Tim Tregubov and Divya Gunasekaran

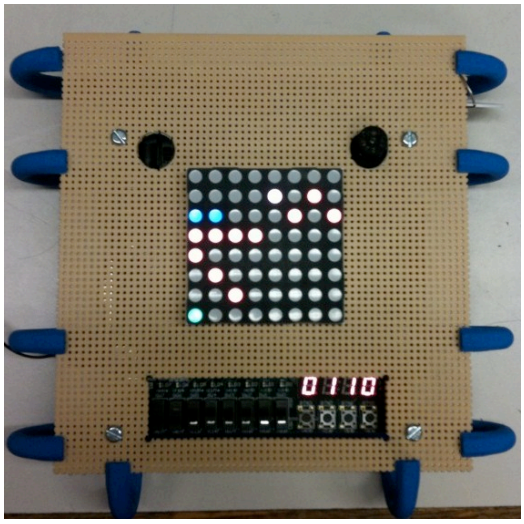
ENGS 31

Final Project

September 1, 2009

TINZ

This Is Not Zelda



Abstract

Our goal for this project was to create a simple, but addicting game that played off of old-fashioned handheld and vintage arcade games, but utilized newer technology that allowed for a more modern feel and the implementation of more exciting features. We succeeded in creating a challenging game that takes the user's movements as inputs via an accelerometer to move the player on the board, which consists of a small 8x8 RGB LED matrix with a seven-segment display showing the timer and score. A piezo is also included to produce different sounds for specific actions in the game. These components contribute to a gaming experience that is stimulating both tactilely, visually and aurally. The game design also makes the game easy to reprogram to include more levels and additional features.

Table of Contents

Section 1: Introduction

Taking the idea of a basic Tilt-A-Maze and combining it with the addictiveness of a classic arcade game like Pac-Man or Frogger, we created a challenging handheld game that has simple visuals, but requires logic, planning and cool nerves on the part of the player.

The goal of TINZ is to evade the enemies in order to get from the start to the finish on the board in the shortest amount of turns and before the timer runs out. Power-ups also exist and when obtained, make the player invincible, allowing contact with an enemy. However, there is a caveat – used power-ups turn into "heat-seeking ghosts" that follow the player and can cause the player to lose if it catches up to the player.

The game console consists of an 8x8 RGB dot matrix of LED lights mounted on top of a rectangular piece of perfboard that covers the FPGA board. The visuals are the various colored lights on the matrix. The player is represented by a green dot, enemies are represented by red, orange and yellow dots, power-ups are represented by blue dots, heat-seekers are represented by bright pink dots and the finish position is represented by a pinkish white dot. To move from position to position, the player must tilt the board in the direction he wishes to move. The timer and score are displayed on the seven-segment display located on the FPGA board, over which the game board is mounted. Sounds are also produced for certain actions during the game.

Game play involves four different levels, each with its own theme (dragon, jungle, volcano and space) and each more difficult than the one before it. Visual and auditory sequences have been set up for the game intro when the game is first turned on, for when the player loses a level and for when the player wins a level. Losing a level resets the game to the current level. Upon winning a level, the game advances to the next level. Each time the player wins all four levels, the game resets to the first level, but with a higher difficulty setting – the accelerometer axes are reversed, swapped or both. There are also switches that allow the user to override the current level of difficulty and choose the difficulty setting. This increases the entertainment value of the game for the same four levels.

Section 2.1: Specifications

The circuit has a power source, power button, sound enable switch, level difficulty switches, level difficulty override switch and movement detected by an accelerometer for inputs. Outputs include an 8x8 RGB dot matrix of LED lights, sound via a piezo, and a seven-segment display (see Figure 2.1 below for location of inputs and outputs).

The power button turns on and starts the game. The dual-axis accelerometer detects movement along the x- and y-axes and feeds these into the *Accelerometer* module. The *ACCELDecoder* module filters these signals and conveys them to the *Player* module, which keeps track of the player's position. Thus, movement detected by the accelerometer is used to move the player on the board. The sound switch enables sound when turned on, which is handled by the *Noises* module and output to the piezo. The level difficulty override switch, when on, allows the player to choose the difficulty setting for any level. The two level difficulty switches are used to actually change the difficulty setting (when the level difficulty override switch is on) and choose between four different difficulty settings.

