Proximity Sensor Pong

Play pong by placing your hand over sensors, moving the paddle to the desired location.

Final Report

Ian Chan	ichan@ualberta.ca
Kevin Au	kau@ualberta.ca
Preferred Lab Days	Monday, Wednesday

Table of Contents

1 Declaration of Original Content

“The design elements of this project and report are entirely the original work of the authors and have not been submitted for credit in any other course except as follows: [10]

Hardware Schematics were based off of [1] and [2].

Proximity sensor circuitry based off of [3] and [4].

Learning to use Xilinx and programming an FPGA heavily relied on [5], [6], and [13].

Pong game similar to the one found at [7].

Space invaders game (not implemented) similar to the one found at [8].

Many ideas and suggestions for the design of the project thanks to [9].

Base VGA code used for the Xilinx FPGA found at [11]. (Thanks Andrew and Nancy!)

This document template used with permission from [12].

Many, many thanks to David for lending us his propeller board for the VGA.

VGA_1280x1024_tiled_driver packaged with Parallax Propeller Tool [14].

Serial communication driver packaged with Parallax Propeller Tool [14].

VGA demo2 used as skeleton code, and was packaged with Parallax Propeller Tool [14].

Serial code for the FPGA was modified from [15].

2 Abstract

Tracking of humans can be useful for many technological applications. It allows for computer systems to understand the position, and to a lesser extent, intention of a user. Computer systems created to respond to human needs may have need to know this information in order to attend to the user, using motion-tracking as a means to determine user position. A few examples of technologies that could be based off this are: climate control based on human presence, customized music that follows specific users around the house, and tailoring security needs to traffic flow.

The final design of our project included two arrays of proximity sensors (also referred to as motion sensors in this document) used to determine the position of a user's hand and using this information to control the movement of a paddle in the game of Pong. This game outputs its video from a propeller board (also referred to as the Proto Board in this document) to a projector (or any other VGA capable display). Possible game clones we are considering are Space Invaders and Brickbreaker. Code handling the serial packets and display were written in basic and located on the propeller board. Code for polling the proximity sensors and sending the serial packets was handled on the Atmel board.

3 Functional Requirements

- The proximity sensors needs to send a signal to the microprocessor when something moves within its intended radius. This requirement was met.

- The Atmel board needs to determine the position of the user based on which proximity sensors are triggered. This was originally supposed to be able to detect full body motion, but we were unable to get the required range on the sensors. This requirement was not quite met, as the sensors only had a small range (5cm to 10cm), but they worked otherwise.
- The Atmel board needs to use the triggered proximity sensors to determine where the pong paddles should be located. The original idea was to use the direction of triggered sensors to move the paddle in the desired direction. However, this would have made the game much more difficult to play, so this requirement was reduced to just positional detection rather than directional. In terms of placing the paddle positionally with respect to the sensors, this requirement was met.
- The Atmel board needs to compute the internal workings of Pong. This requirement was met.
- The Atmel board needs to interface to the VGA connector of the propeller board and send graphics/video instructions (serially, using packets) to the propeller board. This requirement was met with a different board (propeller) than originally intended (FPGA).
- The propeller board needs to parse the packets and render the graphics to create a video output usable by the projector. This requirement was met.
- The propeller board needs to send a video output to the projector based on the user's movement. This requirement was met.
- In order to keep track of the sprites, all sprites will be assigned an ID number. This requirement was met.

4 Design and Description of Operation

Our design requires that we make a panel of motion sensors to detect a user's hand. The panel's sensors then communicate to the processor board, which we use to determine the position and the direction of movement of the user's hand. We intended that the user's actions would be the input method for the game

We are basically making our own motion sensors by assembling an IR emitter and a phototransistor. When an object or hand is within range of the emitter, the infrared light will bounce off it, and some of it will return to the phototransistor. This triggers the phototransistor to send a signal to the processor board.

We are planning on using rows of 12 motion sensors. One row of motion sensors can detect one dimension of movement. Depending on the motion sensor's effective radius, we can increase or decrease the number of motion sensors in a row. This would be adjusted to achieve a sufficient amount of granularity for the least amount of sensors. In this case, granularity would refer to how precisely the user can position the paddle to play Pong.

A motion sensor has a effective range which in this case is a semi-spherical radius directly in front of it. If several of these motion sensors are placed beside each other, with their effective ranges oriented in the same direction, then the overlapping areas become distinct areas of motion. Motion detected by just the first motion sensor will indicate a different position than motion detected by the first and second sensors. Using this technique to isolate and refine position tracking, we can compute direction based on the change of position. This means that the output of all the motion sensors, needs to be collected and computed, by the processor we use. Refer to the appendix for a diagram outlining this method.

Once we understand the user's movement, we can use this as input into the simple game we made. After processing the game on the Atmel processor board, we compute the output used to instruct the Parallax Proto Board how to construct the graphics we wish to display. Generally, it will be things like: a coordinate of where to draw a sprite, the type of sprite associated with that coordinate, a "Pause" screen, etc. The instruction from the Atmel board is passed through the UART2 pins configured on the Atmel board, and use the predefined UART drivers. We send a series of bytes whose order denote the role of the values. This series of bytes will be referred to as a packet from this point on.

On the Proto Board side, we use a predefined UART driver which needs to be configured to the same baud rate as the Atmel's UART driver. The UART driver, as are all the other predefined drivers used by the Proto Board in this project, is packaged with the Parallax programming tool. This serial communication driver has no static pins on which to receive the receive (Rx) and transmit (Tx) data. This means that we had to define which pins it receives and transmits on, which was convenient in that it allowed us to organize the layout of the Proto board for our purposes. In our particular case, some soldering had already been on the board, which allowed for convenient mapping of pins from the Proto Board to the Atmel board. We used a 300 ohm resistor to limit the current between the two boards as a precaution.

