest1};

      always @ ( * )
            case (ships)
                  10'b00_0000_0001:       ship = 0;
                  10'b00_0000_0010:       ship = 1;
                  10'b00_0000_0100:       ship = 2;
                  10'b00_0000_1000:       ship = 3;
                  10'b00_0001_0000:       ship = 4;
                  10'b00_0010_0000:       ship = 5;
                  10'b00_0100_0000:       ship = 6;
                  10'b00_1000_0000:       ship = 7;
                  10'b01_0000_0000:       ship = 8;
                  10'b10_0000_0000:       ship = 9;
                  default:                            ship = 0;
            endcase

endmodule
```

GenBlockNum

```
//11-24-09
//Originally written by Michael Cope & Philip Johnson 1999
//modified by Dan Chan, Nate Pinckney & Dan Rinzler Spring 2005
//Further modified heavily by Samuel Tae Lee Fall 2009 (tlee@hmc.edu)
//This module takes the VSync and DataValid signals and uses them to
//index each block in two squares each composed of 10x10 smaller blocks
module GenBlockNum( input            clk,
                    input            VSync,
                    input            DataValid,
                    output           inBounds, inPlayer1, inPlayer2,
                    output     [7:0] blockNumRAddr,
                    output reg [9:0] row, col);

      // one block is 30x30 pixel
      // one large square is 300x300 pixel
      // left margin of the left square is 10
      // both top and bottom margins are 90 each
      parameter BLOCKSIZE = 30;
      parameter BOARDSIZE = BLOCKSIZE*10;
      parameter SIDEMARGIN = (320-BOARDSIZE)/2;
      parameter TOPMARGIN = (480-BOARDSIZE)/2;



      // in square 1, in square 2, below top margin and above bottom margin
      // column and row numbers as HSync and Vsync scans monitor
      // block column indices and block row indices
      wire inBounds1, inBounds2, inRowBounds;
      wire [7:0] blockNum1, blockNum2;
            // block indices
      reg [9:0] temp;

      reg [5:0] coltemp, rowtemp;
      reg [6:0] blockcol1, blockcol2, blockrow;


      // one condition to increase block column index
      assign increaseBlockCol = (rowtemp==0);

      //assign boundary conditions
      assign inBounds2 =((col>=320+SIDEMARGIN)&&
                        (col<320+SIDEMARGIN+BOARDSIZE)&&
                        (row>=TOPMARGIN)&&(row<TOPMARGIN+BOARDSIZE));

      assign inBounds1 = ((col>=SIDEMARGIN)&&(col<SIDEMARGIN+BOARDSIZE)
                         &&(row>=TOPMARGIN)&&(row<TOPMARGIN+BOARDSIZE));

      assign inBounds = inBounds1 || inBounds2;

      assign inRowBounds = ((row>=TOPMARGIN)&&(row<TOPMARGIN+BOARDSIZE));

      assign inPlayer1 = ((col>=SIDEMARGIN)&&(col<SIDEMARGIN+BOARDSIZE)
                         &&(row>=30)&&(row<80));

      assign inPlayer2 = ((col>=320+SIDEMARGIN)&&
                         (col<320+SIDEMARGIN+BOARDSIZE)
                          &&(row>=30)&&(row<80));



      //This always block assigns column values
      //from 0 to 640 as each different
      //pixel is displayed for a particular row
      //It also assigns column indices to each smaller block in the board
      always @ (posedge clk)
            if (DataValid)
                  begin
                        col <= col + 1;
                        if (coltemp == BLOCKSIZE-1)
                              begin
                                    coltemp <= 0;
                                    if (inBounds1)
                                          blockcol1 <= blockcol1 + 1;
                                    if (inBounds2)
                                          blockcol2 <= blockcol2 + 1;
                              end
                        else
                              if (inBounds)
                                    coltemp <= coltemp + 1;
                  end
            else
                  begin
                        col <= 0;
                        coltemp <=0;
                        blockcol1 <= 0;
                        blockcol2 <= 0;
                  end

      //Like col, temp counts the number of pixels across each row,
      //but we then use
      //temp to incriment a row counter
      //whenever temp = 'd640. This allows us to
      //reference the rows by a number from 0 to 480.
      // This always block also assigns block row indices
      always @ (posedge clk)
            begin
                  if (!VSync) begin
                        temp <= 0;
                        rowtemp <= 0;
                        row <= 0;
                        blockrow <= 0;
                        end
                  else begin
                        if (DataValid)
                              temp <= temp + 1;
                        if (temp == 10'b10_1000_0000) begin
                              row <= row + 1;
                              temp <= 0;
                              if (rowtemp == BLOCKSIZE-1) begin
                                    rowtemp <= 0;
                                    if (inRowBounds)
                                          blockrow <= blockrow + 1;
                                    end
                              else
                                    if (inRowBounds)
                                          rowtemp <= rowtemp + 1;
                                    end
                        end
            end


      assign blockNum1 = blockcol1 + 10*blockrow; // block indices for board1
      assign blockNum2 = blockcol2 + 10*blockrow; // block indices for board2

      // assign ouput blockNumRAddr according to
      // where the Sync is on the screen
      // offset blockNum2 by 100 to save on RAM
      assign blockNumRAddr = inBounds1 ? blockNum1 : blockNum2+8'd100;

endmodule
```

GenSignals

```
// 11-24-09
// Samuel Taeyoung Lee (tlee@hmc.edu)
// This module gets VSync, DataValid, we, player, hitIn,
// blockNumWAddr, shipIn, stage, rightOrDownIn, win, and second
// as well as call instances of other modules to create output signal to VGA
module GenSignals( input        clk,
                   input        VSync,
                   input        DataValid,
                   input        we,
                   input        player,
                   input        hitIn,
                   input  [7:0] blockNumWAddr,
                   input  [2:0] shipIn,
                   input        stage,
                   input        rightOrDownIn,
                   input        win,
                   input        second,
                   output [2:0] Signal);


      wire [1:0] hitInTemp, RAMHitIn, status;
      wire [7:0] blockNumRAddr;
      wire [7:0] RAMBlockNumIn;
      wire sunkIn, sunkOut;
      wire inDest1, inSubm1, inCrui1, inBatt1, inCarr1;
      wire inDest2, inSubm2, inCrui2, inBatt2, inCarr2;
      wire [9:0] row, col;
      wire [3:0] ship;
      wire [3:0] shipSunkWAddr;
      wire beingGuessed;

      wire [9:0] destDim1, submDim1, cruiDim1, battDim1, carrDim1;
      wire [9:0] destDim2, submDim2, cruiDim2, battDim2, carrDim2;
      wire [9:0] destLocX1, destLocX2, destLocY;
      wire [9:0] submLocX1, cruiLocX1, battLocX1, carrLocX1;
      wire [9:0] submLocX2, cruiLocX2, battLocX2, carrLocX2;
      wire [9:0] submLocY, cruiLocY, battLocY, carrLocY;
      wire destChoose1, submChoose1, cruiChoose1, battChoose1, carrChoose1;
      wire destChoose2, submChoose2, cruiChoose2, battChoose2, carrChoose2;
      wire rightOrDown;
      wire flash;