The game board is actually the LED dot matrix, which receives data about which colors to display in which locations by the *Display* and *LEDDriver* modules. The seven-segment display shows the timer and score and is controlled by the *GameTimer* module.

Please see Appendix A.2 for functional block diagram and Appendix A.5 for a more detailed list of the off-board parts used to make this circuit.

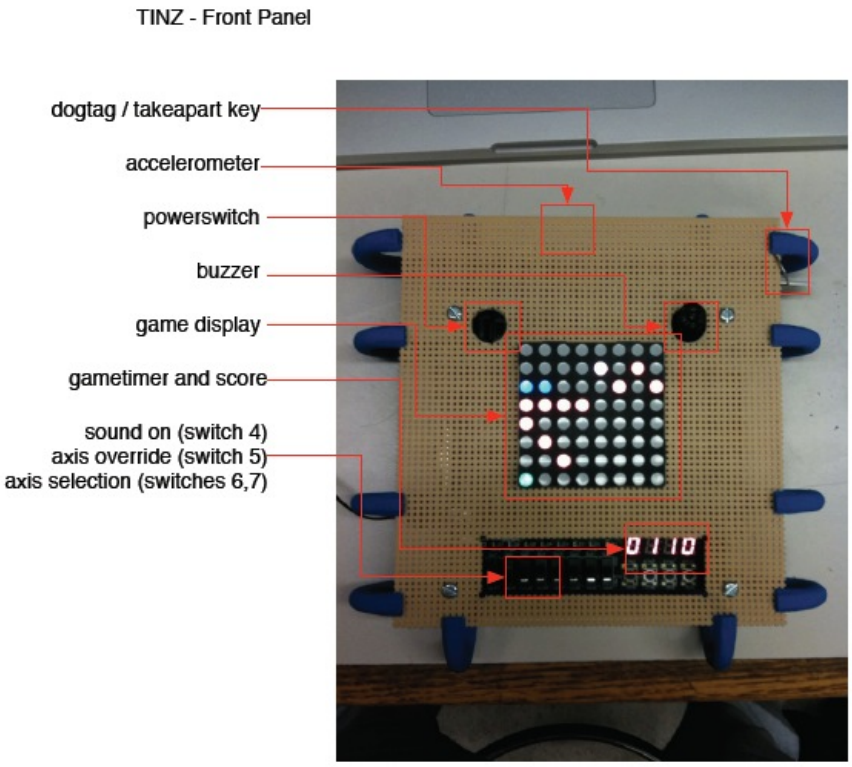

TINZ - Front Panel
dogtag / takeapart key
accelerometer
powerswitch
buzzer
game display
gametimer and score
sound on (switch 4)
axis override (switch 5)
axis selection (switches 6,7)

Figure 2.1

Section 2.2: Operating Instructions

The output pins on the LED backpack need to be wire-wrapped and connected to pins JA2 to JA6 on the FPGA board. The pins soldered onto the accelerometer need to connect to JB7 through JB12 on the FGPA. The ground pin from the piezo needs to be connected to the JC ground pin, while the input pin for the piezo must be wire-wrapped to pin JC10 on the FGPA (see Figure 2.3 below). For a cleaner presentation, the LED matrix can be mounted on a piece of perfboard, which can be placed on top of the FPGA board using standoffs.

The game can be programmed onto the FPGA's onboard ROM. Therefore, all that is needed to run the game is to connect a power source to the FPGA (either by connecting the USB port to a computer or using a wall power cord) and turn on the power switch for the FPGA. Sounds can be disabled by turning off switch four, the difficulty level can be overridden by turning on switch 5, and the difficulty setting can be changed with switches 6 and 7 on the FPGA (see Figure 2.1 in the previous section or Appendix A.1).