After receiving the bytes of data from the Atmel Board, the Proto Board needs to parse the bytes received. Since the UART driver on the Proto Board feeds the bytes received into a buffer (16-byte size, though this is configurable in the driver) , we were able to sequentially grab bytes as we needed them. Using a start identifier byte to differentiate the packet from random noise, we took the next 3 bytes to be valuable information. We then used a stop identifier byte to ensure that the packet was not lagging (hence the previous bytes received would be useless), and that the start identifier had not been recreated by noise through random chance. Refer to the diagram in the appendix that displays the layout of the packet.

Using the VGA driver with tiled graphics, we use the X and Y coordinates to draw the specified sprite. This sprite is generally a group of similar tiles, or a collection of characters to be drawn. The VGA driver draws a tile based on a specified value from a colour palette, which is basically a collection of bitmaps of various configurations. It also draws all of this to a screen buffer, which is constantly being output to the screen. In our particular game, whenever we receive a packet, and it specifies a different image than the one currently on the screen, we clear the area that the sprite will be drawn and then redraw the sprite. By clearing only when there is a change, the animation does not flicker, since it ensures that the majority of the time the image is solid.

5 Datasheet

5.1 Hardware Requirements

We will interface the phototransistors using the GPIO and PIO pins of the Atmel board. We are planning to use two groups of 12 phototransistor/IR emitter pairs.

The software requires about 80kB of space on the Atmel board. This was the number displayed when programming to flash on the Atmel board.

5.2 Parts List

- Parallax Propeller Board
    - Propeller board power supply
    - VGA cable
    - USB-A to male mini USB-B connector
- Atmel AT91EB55 Board
    - Atmel board power supply
    - 2 x 20 pin ribbon cables
    - 1 x 26 pin ribbon cable
    - Bufferboard
- 24 x 950nm IR Emitter (12 for each controller)
    - Part Number: TSAL4400
    - Digikey: 751-1201-ND
- 24 x Silicon NPN Phototransistor (12 for each controller)
    - Part Number: TEFT4300
    - Digikey 751-1041-ND
- Projector or other VGA display

5.3 Power Consumption

- For each controller, the 12 IR emitters are connected in parallel and run at 1.02A and 1.5V.
    - Maximum Power Consumption = IV = (1.02A)(1.5V) = 1.53W
    - Average Power Consumption = IV = (0.8A)(1.5V) = 1.2W
        - 0.8A was the the typical current when running the controller.
    - Idle Power Consumption = N/A
        - The controller does not have an idle state.

- For each controller, the 12 phototransistors (each with a 120kohm resistor connected in series) are connected in parallel and run at 1.20A and 3.3V.
  - Maximum Power Consumption = IV = (1.20A)(3.3V) = 3.96W
  - Average Power Consumption = IV = (1.20A)(3.3V) = 3.96W
    - The phototransistor and the accompanying resistor consume all 1.20A.
  - Idle Power Consumption = N/A
    - The controller does not have an idle state.

5.4 Input Output Table

The propeller board only has a receive and a transmit, so using USART0 or USART1 on the Atmel board is not needed. Instead USART2, which only requires two pins on the bufferboard, is used.

Propeller Board	Bufferboard
Serial TX – Pin 13	RXD2 – JPCOMM Pin 14
Serial RX – Pin 12	TXD2 – JPCOMM Pin 11

Each of the phtotransistors needs to be hooked up to the Atmel board's PIO. These choices for the phototransistors were made for the following reasons:

- Ease of wire wrapping
- Making the controllers mutually exclusive (in the sense that controller 1 only used JPCOMM and JPTIMER and controller 2 only used JPAD)
- Making room for the RXD2 and TXD2 pins needed for the serial connection

Controller 1 Phototransistors	Bufferboard Pin
Phototransistor 1 (Right most)	PA0
Phototransistor 2	PA2
Phototransistor 3	PA1
Phototransistor 4	PA4
Phototransistor 5	PA3
Phototransistor 6	PA6
Phototransistor 7	PA5
Phototransistor 8	PA8

Phototransistor 9	PA25
Phototransistor 10	PA24
Phototransistor 11	PA23
Phototransistor 12 (Left most)	PA20

Controller 2 Phototransistors	Bufferboard Pin
Phototransistor 1 (Left most)	PB7
Phototransistor 2	PB2
Phototransistor 3	PB1
Phototransistor 4	PB0
Phototransistor 5	PB5
Phototransistor 6	PB4
Phototransistor 7	PB3
Phototransistor 8	PA13
Phototransistor 9	PA12
Phototransistor 10	PA11
Phototransistor 11	PA10
Phototransistor 12 (Right most)	PB16

6 Software Design

Please refer to the appendix for the software hierarchy diagram and the control flow diagram.

The four push buttons on the Atmel board are used for various functions. Push button 4 is used to pause/unpause the game. Push button 3 is used to reset the scores to 0. Push button 2 is used to enable/disable AI for player 1. Push button 1 is used to enable/disable AI for player 2. These push buttons were polled because the code was running at sufficient speed to detect the button presses. An artificial delay had to be added so that the polls did not happen too fast. If a button press was detected. This artificial delay was 10 runs of the main loop.