      // logic for placing ships
      // stage is high when players place their ships

      assign rightOrDown = stage ? rightOrDownIn : 1'b0;
      assign destLocX1 = 10'd10;
      assign destLocX2 = 10'd330;
      assign destLocY = 10'd400;
```

```
// muxes for showing ships on VGA.
//different when player is placing ships.
mux2_10bit getDestDim1(10'd20, 10'd10, rightOrDown, destDim1);
mux2_10bit getSubmDim1(10'd30, 10'd10, rightOrDown, submDim1);
mux2_10bit getCruiDim1(10'd30, 10'd10, rightOrDown, cruiDim1);
mux2_10bit getBattDim1(10'd40, 10'd10, rightOrDown, battDim1);
mux2_10bit getCarrDim1(10'd50, 10'd10, rightOrDown, carrDim1);

mux2_10bit getDestDim2(10'd20, 10'd10, ~rightOrDown, destDim2);
mux2_10bit getSubmDim2(10'd30, 10'd10, ~rightOrDown, submDim2);
mux2_10bit getCruiDim2(10'd30, 10'd10, ~rightOrDown, cruiDim2);
mux2_10bit getBattDim2(10'd40, 10'd10, ~rightOrDown, battDim2);
mux2_10bit getCarrDim2(10'd50, 10'd10, ~rightOrDown, carrDim2);

mux2_10bit getSubmLocX1(10'd10, 10'd25, stage, submLocX1);
mux2_10bit getCruiLocX1(10'd10, 10'd40, stage, cruiLocX1);
mux2_10bit getBattLocX1(10'd10, 10'd55, stage, battLocX1);
mux2_10bit getCarrLocX1(10'd10, 10'd70, stage, carrLocX1);

mux2_10bit getSubmLocX2(10'd330, 10'd345, stage, submLocX2);
mux2_10bit getCruiLocX2(10'd330, 10'd360, stage, cruiLocX2);
mux2_10bit getBattLocX2(10'd330, 10'd375, stage, battLocX2);
mux2_10bit getCarrLocX2(10'd330, 10'd390, stage, carrLocX2);

mux2_10bit getSubmLocY(10'd415, 10'd400, stage, submLocY);
mux2_10bit getCruiLocY(10'd430, 10'd400, stage, cruiLocY);
mux2_10bit getBattLocY(10'd445, 10'd400, stage, battLocY);
mux2_10bit getCarrLocY(10'd460, 10'd400, stage, carrLocY);



// signals for displaying ships when choosing a placement for ship
assign destChoose1 = inDest1&&stage&&(~player)&&(shipIn==3'd1);
assign submChoose1 = inSubm1&&stage&&(~player)&&(shipIn==3'd2);
assign cruiChoose1 = inCrui1&&stage&&(~player)&&(shipIn==3'd3);
assign battChoose1 = inBatt1&&stage&&(~player)&&(shipIn==3'd4);
assign carrChoose1 = inCarr1&&stage&&(~player)&&(shipIn==3'd5);

assign destChoose2 = inDest2&&stage&&(player)&&(shipIn==3'd1);
assign submChoose2 = inSubm2&&stage&&(player)&&(shipIn==3'd2);
assign cruiChoose2 = inCrui2&&stage&&(player)&&(shipIn==3'd3);
assign battChoose2 = inBatt2&&stage&&(player)&&(shipIn==3'd4);
assign carrChoose2 = inCarr2&&stage&&(player)&&(shipIn==3'd5);

// signal to indicate when a block is being guessed
assign beingGuessed = (RAMBlockNumIn == blockNumRAddr) && ~we;

// logic for displaying ships that are sunk while playing
// receives ship sunk from PIC, writes it to RAM
// and when that ship address
// is accessed, outputs 1 for sunk, 0 for not sunk.
assign sunkIn = (shipIn != 3'd0);   // ships are decoded 1-5

//addr to write when a ship sinks
assign shipSunkWAddr = (player&&sunkIn) ? shipIn + 4'd4
                       : {1'd0,shipIn}-4'd1;
```

```
// get address to read ship to display on VGA
inWhichShip whichShip(inDest1, inSubm1, inCrui1, inBatt1, inCarr1,
                    inDest2, inSubm2, inCrui2, inBatt2, inCarr2, ship);

// ship RAM
ram2 isShipSunk(clk, we, shipSunkWAddr, ship, sunkIn, sunkOut);

// check if the coordinates are in each ship boxes
InBox inDestroyer1(col, row, destLocX1, destLocY,
                        destDim1, destDim2, inDest1);
InBox inSubmarine1(col, row, submLocX1, submLocY,
                        submDim1, submDim2, inSubm1);
InBox inCruiser1(col, row, cruiLocX1, cruiLocY,
                        cruiDim1, cruiDim2, inCrui1);
InBox inBattleship1(col, row, battLocX1, battLocY,
                        battDim1, battDim2, inBatt1);
InBox inCarrier1(col, row, carrLocX1, carrLocY,
                        carrDim1, carrDim2, inCarr1);

InBox inDestroyer2(col, row, destLocX2, destLocY,
                        destDim1, destDim2, inDest2);
InBox inSubmarine2(col, row, submLocX2, submLocY,
                        submDim1, submDim2, inSubm2);
InBox inCruiser2(col, row, cruiLocX2, cruiLocY,
                        cruiDim1, cruiDim2, inCrui2);
InBox inBattleship2(col, row, battLocX2, battLocY,
                        battDim1, battDim2, inBatt2);
InBox inCarrier2(col, row, carrLocX2, carrLocY,
                        carrDim1, carrDim2, inCarr2);


// assignment for write addr to RAM
assign RAMBlockNumIn = player ? blockNumWAddr + 8'd100 : blockNumWAddr;
// offset player 2's number by 100 to address RAM


// assignment for din to RAM
assign hitInTemp = {1'b0, hitIn};
// make it 2 bit since we have 3 different statuses

// 0 = not guessed (initial), 1 = missed, 2 = hit
assign RAMHitIn = hitInTemp+1'b1;


// get blockNumRAddr for read addr to RAM
// get inBounds, inPlayer1, inPlayer2 for
// boundary conditions for signal
GenBlockNum getBlockNum(clk, VSync, DataValid, inBounds, inPlayer1,
                        inPlayer2, blockNumRAddr, row, col);