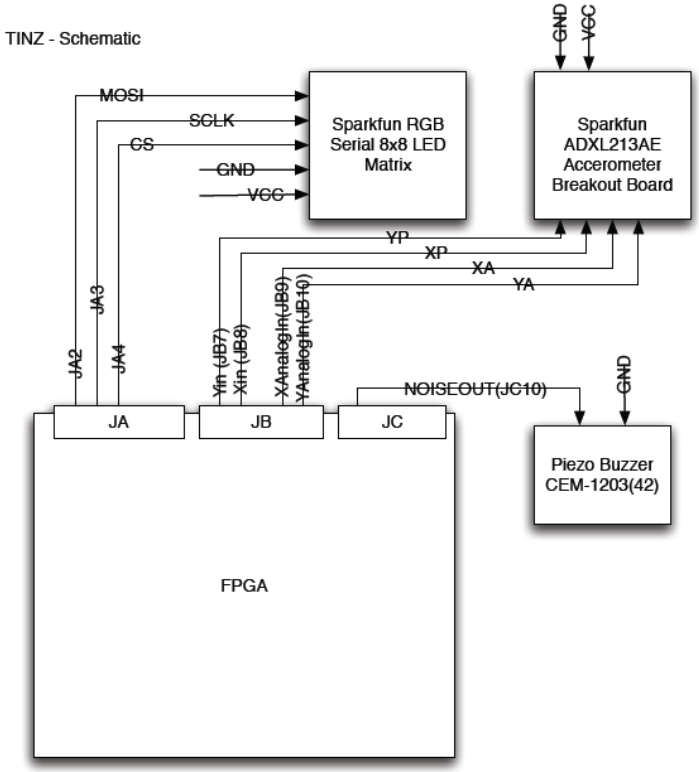


Figure 2.2

Section 2.3: Theory of Operation

Figure 2.3

- **cellGame:**

This is the top level module and it puts all the rest of the following together. It is purely structural and does nothing but connect appropriate signals between high-level modules.

- **MainController**:

MainController is the finite state machine that acts as the controller for the entire game. Upon startup of the game, MainController resets all of the sequences (that is, the "Intro," "Win" and "Death" sequences) and then sends a signal to the *Display* module to display the "Intro" sequence. The controller continues to display this sequence until receiving an active high pulse from *Sequences* indicating that the intro sequence is done. At this point, the controller sends reset signals to the *GameTimer*, *Sequences* and *Player* modules to set the signals in these modules to their initial values. The controller also sets the sevenSegEN signal, which outputs to the *GameTimer* module, to '1' in order to enable the seven segment display on the FPGA board. On the next clock cycle, the controller allows for normal play. The displaySelector, which outputs to the *Display* module, takes on its default value and displays frames from *GameboardROM*. The sevenSegEN (to *GameTimer*), moveEN (to *Player*), displayEN (to *Display*), gameLogicEN (to *GameLogicFSM*) and soundEN (to *Noises*) signals are set high so that the seven segment display shows the timer, the player is allowed to move, the LED matrix cycles through the frames stored in *GameboardROM*, collisions between the player and other game components are evaluated, and sounds corresponding to player movements or certain collisions are enabled. This "Play" state is only exited when the death input for the controller is asserted high (driven from *GameTimer* or *GameLogicFSM*), in which case "StartDeath" is then entered, or if WIN is asserted (from *Gameboard*), in which case "StartWin" is entered. "StartWin" and "StartDeath" reset the win and death sequences and automatically transition to "WinDisplay" and "DeathDisplay" state, respectively. "WinDisplay" and "DeathDisplay" display the win and death sequences, respectively, while enabling sound. When seqDone is asserted, the transition to the "Waiting" state is made, where the last frame of the previous sequence is displayed, the seven segment display is enabled and sevenSegSelector is set to '1' so that the score is displayed and a gameResetTimer is enabled. When gameResetTimer reaches its maximum count, gameReset is asserted and the "Reset" state is entered, starting the game over again.

