**ECE532 Final Report: Laser-Hunting Game**

By:

Deng Pan, Tianyi Yu, Mingwei Ding, Ming Hsu

Apr 13 2017

Table of Contents

[1 OVERVIEW 3](#_Toc479861684)

[1.1 Goals 3](#_Toc479861685)

[1.2 Block Diagram 3](#_Toc479861686)

[1.3 Brief Description of IPs 4](#_Toc479861687)

[1.3.1 Laser Detection IP 4](#_Toc479861688)

[1.3.2 VGA Interface 4](#_Toc479861689)

[1.3.3 Game Software 4](#_Toc479861690)

[1.3.4 Keyboard Interrupts 4](#_Toc479861691)

[1.3.4 Audio Output 4](#_Toc479861692)

[2 OUTCOME 5](#_Toc479861693)

[2.1 Results: 5](#_Toc479861694)

[2.2 Possible Future Improvements 5](#_Toc479861695)

[3 PROJECT SCHEDULE 7](#_Toc479861696)

[3.1 Original Milestones: 7](#_Toc479861697)

[3.2 Actual Milestones: 7](#_Toc479861698)

[3.3 Comparison: 8](#_Toc479861699)

[4. DESCRIPTION OF SYSTEM COMPONENTS 9](#_Toc479861700)

[4.1 Laser Detection Blocks 9](#_Toc479861701)

[4.2 Audio Output Block 9](#_Toc479861702)

[4.3 Processor 10](#_Toc479861703)

[4.4 Keyboard Interrupts 11](#_Toc479861704)

[4.5 Drawing Algorithm Blocks 11](#_Toc479861705)

[5. DESIGN TREE DESCRIPTION 14](#_Toc479861706)

[5.1 Repository Structure 14](#_Toc479861707)

[6 TIPS AND TRICKS 15](#_Toc479861708)

[APPENDIX A – COMPLETE BLOCK DESIGN 17](#_Toc479861709)

# 1 OVERVIEW

## 1.1 Goals

For our project, we envisioned a prototype shooter game implemented on an FPGA. This would model a survival zombie game, target practice for military training, etc. The game would support multiple game modes, difficulties, audio input/output to create a fully immersive shooting experience.

Our original goals were to have a voice command system to allow the user to have voice control for commands in the game, support multiple modes with scoring including multiplayer and single player, and to most importantly create an algorithm to detect the laser pointer to simulate shooting.

## 1.2 Block Diagram

Final Block Design.png

PLB

Figure 1: Block diagram of overall design

## 1.3 Brief Description of IPs

### 1.3.1 Laser Detection IP

Contains all the modules for collecting input VGA data and processing frame-by-frame. First a threshold operation is performed to filter out irrelevant colors and the resultant detection blobs are averaged to produce an x-y coordinate. This output is a running average over multiple frames to produce a smooth output.

### 1.3.2 VGA Interface

This block formats the video data captured and overlays the score, timer and target information read from the PLB. Drawing in each block is done through a separate VGA controller writing pixel-by-pixel to a compositor which combines the data. The resultant image is output on a VGA monitor.

### 1.3.3 Game Software

The main game program is written in software. It generates targets and movement patterns depending on difficulty settings stored in local variables. Detection is done by comparison of the target coordinates with the detected laser coordinates received through the laser detection IP. Score and timer data is written onto the PLB which is then displayed through the VGA controller. The c program is also responsible for interfacing with the interrupt controller.

### 1.3.4 Keyboard Interrupts

The AXI interrupt controller flags incoming UART characters as interrupts. This custom software interrupt handler parses the inputs and adjust game settings accordingly. This is an efficient alternative to continuous polling for user input.

### 1.3.4 Audio Output

This block plays background music for the game based on control signals from the PLB. It also plays a buzz sound when the player successfully hits the target. This IP block is integrated with the Microblaze processor, allowing software commands to mute or change audio tracks.