// write to and read from RAM to use as signal for screen
ram getstatus(clk, we, RAMBlockNumIn, blockNumRAddr, RAMHitIn, status);

// flash to flash signals when someone wins
assign flash = win&&second;
```

```
// output signal assignment
assign Signal[0] = (status==2'b10)&&inBounds                      //red
                    || (status==2'b01)&&inBounds
                    || (~player)&&inPlayer1&&~win
                    || player&&inPlayer2&&~win
                    || inDest1&&sunkOut&&~stage&&~win
                    || inSubm1&&sunkOut&&~stage&&~win
                    || inCrui1&&sunkOut&&~stage&&~win
                    || inBatt1&&sunkOut&&~stage&&~win
                    || inCarr1&&sunkOut&&~stage&&~win
                    || inDest2&&sunkOut&&~stage&&~win
                    || inSubm2&&sunkOut&&~stage&&~win
                    || inCrui2&&sunkOut&&~stage&&~win
                    || inBatt2&&sunkOut&&~stage&&~win
                    || inCarr2&&sunkOut&&~stage&&~win
                    || inDest1&&sunkOut&&~player&&flash
                    || inSubm1&&sunkOut&&~player&&flash
                    || inCrui1&&sunkOut&&~player&&flash
                    || inBatt1&&sunkOut&&~player&&flash
                    || inCarr1&&sunkOut&&~player&&flash
                    || inDest2&&sunkOut&&player&&flash
                    || inSubm2&&sunkOut&&player&&flash
                    || inCrui2&&sunkOut&&player&&flash
                    || inBatt2&&sunkOut&&player&&flash
                    || inCarr2&&sunkOut&&player&&flash;
assign Signal[1] = (status==2'b01)&&inBounds                //green
                    || (~player)&&inPlayer1&&~win
                    || player&&inPlayer2&&~win
                    || (~player)&&inPlayer1&&flash
                    || player&&inPlayer2&&flash
                    || inDest1&&~sunkOut&&~stage&&~win
                    || inSubm1&&~sunkOut&&~stage&&~win
                    || inCrui1&&~sunkOut&&~stage&&~win
                    || inBatt1&&~sunkOut&&~stage&&~win
                    || inCarr1&&~sunkOut&&~stage&&~win
                    || inDest2&&~sunkOut&&~stage&&~win
                    || inSubm2&&~sunkOut&&~stage&&~win
                    || inCrui2&&~sunkOut&&~stage&&~win
                    || inBatt2&&~sunkOut&&~stage&&~win
                    || inCarr2&&~sunkOut&&~stage&&~win
                    || destChoose1
                    || submChoose1
                    || cruiChoose1
                    || battChoose1
                    || carrChoose1
                    || destChoose2
                    || submChoose2
                    || cruiChoose2
                    || battChoose2
                    || carrChoose2
                    || (beingGuessed&&(status==2'b00)&&inBounds);
assign Signal[2] = (status==2'b00)&&inBounds                //blue
                    || (status==2'b01)&&inBounds
                    || (~player)&&inPlayer1&&~win
                    || player&&inPlayer2&&~win
                    || (beingGuessed&&(status==2'b00)&&inBounds);
```

```
endmodule
```

ram

```
// 11-24-09
// This module holds information for each indexed block of the board
// Samule Lee (tlee@hmc.edu)
module ram
       # (parameter N = 8, M=2)
       (
       input  clk,
    input  we,
    input  [N-1:0] waddr,
       input  [N-1:0] raddr,
    input  [M-1:0] din,
    output [M-1:0] dout
    );


      reg [M-1:0] mem [0:2**N-1];

      always @ (posedge clk)
            if (we) mem[waddr] <= din;

      assign dout = mem[raddr];

endmodule
```

ram2

```
// Samuel Taeyoung Lee Fall 2009 (tlee@hmc.edu)
// ram modules to hold ship sunk data
module ram2
       # (parameter N = 4)
       (
       input  clk,
    input  we,
    input  [N-1:0] waddr,
       input  [N-1:0] raddr,
    input                din,
    output               dout
    );


      reg mem [0:2**N-1];

      always @ (posedge clk)
            if (we) mem[waddr] <= din;

      assign dout = mem[raddr];

endmodule
```

Appendix B: C Code

```
//FinalProject2.c
//Kyle Baran   kbaran@hmc.edu        12/7/09

/*This program allows 2 players to play Battleship.  To get input, the PIC polls a keypad
        that is used to enter the current player's guess, and continuously updates
        the FPGA.  A separate button is used to enter that guess.

        The FPGA shows which square will be guessed, which locations have already
        been guessed (along with whether those guesses were hits or misses), which ships
        have been sunk, which player is currently up, and whether or not someone has won,
        as well as the orientation of the current ship to be placed when the game is in placement
        mode.

        The PIC constantly tells the FPGA which square is being guessed, which player is up.
        whether the game is in placement or play mode, and whether someone has won.

        When the game starts, the PIC quickly raises and lowers a reset bit to reset the FPGA
        It then enters placement mode, where each player enters places his or her ships
        consecutively.

        The guess entered on the keypad represents the starting point of the ship, and
        a signal from an external DIP switch controlled by the user which indicates whether
        the ship will extend right or down from that location.  If either player tries to place
        the ship such that part of it will be off the game board or will overlap with an
        already-placed ship, the game ignores that attempt and has the user try again.
        During this mode the PIC constantly tells the FPGA which orientation is currently
        selected so that the FPGA can display that information.

        Once all the ships are placed, the game enters the play stage.  During this stage,
        when the player enters a guess, the PIC determines whether that guess hit
        an opponent's ship and updates variables accordingly.  The players alternate turns,
        guessing one spot at a time. When a guess is made, the PIC tells the FPGA if the guess
        was a hit, which ship was sunk by that guess (if any), and briefly sets an enable bit
        high so that the FPGA knows that an actual guess was high.  The bit is dropped low soon
        after so that further information is not registered by mistake.