If the player won the last level, the level is incremented and the game resets to a new level; otherwise, the game resets to the current level. If the player wins the last level (level 4), the game

resets to level 1, but the difficulty is increased. An increase in difficulty entails a reassignment of the x and y-axes from the accelerometer (see *Play* module for more information).

- **GameLogicFSM**:

GameLogicFSM is the finite state machine used to check for collisions between the player and enemies or power-ups and update the player's current status based on these collisions. When logicEN (from *MainController*) is high, GameLogicFSM continually updates player status based on a two-bit collisionData signal from *Gameboard*. When the player is in the "Unshielded" state, the player will remain in that state if the most significant bit of collisionData is zero, which signifies an unoccupied location. If collision data is "10," the player has encountered a power-up and will move to the "GetShield" state. If collisionData is "11," the player has hit an enemy and will transition to the "DeathState." In the "GetShield" state, a disablePU signal is sent to *Gameboard* to disable the used up power-up, soundSelect is set to '1' and makeSoundLogic is set to "010" for the *Noises* module (see *Noises* for more information), playerColor is set to "01" for the *playerColor* module, which chooses the shielded color, and if the activeSeeker signal from the *Heatseeker* module is low (which means there are no active heatseekers), then initHeatSeeker is asserted and sent to the *Heatseeker* module to convert a used up power-up into a heatseeker (see *Heatseeker* for more information). The player then enters the "Shielded" state. In this state, a timer is enabled so that the player can only lose its shield to a power-up after a set amount of time; this prevents the player from automatically losing its shield immediately after disabling a power-up. The player only exits the "Shielded" state if it collides with an enemy or if it collides with an enabled power-up and the timer has reached its maximum count, thus asserting shieldDepleted. If the player collides with an enemy in the "Shielded" state, the player first enters "WaitBeforeLose," which allows the shieldTimer to reset, and then enters the "LoseToEnemy" state. A disablePU signal is sent to *Gameboard* because some enemies are capable of being disabled, playerColor is set to "10," which is the color signifying a loss of shield to an enemy, soundSelect and makeSoundLogic are set so that *Noises* plays the appropriate sound, and the shieldTimer is enabled. This allows the player to be invincible for a set amount of time. When shieldTimer reaches its maximum count and asserts shieldDepleted, the player then returns to "Unshielded." If the player

collides with a power-up while “Shielded,” the power-up is disabled (so disablePU is asserted and sent to *Gameboard*), soundSelect and makeSoundLogic are set so that *Noises* plays the appropriate sound, and then the player enters the “Unshielded” state on the next clock cycle. If GameLogicFSM receives a gameOver pulse from the *Play* module, which signifies that time ran out, at any time during the game, the player enters “DeathState.” This is a dead-end state in which soundSelect and makeSoundLogic are set so that *Noises* plays the appropriate sound for losing the game and death is set to ‘1.’ A monopulser is included in this module so that the death is converted from a level to a single pulse.

- **Sequences**:

  This module handles the ROM for the "Intro", "Win" and "Death" sequences. It's inputs are row and col (from *Display*) and seqreset (from *GameController*) and it outputs the appropriate colors (to *Display*) as well as a sequence done signal (to *GameController*). All it does is keep a frame counter and a time for setting how fast the framerate is.

  - **(DeathROM|IntroROM|WinROM)**:

    These 4 modules are single port ROM IP Cores. They store an 8 bit color, for each column, for each row, for each frame. The address bits are therefore: 4 bits for frame + 3 bits for row + 3 bits for column. These are loaded from:winseq.coe, introseq.coe, winseq.coe -- which are all created by genCOE.py in coloral mode.