The proximity sensors are also polled. During each run of the main loop, the corresponding PIO lines of each controller are polled. For controller 1, the left most triggered sensor (corresponding to the bottom most location on the screen) takes precedence. For controller 2, the right most triggered sensor (corresponding to the bottom most location on the screen) takes precedence. This allows multiple triggered sensors to cause the paddle to be drawn in only one location. Once again, polling was done because the code was running at sufficient speed.

Each iteration of the main loop would send the following information to the propeller board:

- A packet containing information about the location of player 1's paddle.
- A packet containing information about the location of player 2's paddle.
- A packet containing the tens digit and ones digit of player 1's score.
- A packet containing the tens digit and ones digit of player 2's score.
- 1 to 20 packets containing the location of the ball. This was done to alter the speed of the game. With consecutive hits on the paddles, the game would speed up by sending fewer and fewer packets of the ball

The packets were of the following format:

start identifier	1 byte
spriteID	1 byte
x-coord	1 byte
y-coord	1 byte
stop identifier	1 byte

The start identifier marks the beginning of a new packet. This will be used by the propeller board to determine where to start parsing for drawing information.

The UART driver on the Proto board will store any bytes it receives in a receive buffer. With our sequential process board, we grab a byte from the top of that buffer, parse it, then grab the next byte. This allows us to not worry about parsing it as it comes in. If we had a problem with buffer overflow, we had the option of enlarging the receive buffer, or changing the baud rate.

The sprite ID tells the Proto Board which sprite it is going to redraw, such as the left paddle or the ball. The next 2 bytes contain the X and Y coordinates of the top left corner of where the new sprite is to be drawn. In the case of the score, these two byte are the ASCII code for the digits of a player's score. Lastly, the stop identifier is used to mark the end of the packet. There is also a special packet for the pause screen. It contains a 1 or a 0 in the x-coord location to tell the propeller board whether to write the word pause on the screen or to remove the word pause.

With this information, we instruct the VGA driver to draw a particular tile in a particular location. These locations are simply addresses in the array that acts as the screen buffer. The screen buffer itself holds tile values, which say what should be drawn based on their position in the screen buffer.

Depending on the sprite ID, we might draw the paddle colour, the ball colour, or a character. Also depending on the sprite ID are the group of tiles that are drawn and the characters that are written. (Score, pause)

The X and Y bytes will generally denote the X and Y positions for the sprite, but the score and the pause packets are different. As we know the position of the characters written will be static, we hard code those coordinates. For the case of the score, we split it up into two packets, player one's score and player two's score. Since we display the score as two characters, X and Y contain those characters. For the case of the pause packet, 1 denotes that the string Pause should be written, and 2 denotes that the string Pause should be cleared off the screen.

The AI of the paddles is simply the paddles following the y component of the ball, so it is unbeatable. However, having both AI's on at the same time is a good way to display the maximum speed of the game.

7 Test Plan

As our design project consists of several parts that do not necessarily depend on each other to function, we can separate the testing into 6 stages. Further stages after that are optional.

7.1 Motion sensors

- As the IR emitter produces an output that is not discernable by the human eye, we will be using our cellphone cameras to detect infrared light.
- We can then individually test a prototype circuit that includes the phototransistor. When we are certain that the circuit was built correctly, we will replicate this circuit for every phototransistor-IR emitter pair we require.
- After getting one pair to work, we can extend to 3 or 4 connected in parallel, then all 12 connected in parallel.
- We can stress test the IR emitters to see how long they would last.

7.2 Sensor Signals to ARM board

- After testing the motion sensors from above, they can now be hooked up to the Atmel board and tested on there. A printf statement is used to tell which sensors are currently triggered.
- This is to be done for both controllers.

7.3 Generating graphics on the Propeller Board

- As we were given a VGA driver and plenty of sample code utilizing the driver, first we run some sample code to ensure that our hardware is connected properly.
- After that, we slowly remove items from the sample code until we have working skeleton code that still makes a VGA output, and draws a blank background.

- We can make some variables, and then assign values to those variables before we use them. These variables will basically be placeholders for the X and Y coordinates we will be getting from the packets. If we can draw a tile based on these coordinates, then if we should be able to draw based on new X and Y coordinates.
- After the previous step, we should be able to draw larger images, such as paddles, with a repeat loop. Testing this shows understanding of the screen buffer, and how tiles are organized within it.
- Drawing characters: the VGA demo code has a function that implements this. We can graft that code into ours, and see if we have integrated it properly by drawing one character.

7.4 Communicating graphics between ARM and Propeller Board

- We require the packet in the correct format, so to ensure the ARM board is sending the correct packet, we use hyperterminal, a serial communication program, to display what the ARM is sending in a terminal.
- To determine what the Proto Board is receiving, we can feed the input of the serial receive data line to the output of the programmable pins on the Proto Board.
  The programmable pins on the Proto Board will output serial data to the Parallax Serial Terminal. So if we receive a byte from the Atmel board, we can transmit it to the workstation, and the Parallax Serial Terminal using the programmable pins. (30 and 31)
  Doing this, we can see how the Protoboard is interpreting the data from the Atmel board, and if the output matches the correct packet signal, then our boards are communicating correctly.
- We can test if we are parsing the X and Y coordinates correctly if we tell the board to draw at the locations we receive.
- We can test if we are parsing the sprite ID correctly if we can make the Protoboard behave differently based on the sprite ID. (Draw a pong as opposed to drawing a paddle.)
- We can test if it is parsing the stop sprite correctly by intentionally sending the wrong stop byte through the UART. If the behaviour stops working after this change, (assuming you program it to execute the based on sprite ID if the stop byte is correct), then the Protoboard now requires that the stop byte is correct. We can test the start byte in this same fashion as well.