        When all of a player's ships have been sunk, the PIC sets the victory bit high so that
        the FPGA can show the users that someone has won.  The PIC then enters an infinite loop
        so that nothing else can be changed.
*/

#include <p18f4520.h>
#include <stdlib.h>

void main(void);
void isr(void);
void Decode(int currentState);

int poll, update, i;
int toggle, numLeft, numRight;
char num, hit, hitDes, hitSub, hitCru, hitBat, hitCar;
int counter1 = 0;
int counter2 = 0;
int shipsPlaced;
int rightOrDown = 0;
char Grid1 [100] = {0};
char Grid2 [100] = {0};

//Used to control whether TMR0 calls for polls or just counts for a waiting period
int pollingMode;

// Interrupt vector address
#pragma code high_vector = 0x08
void high_interrupt(void)
{
        _asm
                GOTO isr
        _endasm
}
```

```
//Main code
#pragma code

void main(void)
{
        //Initialize the ships and set their damage statuses
        char des1Column [2];
        char des1Row [2];
        char des1Damage [2] = {0,0};

        char des2Column [2];
        char des2Row [2];
        char des2Damage [2] = {0,0};

        char sub1Column [3];
        char sub1Row [3];
        char sub1Damage       [3] = {0,0,0};

        char sub2Column [3];
        char sub2Row [3];
        char sub2Damage [3] = {0,0,0};

        char cru1Column [3];
        char cru1Row [3];
        char cru1Damage [3] = {0,0,0};

        char cru2Column [3];
        char cru2Row [3];
        char cru2Damage [3] = {0,0,0};

        char bat1Column [4];
        char bat1Row [4];
        char bat1Damage [4] = {0,0,0,0};

        char bat2Column [4];
        char bat2Row [4];
        char bat2Damage [4] = {0,0,0,0};

        char car1Column [5];
        char car1Row [5];
        char car1Damage [5] = {0,0,0,0,0};

        char car2Column [5];
        char car2Row [5];
        char car2Damage [5] = {0,0,0,0,0};

        char des1Sunk = 0;
        char sub1Sunk = 0;
        char cru1Sunk = 0;
        char bat1Sunk = 0;
        char car1Sunk = 0;
        char des2Sunk = 0;
        char sub2Sunk = 0;
        char cru2Sunk = 0;
        char bat2Sunk = 0;
        char car2Sunk = 0;

        int endgame = 0;

        int currentstate = 0;
        int nextstate = 0;
        /*  State values
                0 = PollC0
                1 = PollC1
                2 = PollC2
                3 = DecodeC0
                4 = DecodeC1
                5 = DecodeC2
                6 = Hold
        */
```

```
//Used to hold row and column, respectively
numLeft = 10;
numRight = 10;

//Used to trigger an update of the keypap display
update = 0;

//Used to determine if the current guess was a hit
hit = 0;

//Used for some for loops
i = 0;

//New number gotten from the keypad
num = 10;

//Haven't placed any ships initially
shipsPlaced = 0;

//None of the ships are initially sunk
hitDes = 0;
hitSub = 0;
hitCru = 0;
hitBat = 0;
hitCar = 0;

//Set up interrupts
INTCON = 0b10100000;   //Interrupts are enabled, TMR0 overflow interrupt enables
INTCON3 = 0b00000000;  //Need to do this
T0CON = 0b10000001;            //TMR0 is enabled, 16-bit timer with a prescalar of 2

TRISA = 0b0100000;             //Bit 5 of PORTA is used for getting the "fire" command

ADCON1 = 0b00001111;           //Set PORTA to all-digital mode; need to do this or
                                       PORTA bits won't function as digital inputs

TRISC = 0b01000000;            //PORTC is used for outputting information to the FPGA and
                                       inputting the direction of the ships
                               //when placing the ships
/*
       PORTC[0] controls column 0 of the keypad
       PORTC[1] tells the FPGA whether the game is in ship-placing mode or guess mode
       PORTC[2] tells the FPGA to register the current potential guess as an actual guess
       PORTC[3] tells the FPGA which player is up
       PORTC[4] tells the FPGA that the current player has won
       PORTC[5] clears the FPGA on PIC startup
       PORTC[6] determines, in ship placement, whether the ship is facing right or down
       PORTC[7] tells the FPGA whether the current guess was a hit or not
*/
TRISB = 0b00001111;            //PORTB is used for polling the keypad; the 2 most
                                       significant bits are for columns 0 and 1
                               // and the 4 least significant bits are for the rows, which
                                       are read from to determine button presses
TRISD = 0;                     //PORTD is used for telling the FPGA the current values of
                                       the row and column guess
TRISE = 0;                     //PORTE is used for telling the FPGA which ship is either
                                       due up to be placed or has just been sunk
/*
       000 = no ship
       001 = destroyer
       010 = submarine
       011 = cruiser
       100 = battleship
       101 = aircraft carrier
*/

//Reset all ports
PORTA = 0;
PORTB = 0;
PORTC = 0;
```

```
PORTD = 0b10101010;
PORTE = 0;

//Clear FPGA
PORTCbits.RC5 = 1;
PORTCbits.RC5 = 0;

//Start off with player 1 guessing
PORTCbits.RC3 = 0;

//Start off polling the keypad
pollingMode = 1;

while (!endgame)
{
        //Update the right/down status from RC6
        rightOrDown = PORTCbits.RC6;

        //Until all ships have been placed, the game is in place mode
        if (shipsPlaced < 10)
PORTCbits.RC1 = 1;
        else
PORTCbits.RC1 = 0;

        //If fewer than 5 ships have been placed, then Player 1 is placing; otherwise
                Player 2 is placing
        if (shipsPlaced < 5)
PORTCbits.RC3 = 0;
        else if (shipsPlaced >= 5 && shipsPlaced < 10)              PORTCbits.RC3 = 1;

        //Tells the FPGA which ship is being placed
        if (shipsPlaced == 0 || shipsPlaced == 5)                           PORTE = 1;
        else if (shipsPlaced == 1 || shipsPlaced == 6)                      PORTE = 2;
        else if (shipsPlaced == 2 || shipsPlaced == 7)                      PORTE = 3;
        else if (shipsPlaced == 3 || shipsPlaced == 8)                      PORTE = 4;
        else if (shipsPlaced == 4 || shipsPlaced == 9)                      PORTE = 5;


        //Polls a column to determine if a button has been pressed, and if so, determines
                what the display should be updated to
        if (poll)
        {
                //Reset poll bit
                poll = 0;

                //Poll the first column; if there's exactly one high row, then try to
                        decode, otherwise poll column 1
                if (currentstate == 0)
                {
                        PORTCbits.RC0 = 0;
                        PORTBbits.RB5 = 0;
                        PORTBbits.RB4 = 1;
                        if (PORTBbits.RB0 ^ PORTBbits.RB1 ^ PORTBbits.RB2 ^ PORTBbits.RB3)
                                nextstate = 3;
                        else    nextstate = 1;
                }
                //Poll the second column; if there's exactly one high row, then try to
                        decode, otherwise poll column 2
                else if (currentstate == 1)
                {
                        PORTCbits.RC0 = 0;
                        PORTBbits.RB5 = 1;
                        PORTBbits.RB4 = 0;
                        if (PORTBbits.RB0 ^ PORTBbits.RB1 ^ PORTBbits.RB2 ^ PORTBbits.RB3)
                                nextstate = 4;
                        else    nextstate = 2;
                }
                //Poll the third column; if there's exactly one high row, then try to
                        decode, otherwise poll column 0
                else if (currentstate == 2)
                {
```