- **Noises**:

  The *Noises* module acts as an interface between the piezo and the game. When soundEN (from *MainController*) is high, transitions in the finite state machine are enabled; otherwise, the FSM is restricted to the “Quiet” state in which no sound is produced. *Noises* receives the three-bit signals makeSoundLogic (from *GameLogicFSM*) and makeSoundMove (from *Play*) that dictate which state to enter and thus which frequency to output. The makeSoundLogic signals correspond to actions like obtaining a power-up, losing a power-up and losing the game. The makeSoundMove signals correspond to actions like moving freely on the board and trying to move in a direction that is not allowed (for example, trying to move left when already at the leftmost part of the board). Since events

handled in *GameLogicFSM* and *Play* can overlap, soundSelect is asserted anytime an event in *GameLogicFSM* occurs and noises will be produced for those events rather than events from *Play*. If winSound (from *MainController*) is asserted, a sequence of 19 states is entered, which produces the song played during the win sequence. In each state, the noiseType is declared, which corresponds to a certain frequency that then gets passed to the *Noise* module (see *Noise* for more information). Timers are also used for each state to determine the length of the sound for the action.

- **Noise**:

  This module generates square waves for making noise. It takes as input an on signal and a frequency and outputs a square wave of that frequency. It connects into the *Noises* module.

- **Play:**

This module simply connects the following submodules dealing with player and gametime.

- **ACCELDecoder:**

  This module connects several submodules dealing with the accelerometer input and also takes as input a level difficulty from gamelogic which it then uses to scramble the x/y directions for harder levels. It connects the below modules listed below in the order of appearance. This module's outputs are clock cycle pulses for movement in x/y plus/minus directions. (Notice the analog pin inputs, these take the analog pins and route them to bogus external pins. This is because the FPGA was crashing the accelerometer by sending bogus signals down the "unconnected" pins.)

  - **AccelDetector:**

    This module measures the PWM signal output from the accelerometer. It measures T1 (time up) and T2 total time for one up/down pulse. There exist more complicated ways of measuring but this simple way works for us. It returns two 8 bit values: T1, T2. It is instantiated for both X and Y.

  - **AccelDivision:**

    This is a division IP Core that takes the output of *AccelDetector* and divides T1 by T2 returning a fractional part used to determine movement. It is instantiated for both X and Y.

- **AccelFilter:**

  This module returns a simple 32 sample average on its input signal, it also takes a clock enable. This is to smooth out the accelerometer signal to get rid of jitter. It is instantiated for both X and Y.

- **ThresHysteresis:**

  This module takes the input 8 bit signal from *AccelFilter* and does a hysteresis threshold on it and detects a move either UP or DOWN and sets a pulse on the appropriate output. It also has the capability to change the repeat move speed by level. The constants here are responsible for the angle of tilt required to make a move and need tweaking. It is instantiated for both X and Y.

- **Player:**

This module receives xPlus, xMinus, yPlus and yMinus signals from the *ACCELDecoder* to increment or decrement the player's x and y position and keep track of the number of moves the player makes. It includes two instantiations of *positionCounter*, one to keep track of position on the x-axis and one to keep track of position along the y-axis. The player's x and y position are both output to *Gameboard* and *Display*. The Player module also has three instantiations of *moveCounter*, one for the ones digit for number of moves, another for the tens digit for number of moves and a third for the hundreds digits for number of moves, which are all output to the *GameTimer* module.

- **positionCounter:**

  The positionCounter module is used to keep track of the player's location along an axis. If set is asserted, the position is initialized to initValue (these inputs are not used under the *Player* module) and if rst is asserted, the position is reset to 0. This module receives UP and DOWN signals from the *Player* module in which it is instantiated. If an UP signal is received, the position count is incremented unless the count is already at 7, in which case the count remains at 7 (rolling over to 0 is not permitted). Similarly, if DOWN is received, the position count is decremented unless the count is already at 0, in which case the count remains at 0. This module also outputs a three-bit makeSoundMove signal to the *Player* module, which is then

connected to the *Noises* module. The makeSoundMove signal is assigned one value for when the position is incremented or decremented and a different value for when the position is given a signal to increment or decrement, but cannot because it is already at the upper or lower bound, respectively.

- **moveCounter:**

  The moveCounter is used to keep track of a single digit for the number of moves the player makes. Given a move signal from the *Player* module (which receives it from the *ACCELDecoder* module), the move count (output as moveCount to the *Player* module) is incremented. The count rolls over from 9 to 0 and asserts DOUT to the *Player* module in which it is instantiated to cue the next digit to increment.