# 2 OUTCOME

## 2.1 Results:

For our final outcome, we were successful at meeting some of the goals, requirements and criteria set during the proposal of our project. In the following tables below show which specifications were met, partially met or not completed. Also, there several new requirements that were added or changed during the design process as we adapted our project.

|  |  |
| --- | --- |
| **Feature** | **Status** |
| HDMI input/output | Changed: VGA used instead due to lack of hardware |
| Laser Detection | Completed: Able to detect up to 2 lasers |
| Voice Commands | Changed: Takes keyboard commands instead |
| Multiple Modes | Added: Single player and multiplayer available with two lasers |
| Display Scoring | Completed: Can display score for one single laser for single player and two scores for each of the lasers for multiplayer |
| Audio output | Completed: Have 2 background songs and buzzer for hitting target |
| Timer and Difficulty settings | Added: allow runtime modification of difficulty, duration of each game and various polish improvements |

## 2.2 Possible Future Improvements

If there was extra time to work on our project we would have refined our detection algorithm to ensure that we can accurately detect multiple laser pointers regardless of lighting conditions. Also, more modes could have been added such as survival mode with life counters to add more variety to our game.

Ideally, the VGA input would be replaced by HDMI to allow higher resolution for the output. The main reason why VGA was used for the project was because we didn’t have access to a Nexys Video board or HDMI camera. Also, we would have liked to have stored our custom graphics differently. Instead of having hardware modules draw out a hard-coded image, we could have loaded those any custom graphics into memory first and read them. This would have been easier to create the loading screens and change between them.

If another group were to continue our project, the recommended steps would be to work on fine tuning the detection algorithm by detecting the average lighting of the room and comparing it with areas of high intensity of lighting. In addition, thresholds should be modifiable in software for runtime adjustments.

# 3 PROJECT SCHEDULE

## 3.1 Original Milestones:

|  |  |
| --- | --- |
| Milestone # | What to accomplish |
| 1 | Create a C program for game simulation logic and have i/o using keyboard commands |
| 2 | Testbench demo: Get the testbench working for game simulation logic |
| 3 | Block for laser point detection: convert pointer location to array indices |
| 4 | Create audio i/o block which captures speech and converts it into commands |
| 5 | Mid-Project Demo: Integrate video capture and audio i/o with the game simulation logic |
| 6 | Create HDMI interface block and have it display hard-coded 2D graphics on a monitor |
| 7 | Integrate the HDMI block with rest of system along with final testing and debugging of the game. |

## 3.2 Actual Milestones:

|  |  |
| --- | --- |
| Milestone # | What was accomplished |
| 1 | C program featuring target generation, frame updating and collision checking to demonstrate game logic |
| 2 | Laser detection algorithm and image processing in software, audio input dropped in favor of keyboard commands |
| 3 | Configured OV7670 capture modules, outputs xy coordinates, basic audio IP to play single tune on repeat |
| 4 | Game logic implemented in hardware, tuned detection module for accuracy, added scoring and display on LEDs |
| 5 | Game logic completed and integrated with audio in hardware, added different tunes and features in audio IP |
| 6 | Migrated game logic to software, added interrupts and custom VGA controllers and compositors for overlapping score, timer and video feed. Score and timer are displayed on monitor instead of LEDs |
| 7 | Added multiplayer support for green laser, revised detection algorithm to sort between green and red laser and enable multiplayer in software |

## 3.3 Comparison:

A major difference is that HDMI was replaced with VGA input/output as the lab ran out of HDMI cameras. This led to image processing being done in hardware instead as using VDMA with VGA input costed too much time to configure correctly.

Another revision was that the audio input processing was replaced by UART communication with interrupts as performing FFT and frequency analysis to parse audio information was too complicated for this project.

Aside from those revisions, our project accomplished what we intended in the allotted timeline. Progress was slow as we figured out the basics of Vivado and block designs in the first few weeks but was made up for in weeks 5 and 6 with the migration of game logic to software. Multiplayer support was finished and integrated in the final week in time for the demo.