7.5 Creating the game

- Test that the sending of a packet for each of the sprites works properly. This can be easily verified when the graphics is done.
- Test that the ball moves the full range of the screen from top to bottom.
- Test that the paddles move the full range of the screen from top to bottom, first without the sensors, then with the sensors.
- Test the collision detection between the paddles and ball.

- Test that the ball overlapping the score will not erase it.
- Test that the pause, reset score, and player AI buttons work properly.

8 Results of Experiments and Characterization

8.1 Motion sensors

We attempted to run a direct current through the phototransistor and the IR emitter, using the forward current/voltage settings specified in the datasheet. These values were low and in the range of 100mA and 1.3V. Finding that the performance of the motion sensor pair was lacking (giving us only ~5 cm), we tried to pulse the signal the IR emitter was receiving, giving it large bursts of power, then time to dissipate that power. This increased the range by about 3 cm, but not by a significant enough amount. We settled on pulsing the IR emitter at 200mA, with a pulse duration of 100 microseconds, a voltage of 2.3V, and a period of 10 milliseconds. This is likely to change when we have modified the circuit to make the phototransistor more sensitive.

However, in trying to design an op amp for the phototransistor, the desired output and range were not achieved, so the op amp was removed from the design. Also, the IR emitters were no longer pulsed in the end product, as we decided on just using our hands to control the paddles; the small increase in range did not matter.

8.2 Serial Communication

We decided that we did not need to have extremely stringent error checking for the purposes of this game. Therefore, we chose what we thought to be a relatively low baud rate, so as to reduce noise affecting our packets. We settled on 38400 as it was acceptable for our purposes, and we found no difference in performance if we went to a higher baud rate (57600).

It is notable that our packets do not have a transmission number. The reason being is that if a packet is dropped, it may cause a very momentary glitch in the image, but it is not very noticeable as the next packet would update that sprite to a new location very quickly anyway. It would only be noticeable if we had chronic corruption in our serial line, which emphasizes the importance of a lower baud rate.

8.3 VGA output

With the Atmel sending constant updates to the sprite, we found that if we indiscriminately cleared and then redrew an updated sprite, we would get a flickering image for stationary objects. We found that it did not take too much overhead to test if the sprite being updated was going to be in a different position. Therefore, if it was in the same position, we simply did not redraw the sprite. This way, we removed the flickering images from the game.

8.4 Flashing the Atmel Board

We ran into trouble flashing the Atmel board. The flashing would say it successfully completed, but it would not actually get into the main code. This was determined by having code that would turn off all the LEDs as soon as the main code was reached. In order to fix this, the uC library was removed and the thumb code was disabled for the two remaining libraries and doing a clean build from the top. Thanks to Nancy for finding this out for us!

9 Appendix

9.1 Quick Start Manual

9.1.1 Setting up the Controllers

- The IR emitters and photodiodes are to be wired up in parallel.
- Each of the photodiodes are to be wired up to the I/O lines as specified in the datasheet section.

9.1.2 Setting up the Atmel board

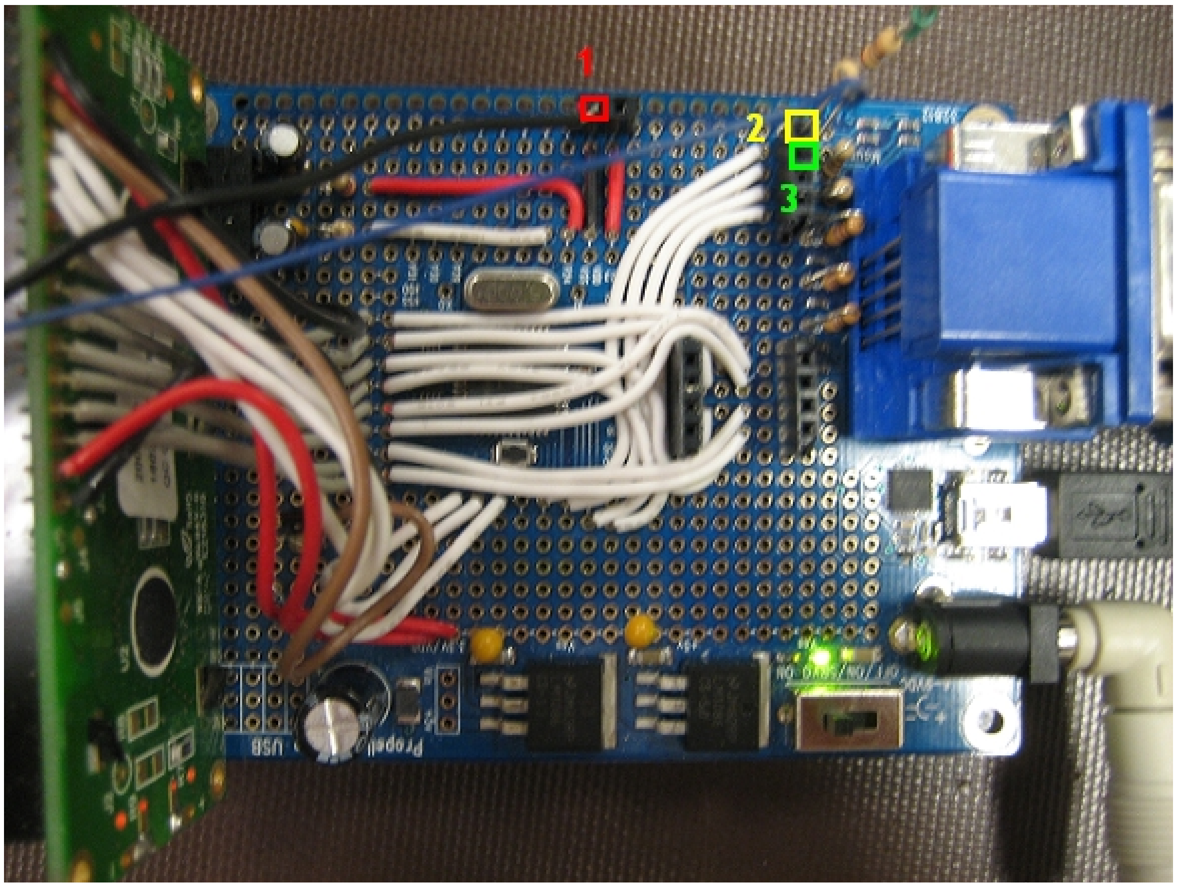
- Build the project directory as found at http://www.ece.ualberta.ca/~CMPE490/winter2010/lab/lab2_tutorial/index.html
- Download the code for the Atmel board at:

  http://www.ece.ualberta.ca/~elliott/cmpe490/projects/2010w/g1_prox_sensor/sourceCode/propellerBoardCode.7z
- Add these to the ram gpj file. You should now be able to build the project. Use ribbon cables to connect the first controller to JPCOMM and JPTIMER and the second controller to JPAD.

9.1.3 Setting up the Propeller Board

- Download the code found at:

  http://www.ece.ualberta.ca/~elliott/cmpe490/projects/2010w/g1_prox_sensor/sourceCode/prope.7z
- You need a USB to mini USB connector to download code onto the propeller board. Once you flash the propeller board, you can remove the connector.
- The pins on the Atmel board will need to be connected to the following locations on the propeller board, where red (1) represents ground, yellow (2) represents the RX on the propeller board, and green (3) represents TX on the propeller board:



9.1.4 Setting up the VGA display

- Find a VGA cable and a capable VGA display.
- Hook up the propeller board to the display.
- Turn on the display, then the propeller board, then the Atmel board.

9.2 Future Work

- We could interface another method of input, such as a RF apparatus to use as a button. The button could be used for firing or pausing.
- We may attempt to implement a 2D motion sensor array, or configuration. Since we would ideally attempt to make two motion sensor panels, should be able to reposition the panels, and set a different mode in order to track 2D movement in a grid area. From there, we could develop another game, that takes advantage of this movement, like Snake. Or Bomberman!
- Using a speaker to play some sound effects for the game.
- Using a microphone to force users to make a minimum volume of sound while moving, otherwise the motion detectors will not respond to their movement.

- The game could track when the user enters and leaves the play area, for the purposes of starting a game, and pausing a game. If there is no current game, and the play area is initially empty, when the user enters the play area the game starts. If he/she leaves while the game is still going, the game pauses, and waits for the user to return.

9.3 Citations

[1] AT91M55800A Reference
http://www.ece.ualberta.ca/%7ECMPE490/documents/atmel/AT91M55800AComplete.pdf

[2] Bufferboard Schematic
http://www.ece.ualberta.ca/%7ECMPE490/documents/BufferPinSchematic.pdf

[3] Silicon NPN Phototransistor Datasheet
http://www.vishay.com/docs/81549/teft4300.pdf

[4] Infrared Emitting Diode Datasheet
http://www.vishay.com/docs/81006/81006.pdf

[5]Virtex II Pro Documentation
http://www.xilinx.com/univ/xupv2p.html

[6]Virtex II Pro User Guide
http://www.stanford.edu/class/ee109/Docs/HardwareSpecs/VirtexII-Pro_UserGuide.pdf

[7]Pong Game
http://www.xnet.se/javaTest/jPong/jPong.html

[8]Space Invaders Game
http://www.spaceinvaders.de/

[9]Nancy Minderman and Dr. Elliott for all their input and suggestions

[10] Project Requirements Webpage
http://www.ece.ualberta.ca/%7Eelliott/cmpe490/projectRequirements.html

[11]VGA code
http://embedded.olin.edu/xilinx_docs/projects/bitvga-v2p.php

[12]Willowglen Inc. for lending us this awesome document template
http://www.willowglen.ca

[13]Schematic documentation
http://www.xilinx.com/univ/XUPV2P/Documentation/EXTERNAL_REV_C_SCHEMATICS.pdf

[14] Propeller Tool, including drivers and sample code
http://www.parallax.com/ProductInfo/Microcontrollers/PropellerGeneralInformation/PropellerMediaPage/tabid/832/Default.aspx

[15] Serial port code for the FPGA
http://www.fpga4fun.com/SerialInterface.html

9.4 Evaluation of the FPGA

During the majority of our development time, we attempted to use a Field Programmable Gate Array to handle the graphics portion of our project. Originally, the power and the availability of the FPGA seemed to make it an acceptable choice, even though we were warned it would require a lot of time to make it functional. The FPGA turned out to be an inefficient use of time, for several reasons.

The FPGA is too precise. It has the ability to do a great many things, but in order to distinguish between these things it requires a very specific set of instructions. Considering that the timing of any processes specified need to be considered, and that everything is on an extremely basic/fundamental level, accomplishing very simple tasks require a large amount of time and effort to complete.

Another aspect that was very time-consuming was the compile time. Now, some of the problems regarding this can be attributed to the environment that we developed on, and not the FPGA or its associated tools. The ECE admins require that anything that goes over the network needs to be scanned, and checked for viruses. So all the files that the ISE project navigator made (log files, reports, etc. from synthesis and translating) were scanned. This slowed down the compilation process by a fair amount. Turning off the additional FSecure virus scanning would reduce compile times by 5-7 minutes.