- **GameTimer:**

  Depending on the sevenSegSelector input (from *MainController*), this module outputs either the game timer or the score to the seven segment display. It includes an instantiated *AnodeDriver* to allow for a multiplexed display and has three instantiations of *unitCounter*, one for the ones digits of the timer, another for the tens digit of the timer and a third for the hundreds digit of the timer. It has three inputs from the *Player* module representing the three digits of the score. It also has resetGameT and sevenSegEN inputs from *MainController* to reset the game timer and enable the seven segment display, respectively.

  - **unitCounter:**

    This module is used to keep track of a single digit for the game timer. When timerEN is high (from *MainController*), the count (output as currTime to *GameTimer*) decrements and asserts a DOUT signal (to *GameTimer*) when it rolls back from 0 to 9, indicating that the next digit should be decremented. It can be initialized to a specified value using GENERIC MAP.

  - **anodeDriver:**

    This module consists of a 2-bit counter and 2-to-4 decoder used to create a multiplexed display on the seven-segment display. When displayEN (from *MainController*) is high, the an output (to the FPGA board) cycles through the four digits of the seven segment display at a frequency that gives the impression that all four digits are being displayed simultaneously.

- **PlayerColor:**

  This module is responsible for the "breathing" color changing of the player color. It takes a selection input that selects between unshielded/shielded/on to of enemy and it runs through various colors at an internal framerate. The color constants for the player "dot" are defined here as is the framerate. All it returns is the appropriate player color for the "state" that the player is in.

- **Display:**

  This module is the interface between the ROMs containing color data for each location on the board and the LED driver that communicates to the LED backpack. It has a finite state machine to coordinate the exchange of data from the ROMs to the LED driver. If displayEN is high (from *MainController*), the finite state machine in Display cycles through a process of loading a byte of color data into a 64-byte shift register from either *Gameboard* or *Sequences* (depending on displaySelect input from *MainController*), incrementing location and loading a byte of color data for the next location into the 64-byte register. When the register is full, a displayReady signal is sent to *LEDDriver* and Display shifts bits of data from the 64-byte register contained in *shiftRegister* to *LEDDriver* when it issues a shiftToLED signal. When the *LEDDriver* asserts a displayDone signal, the Display module begins the process of loading the 64-byte register again.

  - **shiftRegisters:**

    This module contains a 64-byte right shift register that can also do an 8-bit parallel load. It is instantiated in *Display* and receives a dataInReady_tick from *Display* to load 8-bit data input funneled through *Display*. *LEDDriver* send a shiftOut signal through *Display* to right shift and return the least significant bit through the outBit output. This module also keeps track of number of the number of bits loaded into register and asserts regFilled output to *Display* when the register is full. If the reset input is asserted, the register and bit count are reset to zero.

- **LEDDriver:**

  This module does the SPI communication with the 8x8 RGB LED-Matrix Backpack from Sparkfun. It does the signals for MOSI/SCLK/CS for SPI. It waits for a GoDisplay signal from

Display and then will shift a bit onto the MOSI line at the appropriate time from a signal from the *Display* shift registers. Every time it shifts it sends out a shift signal to *Display* requesting the next bit. The datasheet for the backpack from sparkfun (see Appendix E) was helpful in creating this module, as was a Wikipedia page on SPI bus interfaces (see Section 6).

- **GameBoard**:

  This module is responsible for the gameboard layout. It connects *AttrLookup* with *GameBoardROM* and gives *Display* the appropriate color for a position and gives *GameLogicFSM* the appropriate collision data for another position (as these two run separately). It contains the frame counter for the gameboard and gives the right information for the current frame. Thus it is responsible for the gameboard layout and animation of everything but the player and the ghost powerup. It gets the row+col for collision from *Player* for sending the collision bits to *GameLogicFSM*, and another row+col from *Display* for setting the color for that position. It also outputs the "win" condition of a level as the finish position is just another "type" in the gameboard (enabling it to also be animated).