# 4. DESCRIPTION OF SYSTEM COMPONENTS

## 4.1 Laser Detection Blocks

The Laser Detection IP block is a combination of several sub-IP blocks. The OV7670 camera input block first detects the image and send the pixels to the VGA input block. Then the VGA input block will apply threshold detection algorithm to detect the green and red lasers then convert them into some pixels with fixed RGB values. After the threshold detection, the processed pixels are sent into the Bram memory frame buffer for storage. Once the pixels are stored in the memory buffer, the Laser detection processing buffer will extract the pixel data from the memory and calculates an averaged x and y coordinates for both red and green laser. As for the output of this whole laser detection IP block, the averaged x and y coordinates of the lasers are passed to the bus.

The final calibrated threshold values are shown below.

// [15:12]: red intensity

// [10:7]: green intensity

// [4:1]: blue intensity

// Red Detection - Strong White

if (d\_latch[15:12] > 4'b1110 && d\_latch[10:7] > 4'b1110 && d\_latch[4:1] > 4'b1110)

// Green Detection - Weak White

else if ( d\_latch[15:12] > 4'b1000 && d\_latch[10:7] > 4'b1000 && d\_latch[4:1] > 4'b1000 )

## 4.2 Audio Output Block

The audio output block has two main functionalities: Providing different background music and providing a buzzer sound when the player successfully hits a target.

To allow the Nexys 4 board to output sound, we need to send pulse width modulation signals to the board. In our implementation, we have two clocks. The first clock is a 6MHz clock, that clock is used to generate different tones of sound. For different tones, each has a different starting value, which is incremented by 1 every clock cycle. Upon until the value reached a fixed value (which is 16383 in our design), we flip the “speaker” signal. With this method, we can generate different pulses with different width, which will be the signal of generating different tones. A narrower pulse has a higher tone and a wider pulse has a lower tone.

After we’ve constructed different tone, we can make our own music with those tones. We can select different tones and put them in different orders. Every 4Hz clock cycle, the tone will change to the next tone, thus providing different music.

The mute signal and the switch music signals are controlled by the Microblaze Processor. The player can send signals through the SDK console to mute or switch music.

The Buzzer sound is also controlled by the Microblaze processor. Whenever a player hits a target, a signal is sent to this audio output block from the Microblaze processor and the background music will switch to the buzzer sound for a very short duration, then switch back to where the music has stopped.

## 4.3 Processor

The majority of the game program is implemented in software. The microblaze processor receives input via the AXI bus from our peripheral modules and writes outputs to the VGA controllers. The game program behaves as follows:

1. Initialize interrupt registers and enable interrupts
2. Initialize random seed, remaining time and difficulty variables, and audio controller, scores and screen selection registers
3. Generate a target with randomized x and y coordinates
4. Wait for keyboard interrupt to select game mode (SEE NEXT SECTION 4.1)
5. Loop forever:
   1. Move the target a set direction and distance and write the new coordinates to the VGA controllers to display. The distance and direction change periodically based on the difficulty, the target is wrapped around the screen if it ends up outside the display area
   2. Get red and green laser detection coordinates from camera IP
   3. Check if detected coordinates are within a threshold to the target, if so, increment score of the player that hit the target and write the new value to the score\_display module
   4. Decrease the time remaining and write the new value to the timer\_display module
   5. Anytime, the user can access menu functionality by triggering the keyboard interrupts, the details are described in the next section

## 4.4 Keyboard Interrupts

A menu system is implemented using UART keyboard interrupts. The microblaze is configured with a custom interrupt service handler (ISR) which collects the characters and parses them, allowing the user to select a variety of options such as muting the background music, pausing the game, restarting, selecting difficulty and choosing between 1P and 2P modes.