```
                PORTCbits.RC0 = 1;
                PORTBbits.RB5 = 0;
                PORTBbits.RB4 = 0;
                if (PORTBbits.RB0 ^ PORTBbits.RB1 ^ PORTBbits.RB2 ^ PORTBbits.RB3)
                        nextstate = 5;
                else    nextstate = 0;
        }

        //If decoding column 0 and still only one high row, then decode, otherwise
                 go back and poll column 0
        else if (currentstate == 3)
        {
                PORTCbits.RC0 = 0;
                PORTBbits.RB5 = 0;
                PORTBbits.RB4 = 1;
                if (PORTBbits.RB0 ^ PORTBbits.RB1 ^ PORTBbits.RB2 ^ PORTBbits.RB3)
                {
                        nextstate = 6;
                        Decode(currentstate);
                }
                else    nextstate = 0;
        }
        //If decoding column 1 and still only one high row, then decode, otherwise
                 go back and poll column 1
        else if (currentstate == 4)
        {
                PORTCbits.RC0 = 0;
                PORTBbits.RB5 = 1;
                PORTBbits.RB4 = 0;
                if (PORTBbits.RB0 ^ PORTBbits.RB1 ^ PORTBbits.RB2 ^ PORTBbits.RB3)
                {
                        nextstate = 6;
                        Decode(currentstate);
                }
                else            nextstate = 1;

        }
        //If decoding column 2 and still only one high row, then decode, otherwise
                 go back and poll column 2
        else if (currentstate == 5)
        {
                PORTCbits.RC0 = 1;
                PORTBbits.RB5 = 0;
                PORTBbits.RB4 = 0;
                if (PORTBbits.RB0 ^ PORTBbits.RB1 ^ PORTBbits.RB2 ^ PORTBbits.RB3)
                {
                        nextstate = 6;
                        Decode(currentstate);
                }
                else            nextstate = 2;

        }
        //Holding state to ensure that holding down a button doesn't register as
                 multiple presses
        else if (currentstate == 6)
        {
                if (~PORTBbits.RB0 && ~PORTBbits.RB1 &&
                        ~PORTBbits.RB2 && ~PORTBbits.RB3)
                {
                        nextstate = 0;
                }
                else    nextstate = 6;
        }


        //Update the current state
        currentstate = nextstate;
}


//If the number has changed, update the numbers
```

```
if (update)
{
        update = 0;
        numLeft = numRight;
        numRight = num;

        //PORTD[7:4] contains the row information and PORTD[3:0] contains the
                column information
        PORTD = ((numLeft*16)+numRight);
}

//Controls the initial ship placement
if (~PORTAbits.RA5 && numLeft != 10 && numRight != 10 && shipsPlaced < 10)
{

        //Don't want to poll for the moment
        pollingMode = 0;

        //Logic for placing Player 1's destroyer
        //If the player wants it to be placed horizontally (right), check to see
                if it's a valid placement
        //If so, fill in the rest of the spaces horizontally
        if (shipsPlaced == 0 && rightOrDown && ((numLeft + 1) <= 9))
        {
                for (i = 0; i < 2; i++)
                {
                        des1Row[i] = numLeft + i;
                        des1Column[i] = numRight;
                }
                Grid1[des1Row[0]*10+des1Column[0]] = 1;
                Grid1[des1Row[1]*10+des1Column[1]] = 1;
                shipsPlaced++;
        }
        //If the player wants it to be placed vertically (down), check to see if
                it's a valid placement
        //If so, fill in the rest of the spaces vertically
        else if (shipsPlaced == 0 && !rightOrDown && ((numRight + 1) <= 9))
        {
                for (i = 0; i < 2; i++)
                {
                        des1Row[i] = numLeft;
                        des1Column[i] = numRight + i;
                }
                Grid1[des1Row[0]*10+des1Column[0]] = 1;
                Grid1[des1Row[1]*10+des1Column[1]] = 1;
                shipsPlaced++;
        }

        //Logic for placing Player 1's submarine
        else if (shipsPlaced == 1 && rightOrDown && ((numLeft + 2) <= 9) &&
                !(Grid1[numLeft*10+numRight] || Grid1[(numLeft+1)*10+numRight] ||
                Grid1[(numLeft+2)*10+numRight]))
        {
                for (i = 0; i < 3; i++)
                {
                        sub1Row[i] = numLeft + i;
                        sub1Column[i] = numRight;
                }
                Grid1[sub1Row[0]*10+sub1Column[0]] = 1;
                Grid1[sub1Row[1]*10+sub1Column[1]] = 1;
                Grid1[sub1Row[2]*10+sub1Column[2]] = 1;
                shipsPlaced++;
        }
        else if (shipsPlaced == 1 && !rightOrDown && ((numRight + 2) <= 9) &&
                !(Grid1[numLeft*10+numRight] || Grid1[numLeft*10+(numRight+1)] ||
                Grid1[numLeft*10+(numRight+2)]))
        {
                for (i = 0; i < 3; i++)
                {
                        sub1Row[i] = numLeft;
                        sub1Column[i] = numRight + i;
                }
                Grid1[sub1Row[0]*10+sub1Column[0]] = 1;
                Grid1[sub1Row[1]*10+sub1Column[1]] = 1;
                Grid1[sub1Row[2]*10+sub1Column[2]] = 1;
                shipsPlaced++;
        }

        //Logic for placing Player 1's cruiser
        else if (shipsPlaced == 2 && rightOrDown && ((numLeft + 2) <= 9) &&
                !(Grid1[numLeft*10+numRight] || Grid1[(numLeft+1)*10+numRight] ||
                Grid1[(numLeft+2)*10+numRight]))
        {
                for (i = 0; i < 3; i++)
                {
                        cru1Row[i] = numLeft + i;
                        cru1Column[i] = numRight;
                }
                Grid1[cru1Row[0]*10+cru1Column[0]] = 1;
                Grid1[cru1Row[1]*10+cru1Column[1]] = 1;
                Grid1[cru1Row[2]*10+cru1Column[2]] = 1;
                shipsPlaced++;
        }
        else if (shipsPlaced == 2 && !rightOrDown && ((numRight + 2) <= 9) &&
                !(Grid1[numLeft*10+numRight] || Grid1[numLeft*10+(numRight+1)] ||
                Grid1[numLeft*10+(numRight+2)]))
        {
                for (i = 0; i < 3; i++)
                {
                        cru1Row[i] = numLeft;
                        cru1Column[i] = numRight + i;
                }
                Grid1[cru1Row[0]*10+cru1Column[0]] = 1;
                Grid1[cru1Row[1]*10+cru1Column[1]] = 1;
                Grid1[cru1Row[2]*10+cru1Column[2]] = 1;
                shipsPlaced++;
        }