However, the compile times for the FPGA were still significant, even without the virus scanning. We estimate that the average compile time was around 15-20 minutes. Part of the large compile times were due to the large screen buffer array that was used to draw the screen, however this was unavoidable for our project. We estimate that the synthesis of the large 2D register increased the compile time by about 8 minutes. This indicates to us that for our particular project, the FPGA is not a productive or efficient tool to use.

Finally, one of the largest problems we had with the FPGA were efficient testing methods. Originally, since we were developing using Verilog, we thought we could not use Modelsim (the simulator tool for the FPGA) to test our code and to verify that it functioned in the manner we intended. The error message at the time indicated that libraries were missing. As a result, we had to compile and test our results directly. This was sufficient for ensuring that the upper module that generated/drew the screen worked correctly. While it was time-consuming, having the VGA output to a monitor provided enough information to debug our code.

However, when we created a module to parse the packet and instruct the upper module to draw a sprite, we encountered more difficulty in discerning what was actually occurring within the code. We eventually resorted to mapping various signals and registers to the LED's on the FPGA, and running the code very slowly so human eyes could tell what was occurring. At this point, we hit a wall with the code, as the results of our testing indicated that we were outputting the X, Y, and SpriteID data correctly from the parser module in the same manner as the test module that did draw sprites on the screen as intended. Unable to discern the difference between the two signals, we went tried to make Modelsim work. It turned out that the ECE administration had a program called Steady State in effect, and that this program was locking out our access to the libraries we needed. Our lab instructor helped us rectify this, and after figuring out that Modelsim was pointed to the wrong library locations, we were able to actually simulate our code.

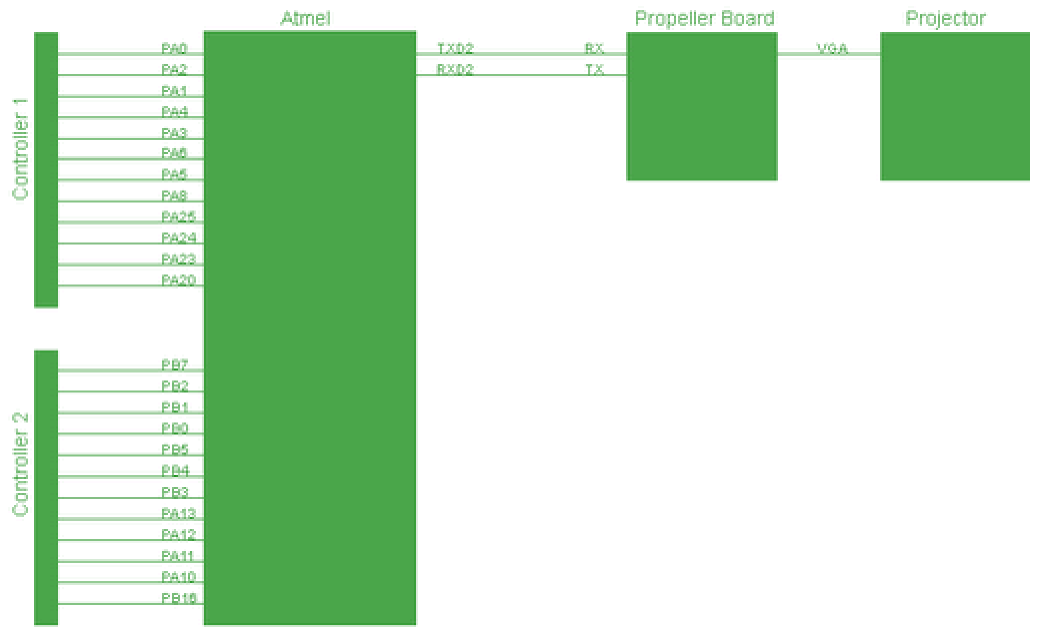
The simulations indicated that the state machine we made to parse the packet was incorrectly passing every other packet. After changing this, and struggling with the FPGA some more, and getting nowhere, we went back to making compiles to verify the ModelSim results. After having wasted a lot of time with ModelSim, it turned out that registers that were said to be changing in the simulation were not actually changing on the FPGA. (Side note: We spent a lot of time verifying that the connections between the modules in the testbench were the same as in the actual programming file. So we are fairly certain that ModelSim was incorrect.)

The current iteration of the project code, after having made changes with ModelSim, does not correctly take in bytes and place them into the X, Y, and spriteID registers for output out of the module. Having lost progress, we gave up on the FPGA in favour of the Proto Board. It turned out the Proto Board had the right amount of complexity for the project, and we were able to implement a working version within a few hours.

In summary, the FPGA turned out to be too complex and time-consuming for the simple role we needed it to fulfill. As such, it was not the appropriate tool, and detracted from the final functionality we could have had.