The input is sent one byte at a time through the UART interface to the AXI interrupt controller, which flags the interrupt source address and triggers a software interrupt. The interrupt is held high while the UART FIFO buffer is non-empty, allowing more complex recognition patterns involving multiple characters. The program execution is suspended and jumped to the ISR to parse the data. We receive the bytes one at a time, which will collect the input as well as clear the buffer. Depending on the input, the ISR updates relevant global variables and writes to hardware registers through the AXI interface before returning to the jump point in the main program.

## 4.5 Drawing Algorithm Blocks

The drawing algorithm is used for determining the colour of each pixel and writing this data to BRAM. This stored data is then read by Microblaze processor and sent to the monitor as a VGA signal.

The main drawing algorithm is split into multiple IP modules in our program. In appendix A, it shows the entire layout of entire drawing algorithm block and how they are connected. Depending on the signal sent from the interrupts, different IPs are active at different times. When active the IPs then determine the colour of the pixels that are assigned to change and the following below are the list of functions of each grouping of IPs:

1. This grouping is responsible for displaying the score obtained when hitting the target. The score is determined by the game processing logic and is sent to score\_display 1 and 2. Depending on the mode, for single player only score\_display 1 is active and for multiplayer both score\_display IPs are active. Notice for score\_display 2 there is an extra mode signal to determine if it is in multiplayer. The score is displayed as a 2-digit number on the top right of the screen and is written pixel-by-pixel depending on each digit. Also for multiplayer, the score is red for the red laser pointer and green for the green laser pointer. The data for the colour of each pixel is then combined together in the compositor before moving to the next stage

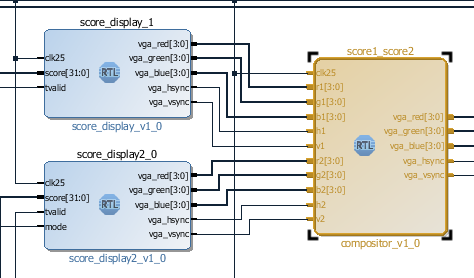


Figure 2: Score display blocks

1. The IP block for creating the different screens is in figure 2 below. This includes an IP for drawing the layout of the menu, pause and game over screen.

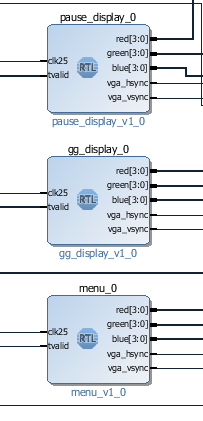


Figure 3: Display modules for different screens

1. The timer display IP is used to display the time left for the game. This IP takes in the amount of time left for the game determined by the game logic processing block and displays it as 2 digits on the top left of the screen. Depending on the amount of time left, the digits change colour from green (99s to 50s), to yellow (49s to 10s) and then red (10s to 0s).

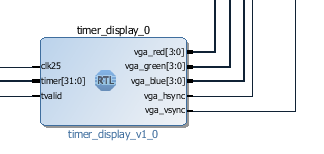


Figure 4: Timer display module

1. The last block, is used to select which things are to displayed at what stage of the game. For camera\_score, score\_gg and score\_camera\_timer IPs, they all combine display data from the previous blocks or PMOD camera data and send it to the screen select IP. The camera\_score combines all data from the PMOD camera and score data, the score\_gg combines data from score display and game over screen, and the score\_camera\_timer combines the timer display with camera and score data. The compositors overlay the inputs to make sure scores are visible during gameplay. Lastly, the screen\_select IP is controlled by the interrupt commands and determine whether the menu, game, pause or game over screen is displayed.