        //Logic for placing Player 1's battleship
        else if (shipsPlaced == 3 && rightOrDown && ((numLeft + 3) <= 9) &&
                !(Grid1[numLeft*10+numRight] || Grid1[(numLeft+1)*10+numRight] ||
                Grid1[(numLeft+2)*10+numRight] || Grid1[(numLeft+3)*10+numRight]))
        {
                for (i = 0; i < 4; i++)
                {
                        bat1Row[i] = numLeft + i;
                        bat1Column[i] = numRight;
                }
                Grid1[bat1Row[0]*10+bat1Column[0]] = 1;
                Grid1[bat1Row[1]*10+bat1Column[1]] = 1;
                Grid1[bat1Row[2]*10+bat1Column[2]] = 1;
                Grid1[bat1Row[3]*10+bat1Column[3]] = 1;
                shipsPlaced++;
        }
        else if (shipsPlaced == 3 && !rightOrDown && ((numRight + 3) <= 9) &&
                !(Grid1[numLeft*10+numRight] || Grid1[numLeft*10+(numRight+1)] ||
                Grid1[numLeft*10+(numRight+2)] || Grid1[numLeft*10+(numRight+3)]))
        {
                for (i = 0; i < 4; i++)
                {
                        bat1Row[i] = numLeft;
                        bat1Column[i] = numRight + i;
                }
                Grid1[bat1Row[0]*10+bat1Column[0]]= 1;
                Grid1[bat1Row[1]*10+bat1Column[1]]= 1;
                Grid1[bat1Row[2]*10+bat1Column[2]]= 1;
                Grid1[bat1Row[3]*10+bat1Column[3]] = 1;
                shipsPlaced++;
        }

        //Logic for placing Player 1's aircraft carrier
        else if (shipsPlaced == 4 && rightOrDown && ((numLeft + 4) <= 9) &&
                !(Grid1[numLeft*10+numRight] || Grid1[(numLeft+1)*10+numRight] ||
                Grid1[(numLeft+2)*10+numRight] || Grid1[(numLeft+3)*10+numRight]
                || Grid1[(numLeft+4)*10+numRight]))
        {
                for (i = 0; i < 5; i++)
                {
                        car1Row[i] = numLeft + i;
                        car1Column[i] = numRight;
                }
                Grid1[car1Row[0]*10+car1Column[0]] = 1;
                Grid1[car1Row[1]*10+car1Column[1]] = 1;
                Grid1[car1Row[2]*10+car1Column[2]] = 1;
                Grid1[car1Row[3]*10+car1Column[3]] = 1;
                Grid1[car1Row[4]*10+car1Column[4]] = 1;
                shipsPlaced++;
        }
        else if (shipsPlaced == 4 && !rightOrDown && ((numRight + 4) <= 9) &&
                !(Grid1[numLeft*10+numRight] || Grid1[numLeft*10+(numRight+1)] ||
                Grid1[numLeft*10+(numRight+2)] || Grid1[numLeft*10+(numRight+3)]
                || Grid1[numLeft*10+(numRight+4)]))
        {
                for (i = 0; i < 5; i++)
                {
                        car1Row[i] = numLeft;
                        car1Column[i] = numRight + i;
                }
                Grid1[car1Row[0]*10+car1Column[0]] = 1;
                Grid1[car1Row[1]*10+car1Column[1]] = 1;
                Grid1[car1Row[2]*10+car1Column[2]] = 1;
                Grid1[car1Row[3]*10+car1Column[3]] = 1;
                Grid1[car1Row[4]*10+car1Column[4]] = 1;
                shipsPlaced++;
        }


        //Logic for placing Player 2's destroyer
        else if (shipsPlaced == 5 && rightOrDown && ((numLeft + 1) <= 9))
        {
                for (i = 0; i < 2; i++)
                {
                        des2Row[i] = numLeft + i;
                        des2Column[i] = numRight;
                }
                Grid2[des2Row[0]*10+des2Column[0]] = 1;
                Grid2[des2Row[1]*10+des2Column[1]] = 1;
                shipsPlaced++;
        }
        else if (shipsPlaced == 5 && !rightOrDown && ((numRight + 1) <= 9))
        {
                for (i = 0; i < 2; i++)
                {
                        des2Row[i] = numLeft;
                        des2Column[i] = numRight + i;
                }
                Grid2[des2Row[0]*10+des2Column[0]] = 1;
                Grid2[des2Row[1]*10+des2Column[1]] = 1;
                shipsPlaced++;
        }

        //Logic for placing Player 2's submarine
        else if (shipsPlaced == 6 && rightOrDown && ((numLeft + 2) <= 9) &&
                !(Grid2[numLeft*10+numRight] || Grid2[(numLeft+1)*10+numRight] ||
                Grid2[(numLeft+2)*10+numRight]))
        {
                for (i = 0; i < 3; i++)
                {
                        sub2Row[i] = numLeft + i;
                        sub2Column[i] = numRight;
                }
                Grid2[sub2Row[0]*10+des2Column[0]] = 1;
                Grid2[sub2Row[1]*10+des2Column[1]] = 1;
                Grid2[sub2Row[2]*10+des2Column[2]] = 1;
                shipsPlaced++;
        }
        else if (shipsPlaced == 6 && !rightOrDown && ((numRight + 2) <= 9) &&
                !(Grid2[numLeft*10+numRight] || Grid2[numLeft*10+(numRight+1)] ||
                Grid2[numLeft*10+(numRight+2)]))
        {
                for (i = 0; i < 3; i++)
                {
                        sub2Row[i] = numLeft;
                        sub2Column[i] = numRight + i;
                }
                Grid2[sub2Row[0]*10+des2Column[0]] = 1;
                Grid2[sub2Row[1]*10+des2Column[1]] = 1;
                Grid2[sub2Row[2]*10+des2Column[2]] = 1;
                shipsPlaced++;
        }