9.5 Component Hierarchy

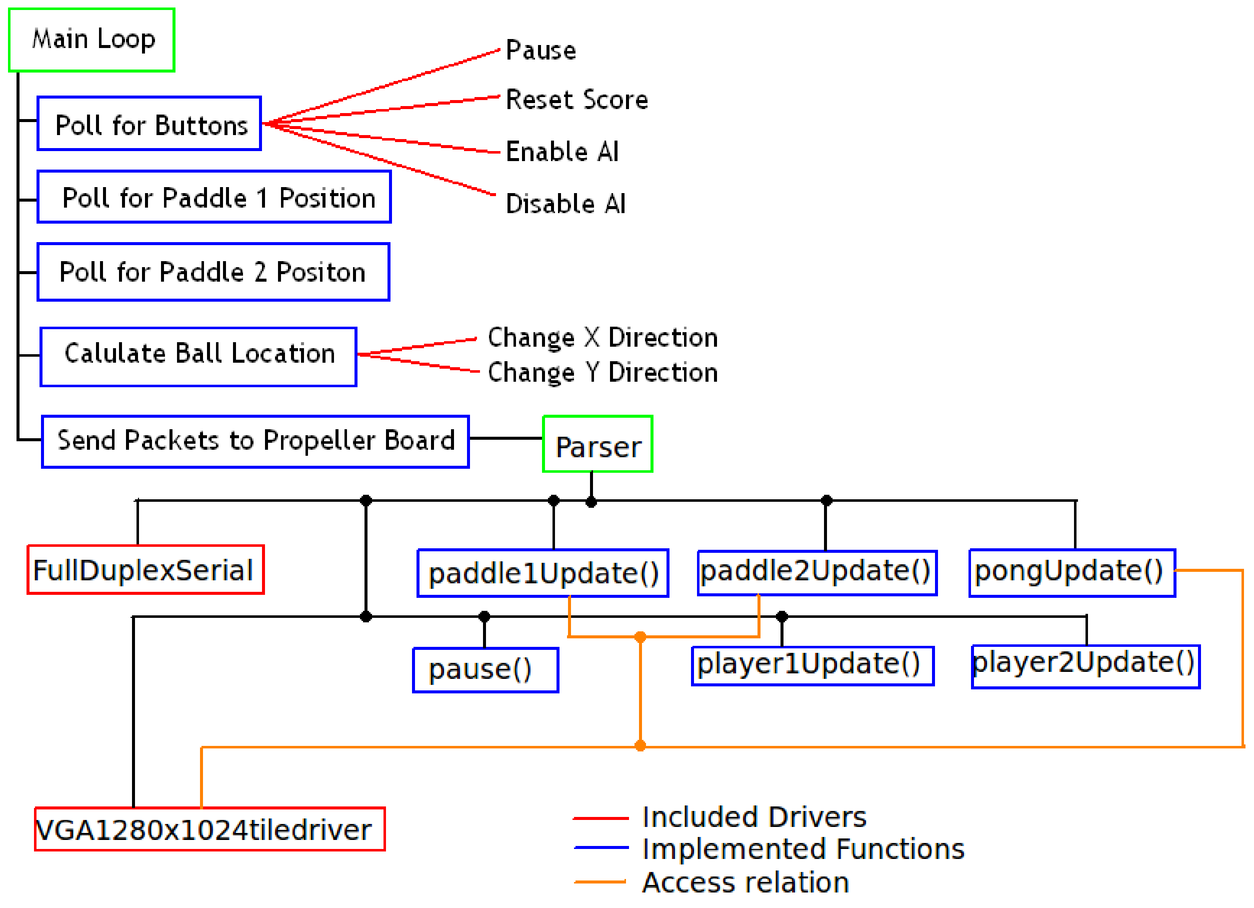
9.5.1 Full Schematic



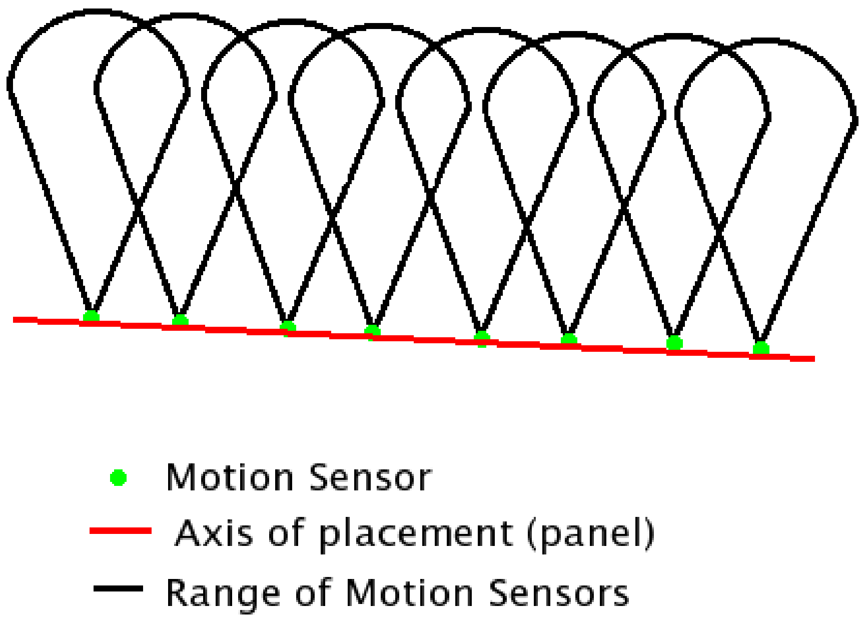
9.5.2 FPGA Schematics

Refer to the FPGA schematics released by Xilinx, which can be found on page 30 to 32 of the schematics documentation [13].

9.5.3 Graphical Hierarchy



9.5.4 Sensor Apparatus



9.6 Source Code

The status of each of file can be one of the following:

- **[N]**ot compiled successfully
- **[C]**ompiled without errors
- **[E]**xecuted or otherwise demonstrated
- **[T]**ested and passed

9.6.1 Motion Sensors

- **[T]** motionSensors.c – Contains code for polling the on-board push buttons, polling for the sensors, and sending packets to the Propeller board.
- **[T]** motionSensors.h – The header file containing function declarations and #defines
- **[T]** enable_interrupts.arm – Enables interrupts when downloading code to ram using the slingshot.