  - **AttrLookup**:

    This module handles looking up "type" attributes. The attributes are whether a type is enabled or disabled, color, and finish. This allows for a powerup to be "used up", for enemies to be disabled by certain logic, and for determining if a win condition occurred. This takes three "type" addresses for reading color/enabled bit and for write. If it gets a disable signal it will disable the "type" addressed by the write address. It will return the color of the "type" addressed by the read color address, and the enabled bit for the "type" addresses by the read enabled address. The enabled bit / read enabled signal are both connected to *GameLogicFSM*, while read color and the color bits are connected to *Display*, both via *GameBoard*. This module was simpler to write without using a ROM and so it is simply constants and flipflops for the various bits all hardcoded. For all practical purposes this is a relational lookup table for attributes connected to *GameBoardROM* by the "type" identifier.

  - **GameBoardROM:**

    This is an IP Core Dual Ported ROM module. It is 8bits wide by 4096 tall. The addressing works

like this: 2bits for level + 4 bits for frame + 3 bits for row + 3 bits for column. This is populated with gameboard.coe which is generated by genCOE.py in positional mode. The 8 bits stored represent the following: 1st msb is this position occupied or not, 2nd bit is whether it is a powerup (0) or an enemy (1), and the other 6 bits are a "type" identifier for that particular enemy or powerup. This ROM is not registered and the first read address is connected to *Display* and the second read address is connected to *GameLogicFSM* via *GameBoard*.

- **HeatSeaker:**

This module converts disabled power-ups into "heatseeking ghosts" that follow the player on the board and can kill the player. This module includes two instantiations of *positionCounter* and one of *clockDivider* to keep track of the heatseeker's x and y location and regular the speed of the heatseeker's movement, respectively. Heatseeker outputs an activeSeeker signal to *GameLogicFSM* indicating whether a heatseeker is currently enabled because there can only be one active heatseeker on the board at any given time. The Heatseeker module receives the player's x and y position from the *Player* module and initializes a heatseeker at the position where it receives an initHeatSeeker signal from *GameLogicFSM*. The activation of the heatseeker is delayed, however, to allow the player to gain some distance between itself and the heatseeker. Two processes – for the x- and y-axes – direct the movement of the heatseeker based on its position with respect to the player and send the appropriate signals to the *positionCounter* modules that keep track of the heatseeker's position. The heatseeker's x and y position and hardcoded color are output to the *Display* module. If the heatseeker's position and the player's position are equivalent, a seekerHit signal is asserted and output to *GameLogicFSM*; a monopulser is included in this module to convert the seekerHit signal from a level to a pulse.

  - **positionCounter**

    The positionCounter module is used to keep track of the heatseeker's position and is used in the same way that is described under the *Player* module.

  - **ClockDivider**

    The ClockDivider is used to slow down the heatseeker's movements so that it does not catch up to the player too quickly.

- **ClockDivider**:

  This is a generic little module that models clock dividers. These were frequently used in modules.

Section 2.4: Construction and Debugging

After the block diagram and modules for the game were designed, the modules were divided between both group members and the code was developed for each one. Before the off-board components like the accelerometer, LED matrix and piezo were obtained, the modules were tested by simulating them in Xilinx. Once the accelerometer and LED matrix were obtained, each was individually connected to the FPGA board and the *ACCELDecoder* and *LEDDriver* modules were tested. These modules were created specifically for the purposes of communicating with these external devices.

For the accelerometer, unused inputs began interfering with other inputs. Initially, we believed there was an error with the code or the accelerometer. After using an oscilloscope to look at the signals delivered by the accelerometer, however, we concluded that the accelerometer itself was not the problem. We were not able to detect any obvious errors within the code either. The problem was very eventually resolved by connecting the unused inputs to unused signals within the module. Thus making sure that the FPGA was not setting those pins randomly high and crashing the accelerometer.

We were able to test the LED matrix and *LEDDriver* module by outputting 64-byte strings of data from *LEDDriver* to the LED matrix to see the order that these bytes of color data were displayed on the LED matrix.