        //Logic for placing Player 2's cruiser
        else if (shipsPlaced == 7 && rightOrDown && ((numLeft + 2) <= 9) &&
                !(Grid2[numLeft*10+numRight] || Grid2[(numLeft+1)*10+numRight] ||
                Grid2[(numLeft+2)*10+numRight]))
        {
                for (i = 0; i < 3; i++)
                {
                        cru2Row[i] = numLeft + i;
                        cru2Column[i] = numRight;
                }
                Grid2[cru2Row[0]*10+cru2Column[0]] = 1;
                Grid2[cru2Row[1]*10+cru2Column[1]] = 1;
                Grid2[cru2Row[2]*10+cru2Column[2]] = 1;
                shipsPlaced++;
        }
        else if (shipsPlaced == 7 && !rightOrDown && ((numRight + 2) <= 9) &&
                !(Grid2[numLeft*10+numRight] || Grid2[numLeft*10+(numRight+1)] ||
                Grid2[numLeft*10+(numRight+2)]))
        {
                for (i = 0; i < 3; i++)
                {
                        cru2Row[i] = numLeft;
                        cru2Column[i] = numRight + i;
                }
                Grid2[cru2Row[0]*10+cru2Column[0]] = 1;
                Grid2[cru2Row[1]*10+cru2Column[1]] = 1;
                Grid2[cru2Row[2]*10+cru2Column[2]] = 1;
                shipsPlaced++;
        }

        //Logic for placing Player 2's battleship
        else if (shipsPlaced == 8 && rightOrDown && ((numLeft + 3) <= 9) &&
                !(Grid2[numLeft*10+numRight] || Grid2[(numLeft+1)*10+numRight] ||
                Grid2[(numLeft+2)*10+numRight] || Grid2[(numLeft+3)*10+numRight]))
        {
                for (i = 0; i < 4; i++)
                {
                        bat2Row[i] = numLeft + i;
                        bat2Column[i] = numRight;
                }
                Grid2[bat2Row[0]*10+bat2Column[0]] = 1;
                Grid2[bat2Row[1]*10+bat2Column[1]] = 1;
                Grid2[bat2Row[2]*10+bat2Column[2]] = 1;
                Grid2[bat2Row[3]*10+bat2Column[3]] = 1;
                shipsPlaced++;
        }
        else if (shipsPlaced == 8 && !rightOrDown && ((numRight + 3) <= 9) &&
                !(Grid2[numLeft*10+numRight] || Grid2[numLeft*10+(numRight+1)] ||
                Grid2[numLeft*10+(numRight+2)] || Grid2[numLeft*10+(numRight+3)]))
        {
                for (i = 0; i < 4; i++)
                {
                        bat2Row[i] = numLeft;
```

```
                        bat2Column[i] = numRight + i;
                }
                Grid2[bat2Row[0]*10+bat2Column[0]] = 1;
                Grid2[bat2Row[1]*10+bat2Column[1]] = 1;
                Grid2[bat2Row[2]*10+bat2Column[2]] = 1;
                Grid2[bat2Row[3]*10+bat2Column[3]] = 1;
                shipsPlaced++;
        }

        //Logic for placing Player 2's aircraft carrier
        else if (shipsPlaced == 9 && rightOrDown && ((numLeft + 4) <= 9) &&
                 !(Grid2[numLeft*10+numRight] || Grid2[(numLeft+1)*10+numRight] ||
                 Grid2[(numLeft+2)*10+numRight] || Grid2[(numLeft+3)*10+numRight]
                 || Grid2[(numLeft+4)*10+numRight]))
        {
                for (i = 0; i < 5; i++)
                {
                        car2Row[i] = numLeft + i;
                        car2Column[i] = numRight;
                }
                Grid2[car2Row[0]*10+car2Column[0]] = 1;
                Grid2[car2Row[1]*10+car2Column[1]] = 1;
                Grid2[car2Row[2]*10+car2Column[2]] = 1;
                Grid2[car2Row[3]*10+car2Column[3]] = 1;
                Grid2[car2Row[4]*10+car2Column[4]] = 1;
                shipsPlaced++;
        }
        else if (shipsPlaced == 9 && !rightOrDown && ((numRight + 4) <= 9) &&
                 !(Grid2[numLeft*10+numRight] || Grid2[numLeft*10+(numRight+1)] ||
                 Grid2[numLeft*10+(numRight+2)] || Grid2[numLeft*10+(numRight+3)]
                 || Grid2[numLeft*10+(numRight+4)]))
        {
                for (i = 0; i < 5; i++)
                {
                        car2Row[i] = numLeft;
                        car2Column[i] = numRight + i;
                }
                Grid2[car2Row[0]*10+car2Column[0]] = 1;
                Grid2[car2Row[1]*10+car2Column[1]] = 1;
                Grid2[car2Row[2]*10+car2Column[2]] = 1;
                Grid2[car2Row[3]*10+car2Column[3]] = 1;
                Grid2[car2Row[4]*10+car2Column[4]] = 1;
                shipsPlaced++;
        }

        if (shipsPlaced == 10)
        {
                //If all the ships have been placed, then reset the ship output and
                         set the game to play mode
                PORTE = 0;
                PORTCbits.RC3 = 0;
        }

        //Reset the row and column information after a placememnt attempt
        numLeft = 10;
        numRight = 10;
        PORTD = 0b10101010;
        pollingMode = 1;

}
//If "fire" button is pushed and a valid target has been entered (2 digits are
         present) and the game's in play mode
else if (~PORTAbits.RA5 && numLeft != 10 && numRight != 10 && shipsPlaced == 10)
{
        //Change TMR0 from calling polls to just waiting
        pollingMode = 0;

        //Reset hit bit
        hit = 0;

        //Hit logic if Player 1 has guessed
```

```
if (~PORTCbits.RC3)
{
        //Checks if destroyer is hit
        for (i = 0; i < 2; i++)
        {
                if ((des2Row[i] == numLeft) && (des2Column[i] == numRight))

                {
                        des2Damage[i] = 1;
                        hit = 1;
                        hitDes = 1;
                }
        }
        //Checks if cruiser or submarine have been hit
        for (i = 0; i < 3; i++)
        {
                if ((sub2Row[i] == numLeft) && (sub2Column[i] == numRight))

                {
                        sub2Damage[i] = 1;
                        hit = 1;
                        hitSub = 1;
                }
                if ((cru2Row[i] == numLeft) && (cru2Column[i] == numRight))

                {
                        cru2Damage[i] = 1;
                        hit = 1;
                        hitCru = 1;
                }
        }
        //Checks if battleship has been hit
        for (i = 0; i < 4; i++)
        {
                if ((bat2Row[i] == numLeft) && (bat2Column[i] == numRight))

                {
                        bat2Damage[i] = 1;
                        hit = 1;
                        hitBat = 1;
                }
        }
        //Checks if aircraft carrier has been hit
        for (i = 0; i < 5; i++)
        {
                if ((car2Row[i] == numLeft) && (car2Column[i] == numRight))

                {
                        car2Damage[i] = 1;
                        hit = 1;
                        hitCar = 1;
                }
        }