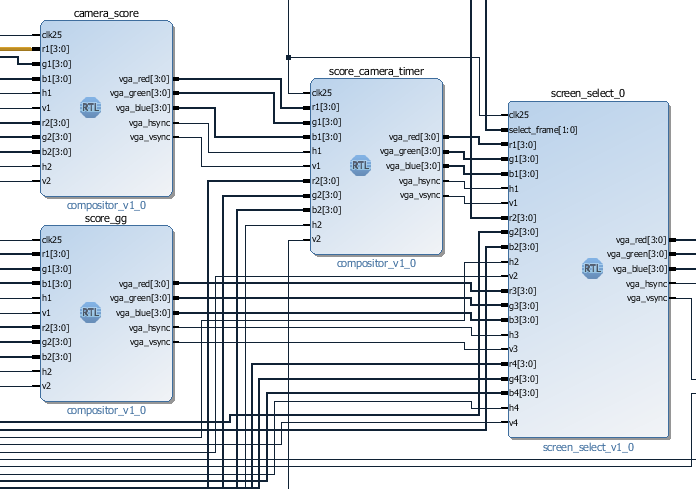


Figure 5: Screen election module

# 5. DESIGN TREE DESCRIPTION

Our project in its entirety is available at <https://github.com/fishinsea/ECE532_G18_LaserHuntingGame/>

## 5.1 Repository Structure

- *src*: the project files

- *docs*: report, presentation and videos

- *G18\_IPSourceFiles*: source code for IPs

- *src\laser\_hunting\_clean.sdk\interrupts\src* contains c code for the game software

- *src\laser\_hunting\_clean.srcs\sources\_1\imports* contains the audio ip

- *src\laser\_hunting\_clean.srcs\sources\_1\new* contains custom vga controllers for displaying to the screen

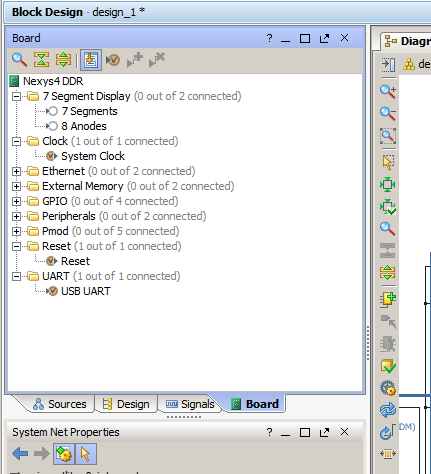
- *src\laser\_hunting\_clean.srcs\constrs\_1* contains the constraints files complementing the default constraints from the Nexys4 Board files

- *G18\_IPSourceFiles\Dual\_Laser\src\ov7670\_capture* contains the detection thresholds for color detection

- *G18\_IPSourceFiles\Dual\_Laser\src\marker\_detection* contains the algorithm for computing the coordinates for the detected lasers

# 6 TIPS AND TRICKS

Running synthesis takes a long time, here are some tips to reduce the number of times you have to do it.

* X\_intc.h, a required include for initializing the interrupt controller requires at least 16kB of local memory. Make sure to have enough for that and your software if you’re planning to use interrupts (64kB is probably safe).
* Double check pin constraints, and use the board files if possible, the SF design center computers have board files for the Nexys4 DDR and the DDR tutorial has instructions on how to add them to your project. They can help you find out if your pins are configured correctly as well as not require you to manually specify constraints for common ports such as GPIO. Making a GPIO module’s port external automatically connects it to its corresponding port on the board. Simply click the “Board” tab to see which ports are connected. (Note this does not manage PMods or other external circuitry unless the IPs you’re using specifically has the proper interface axis, you still have to add JA, JB and VGA pin constraints manually if using the handout code for the OV7670 camera). A sample image is shown below
  + 
* Behavioural simulation can be done without running synthesis. If your inputs are relatively easy to model using hard coded verilog, write a quick testbench and confirm results in simulation.
* Make sure to initialize registers when required, X signals in simulation most likely means you forgot to initialize a register you’re using
* Move as much as possible to software, SDK debugging tools are much better than synthesizing ILA cores into your design, and running a software program takes 15 seconds compared to 15 minutes of changing hardware

# APPENDIX A – COMPLETE BLOCK DESIGN