Once we had the accelerometer and LED matrix functioning properly, we were able to test other modules by "wiring" them together in a module and seeing if they displayed properly on the LED matrix and exhibited the correct behavior.

Because there were a large number of modules and signals in our design, testing and debugging was sometimes difficult because there were many possible places for error. After observing errors or incorrect behavior on the LED matrix, going back to the code and simulating individual modules or a few connected modules in Xilinx often proved to be helpful.

Toward the end of the project we ran into a particular problem with the accelerometer. In hot muggy conditions the nominal 50% duty cycle at level position would skew to more than 75%. This made it impossible to use. Connecting it to an oscilloscope we found that the analog outputs remained unaffected but the PWM digital outputs were misbehaving. However, once the accelerometer was in airconditioned space it would start working properly again. We never worked out the cause although are suspecting damage to the accelerometer during soldering.

Section 3: Justification and Evaluation of Design

Our design largely benefited from its modular nature. Dividing the design into several modules made it easier to divide the responsibilities of the project among the group members and also made the task of simulating smaller components easier.

However, the large number of modules and the implementation of several levels of sub-modules made adding additional signals or features a tedious and painstaking process. This involved changing the ports in the component declarations in several different modules to make one change. To mitigate the tediousness of this task, it might have been better to decrease the number of instantiated modules and incorporate the code directly into the higher-level modules. This would of course decrease the compactness of the code for these higher-level modules, so discretion would have to be used when doing this.

Because adding additional features did become an annoying task, it also would have been better to have thought out what additional features we would likely implement and write code in such a way that would have required fewer changes to the modules in order to implement these additional features.

The most significant improvement that could be made to our design would be to use a microcontroller to take on a significant part of the game related functions from such modules as the *MainController* and *GameLogicFSM*. A microcontroller is better suited to the needs of this game design. Using a microcontroller would have made this logic easier to implement and also would have made our physical design more compact. We could have used the FPGA fabric for the more real-time parallel components such as the displays and accelerometer input decoding.

Section 4: Conclusion

As stated earlier, our goal was to create a visually appealing and challenging handheld game. Our original project proposal might have seemed overambitious, but most of the objectives presented in it were successfully carried out. The idea of having moving enemies and power-ups, multiple levels, a game timer and sound were all implemented. As we worked on the project, we actually came up with new ideas and were able to implement those as well. We implemented "heat-seeking ghosts" and not only did we have multiple levels, but we also have multiple difficulty settings for the levels. The visual design of each level also exceeded our expectations. We did not expect levels to have themes or for the visuals on the LED display to appear as anything besides colored dots. However, we were able to create reasonably cohesive images and forms with the LED display that produced a much richer and more entertaining visual experience.

One feature we were unfortunately unable to implement was controlling the game timer with input from a heart rate monitor (that is, the faster the heart rate input, the faster the game timer ran). We did not have enough time to work on this feature and more importantly we lacked the proper equipment (specifically, a SPI-bus heart rate monitor) to implement it.

For future groups, we recommend thinking the design through thoroughly before trying to implement it. Before writing any code, we made major revisions to our design that made our modules simpler, cleaner and actually feasible. Our design originally entailed passing 64-byte strings of data between modules and comparing the individual bits of data using comparators. This would have been an unnecessary waste of FPGA resources and clock cycles. Having a good design greatly facilitates the actual creation of the project.

Section 5: Acknowledgements

We'd like to thank Professor Hansen and Adam Marano, Austin Zheng and Josiah Gruber for all their time, energy and clear love for the craft. We'd like to thank Adam Marano in particular as our final project TA. He spent many a night in the lab with us, keeping us company no matter what. Thank you guys for a great ENGS31!

Section 6: References

- Useful information for working with SPI bus interface between *LEDDriver* and the LED backpack

  <http://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus>

- Helpful information used to develop the *ACCELDecoder* module and work with the accelerometer

  <http://www.analog.com/en/sensors/inertial-sensors/adxl213/products/product.html>