        //Determine if a ship has been sunk this turn, and update both
                internal logic and the FPGA accordingly
        if (des2Damage[0] && des2Damage[1] && hitDes)
        {
                PORTE = 1;
                des2Sunk = 1;
        }
        if (sub2Damage[0] && sub2Damage[1] && sub2Damage[2] && hitSub)
        {
                PORTE = 2;
                sub2Sunk = 1;
        }
        if (cru2Damage[0] && cru2Damage[1] && cru2Damage[2] && hitCru)
        {
                PORTE = 3;
                cru2Sunk = 1;
        }
```

```
                if (bat2Damage[0] && bat2Damage[1] && bat2Damage[2] &&
                         bat2Damage[3] && hitBat)
                {
                        PORTE = 4;
                        bat2Sunk = 1;
                }
                if (car2Damage[0] && car2Damage[1] && car2Damage[2] &&
                         car2Damage[3] && car2Damage[4] && hitCar)
                {
                        PORTE = 5;
                        car2Sunk = 1;
                }

        }
        //Hit logic if player 2 has guessed
        else
        {
                //Checks if destroyer is hit
                for (i = 0; i < 2; i++)
                {
                        if ((des1Row[i] == numLeft) && (des1Column[i] == numRight))

                        {
                                des1Damage[i] = 1;
                                hit = 1;
                                hitDes = 1;
                        }
                }
                //Checks if cruiser or submarine have been hit
                for (i = 0; i < 3; i++)
                {
                        if ((sub1Row[i] == numLeft) && (sub1Column[i] == numRight))

                        {
                                sub1Damage[i] = 1;
                                hit = 1;
                                hitSub = 1;
                        }
                        if ((cru1Row[i] == numLeft) && (cru1Column[i] == numRight))

                        {
                                cru1Damage[i] = 1;
                                hit = 1;
                                hitCru = 1;
                        }
                }
                //Checks if battleship has been hit
                for (i = 0; i < 4; i++)
                {
                        if ((bat1Row[i] == numLeft) && (bat1Column[i] == numRight))

                        {
                                bat1Damage[i] = 1;
                                hit = 1;
                                hitBat = 1;
                        }
                }
                //Checks if aircraft carrier has been hit
                for (i = 0; i < 5; i++)
                {
                        if ((car1Row[i] == numLeft) && (car1Column[i] == numRight))

                        {
                                car1Damage[i] = 1;
                                hit = 1;
                                hitCar = 1;
                        }
                }

                //Determine if a ship has been sunk this turn, and update both
                         internal logic and the FPGA accordingly
                if (des1Damage[0] && des1Damage[1] && hitDes)
                {
                        PORTE = 1;
                        des1Sunk = 1;
                }
                if (sub1Damage[0] && sub1Damage[1] && sub1Damage[2] && hitSub)
                {
                        PORTE = 2;
                        sub1Sunk = 1;
                }
                if (cru1Damage[0] && cru1Damage[1] && cru1Damage[2] && hitCru)
                {
                        PORTE = 3;
                        cru1Sunk = 1;
                }
                if (bat1Damage[0] && bat1Damage[1] && bat1Damage[2] &&
                         bat1Damage[3] && hitBat)
                {
                        PORTE = 4;
                        bat1Sunk = 1;
                }
                if (car1Damage[0] && car1Damage[1] && car1Damage[2] &&
                         car1Damage[3] && car1Damage[4] && hitCar)
                {
                        PORTE = 5;
                        car1Sunk = 1;
                }
        }

        //If all ships sunk, victory!
        if ((des1Sunk && sub1Sunk && cru1Sunk && bat1Sunk && car1Sunk) |
                (des2Sunk && sub2Sunk && cru2Sunk && bat2Sunk && car2Sunk))
                endgame = 1;

        //Tell FPGA if the guess was a hit or not
        if (hit)                PORTCbits.RC7 = 1;
        else                    PORTCbits.RC7 = 0;

        //Tell the FPGA that a guess was confirmed
        PORTCbits.RC2 = 1;

        //Stop polling for now
        pollingMode = 0;

        //Pause so that current player can see result of his/her guess
        while(!pollingMode)
        {}

        //Stop sending valid bit to FPGA
        PORTCbits.RC2 = 0;
        if (!endgame)
        {
                //If the game hasn't ended, switch players
                if (PORTCbits.RC3)              PORTCbits.RC3 = 0;
                else                                    PORTCbits.RC3 = 1;

                //Reset row/column information
                numLeft = 10;
                numRight = 10;
                PORTD = 0b10101010;

                //Reset secondary ship hit status
                hitDes = 0;
                hitSub = 0;
                hitCru = 0;
                hitBat = 0;
                hitCar = 0;
        }

        //Reset the ship sunk output
        PORTE =0;
```

```
		}

	}

	//Victory condition; just set the victory bit high
	while (1)
	{
		PORTCbits.RC4 = 1;
	}

}

//Decodes the current button press from the current values of PORTB
void Decode(int currentstate)
{
	if (PORTBbits.RB0 ^ PORTBbits.RB1 ^ PORTBbits.RB2 ^ PORTBbits.RB3)
	{
		if (currentstate == 3)
		{
			if (PORTBbits.RB0)
			{
				num = 1;
				update = 1;
			}
			else if (PORTBbits.RB1)
			{
				num = 4;
				update = 1;
			}
			else if (PORTBbits.RB2)
			{
				num = 7;
				update = 1;
			}
			//Row 3 would be E, but we're not using it
		}
		else if (currentstate == 4)
		{
			if (PORTBbits.RB0)
			{
				num = 2;
				update = 1;
			}
			else if (PORTBbits.RB1)
			{
				num = 5;
				update = 1;
			}
			else if (PORTBbits.RB2)
			{
				num = 8;
				update = 1;
			}
			else if (PORTBbits.RB3)
			{
				num = 0;
				update = 1;
			}
		}
		else if (currentstate == 5)
		{
			if (PORTBbits.RB0)
			{
				num = 3;
				update = 1;
			}
			else if (PORTBbits.RB1)
			{
				num = 6;
				update = 1;
```

```
                        }
                        else if (PORTBbits.RB2)
                        {
                                num = 9;
                                update = 1;
                        }
                        //Row 3 would be F, but we're not using it
                }
        }
}


//Interrupt handler
#pragma interrupt isr
void isr(void)
{
        //TMR0 interrupt logic
        if (INTCONbits.TMR0IF)
        {

                //If TMR0 is being used to poll the keypad
                if (pollingMode)
                {
                        poll = 1;
                }
                //If TMR0 is just a wait timer
                else
                {
                        counter2++;
                        //20 TMR0 overflows- pause after guess is entered
                        if (counter2 == 20)
                        {
                                counter2 = 0;
                                //Want to restart polling mode
                                pollingMode = 1;
                        }
                }

                INTCONbits.TMR0IF = 0;
        }
}
```