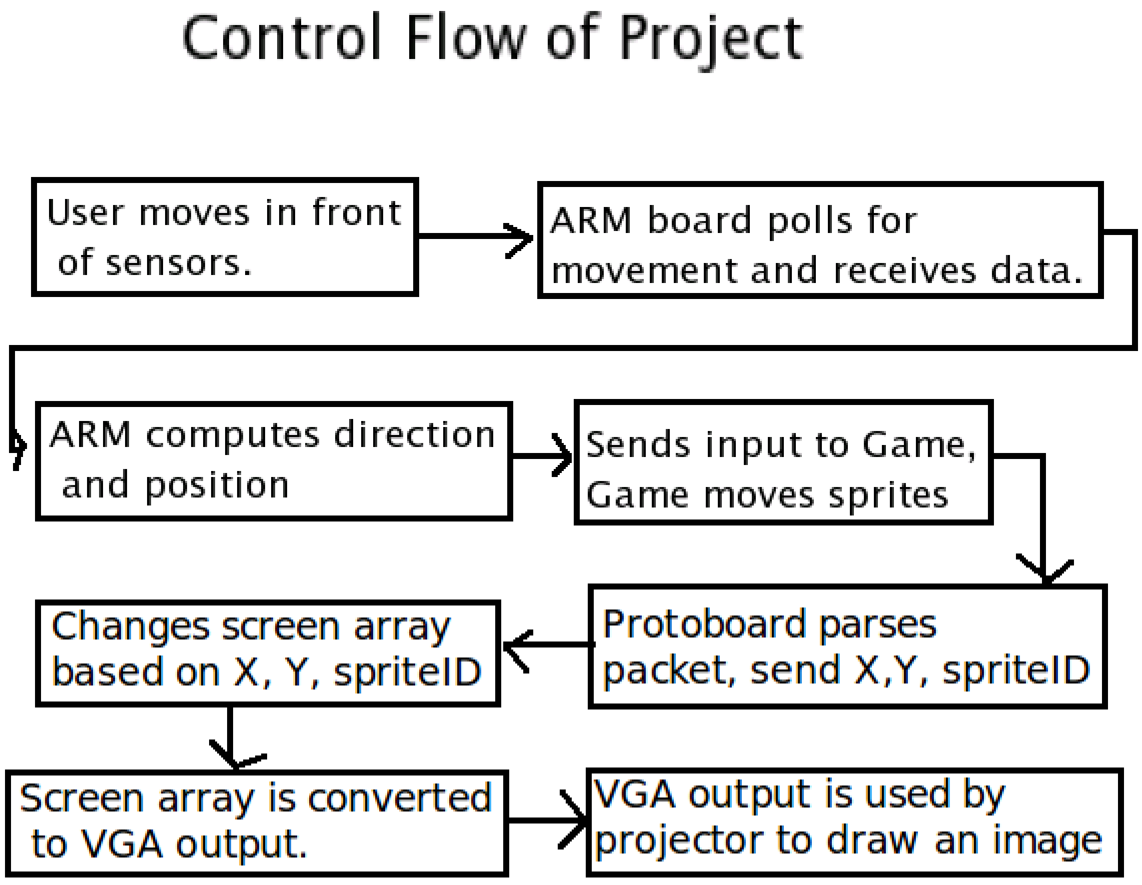
9.6.2 Propeller Board

- **[T]** Parser.spin – Parses the packet received from the Atmel board and draws the sprite on the screen
- **[T]** TestParser.spin – Tested the serial between the Atmel board and propeller board. Everything sent to the propeller board was echoed to the Parallax terminal.

9.6.3 FPGA

- **[T]** async_receiver.v – Serial receiver.
- **[T]** async_transmitter.v – Serial transmitter.
- **[T]** CLOCK_GEN.v – Generates a buffered clock and a pixel clock.
- **[T]** COLOR_BARS.v – Creates colored bars on a screen, one of them changing.
- **[E]** MAIN.v – Toplevel module that connects all the modules together.
- **[T]** SCREEN_BUFFER.v – Draws sprites onto the screen.
- **[T]** SVGA_DEFINES.v – Defines values for different video settings
- **[T]** SVGA_TIMING_GENERATION.v – Produces a v_sync and h_sync for different video settings.
- **[C]** S_RECEIVER.v – This is supposed to parse the packet, and return X,Y and spriteID.
- **[T]** S_receiver_TB.v – Testbench for S_RECEIVER.v
- **[T]** TestXYSprite.v – Test module that sends different X, Y, and spriteID signals. Meant to test the functionality of SCREEN_BUFFER.v.
- **[T]** TestXYSprite_TB.v – Testbench for TestXYSprite_TB.v
- **[T]** TEST_RX.v – Test module that returns different packets to S_receiver. Meant to test S_receiver's ability to parse packets.
- **[T]** TEST_RX_tb.v – Testbench for TEST_RX.v
- **[T]** UART.v – Takes the async_receiver's output, and clocks it out as a byte.
- **[T]** VIDEO_OUT.v – VGA driver for the FPGA.
- **[T]** VIDEO_RAM.v – RAM that stores colour data for the VGA driver whenever it requires it.

9.7 Control Flow


Control Flow of Project
User moves in front of sensors.
ARM board polls for movement and receives data.
ARM computes direction and position
Sends input to Game, Game moves sprites
Protoboard parses packet, send X,Y, spriteID
Changes screen array based on X, Y, spriteID
Screen array is converted to VGA output.
VGA output is used by projector to draw an image