ECE 350 Final Project Report - Epic Web Hero

Carrie Hunner (clh87) and Harry Ross (hgr8)
GitHub - https://github.com/hglr24/EpicWebHero
Duke University Pratt School of Engineering

Introduction

Our goal with this project was to create a shooting gallery style game as might be seen at a carnival. Based off of Spider-Man, the player has a pair of gloves that can be worn. Each glove is equipped with a laser diode attached to the back of the glove and a button attached to the palm area. When the player presses the button, a signal is sent from the glove to the FPGA. The FPGA has a module that monitors the signal and enables the laser diode and then enforces a cooldown period to ensure that the player cannot simply hold the button down and sweep the laser. The player is shooting at a cityscape laden with targets, both good and evil. Each target has an LED used to signal when the target is active and available to be shot, as well as a photoresistor to sense when the target is hit by the laser. These are connected to the FPGA which determines which two targets are active as well as receives signals to handle when a target is shot. The player's score is displayed on the FPGA's 7-segment displays and increases by 150 with every villain hit and decreases by 100 with every civilian casualty. Additionally, as the player's score increases, the amount of time that a target is active decreases, thus increasing the game's difficulty. The player has 2 minutes to get as high a score as they can.

Registers

To allow for hardware and software to interact, certain registers were agreed upon to only be written to by one or the other. The following table includes all of the registers necessary for hardware and software interaction.

Register Number	Register Name	Expected Values	Purpose	Writer
1	$bp	Anything other than 0 = pressed	Start/Reset Button	IO asynchronous
2	$t0hit	0 = not hit, 1 = hit	Target 1 hit	IO asynchronous
3	$t1hit	0 = not hit, 1 = hit	Target 2 hit	IO asynchronous
4	$t0active	Number [0, 9]	Active Target 1 assigned by MIPS	Instruction Written
5	$t1active	Number [0, 9]	Active Target 2 assigned by MIPS	Instruction Written

6	$timer1	0 = timer done, 1 = timer active	timera completion checks	IO asynchronous
7	$timer2	0 = timer done, 1 = timer active	timerb completion checks	IO asynchronous
8	$gametimer	0 = timer done, 1 = timer active	timerc completion checks	IO asynchronous
9	$score	Between 0 and 9999	Keeping Score	Instruction Written

FPGA Input and Output

Inputs

Palm Buttons

A pushbutton is attached to each of the player's gloves. It is a part of a MOSFET circuit (see the Circuits section) and the output of that circuit is fed into the FPGA. When the button is unpressed, the FPGA reads a HIGH signal. When pressed, the FPGA reads a LOW signal. These two inputs (one for each hand) give the FPGA information as to when the player attempts to shoot.

Photoresistors

A photoresistor is attached to each of the ten targets. Each of these is a part of a MOSFET circuit (see the Circuits section) and the outputs of those circuits are fed into the FPGA. When the photoresistor is in resting position(i.e. ambient light) the FPGA receives a HIGH signal. When the laser hits the photoresistor, the FPGA receives a LOW signal. Using these inputs, the FPGA can track which targets were hit and cross-reference that information with which targets were active to induce some resulting effect.

Start/Reset Button

One of the pushbuttons on the FPGA is designated as the start/reset button. This input is bound to one of the regfile registers, and MIPS instructions respond accordingly.

Outputs

Target Selection Bits

Two targets can be active at any given moment, determined by FPGA outputs bound to regfile registers. To save on GPIO pins, we implemented two 4 to 10-bit hardware decoders. These each receive four bits from the FPGA indicating which target number should be activated. An array of OR gates between the two decoder outputs ensures that two signals can be sent to the same target at once without malfunction.

Laser Enable Signal

The FPGA has GPIO pins designated for the one lasers enable signal on each glove. When one of the laser's enable outputs goes HIGH, the corresponding laser turns on. In having control over the palm button inputs as well as the ability to trigger the lasers, the FPGA can enforce a cool-down period such that the user cannot simply hold his/her fingers and constantly be emitting a laser.

Score

Four of the 7-segment displays on the FPGA are designated for displaying the player's score. A Verilog module was written that takes in the binary score of the player and translates that to the 7 segment displays.

Changes Made to Processor

randn Instruction

A Verilog module was created that is capable of generating a pseudorandom number between 0 and 9. The following equations were used for the random number generation.

$$ranNum_{i+1} = (3 * ranNum_i + rand_{count}) \ \% \ score$$

$$0 \leq ranNum \leq 9 = ranNum \ \% \ 10$$

The above equation receives a counter input that increments with each clock cycle as well as the player's score and the previous number generated, with the initial seed being zero. The value three is arbitrary and could be changed. Every clock cycle, a random number is generated and output by this module. After the random number is generated, the value is modulo with 10 to ensure that we get a value in the range [0:9].

This algorithms suits our needs because it is dependent on a player's score at an instant in time as well as a counter that increments every clock cycle. The odds of a player hitting enemies in the exact same sequence at the exact same time stamp are improbable, thus enabling us to have a different sequence nearly every time.

The randn instruction was created as a JII type instruction. This is because the only input from the instruction is the destination register where the random value should be stored.

Since the random number generator Verilog module relies on the current game score (stored in skeleton), it made the most sense to have the module reside in the skeleton and output its random value into an additional port of the processor. Since the random value changes every clock cycle, the processor need only select the value instead of ALU output, MultDiv output, etc. in the execute stage of a randn instruction to ensure it gets stored in the designated register.

timer Instructions

The timer instructions - timera, timerb, and timerc - serve to begin one of three respective timers with a specified length (in seconds) read from a register. As with randn, this instruction is JII-type, but instead of using $rd as a write port, it is used as a read port. In the execute stage of the pipeline, the processor will send a start signal to the timer specified by the instruction choice along with the timer length. This requires that additional I/O ports exist in the processor for these signals to be communicated. Once a timer receives a start signal, it latches the length it sees on that clock cycle and begins counting down, outputting a high signal on its 1-bit output port until it is complete. This design ensures that the timer will not be affected if the timer length signal from the processor changes mid-count without a start signal. Each timer outputs its 1-bit utilization signifier (high while counting, low when done) directly to a dedicated register in the regfile. This means that subsequent MIPS instructions can ping these registers intermittently to determine if a timer is complete.

halt Instruction

One custom instruction that proved necessary was the halt instruction, which halted processor progress upon execution. If halt is processed, the program hangs in a constant state that allows for easy debugging. Halting was also necessary to prevent non-looping programs from restarting upon completion after being blasted. This command was not used in the final game script implementation.

Register File

The register file saw a number of modifications to accommodate our new design. Many components of our device require direct read/write access to the regfile, so additional I/O ports were added to account for them. For example, our "real-life" inputs (flex sensors, photoresistors, etc.) require direct regfile access to allow later MIPS instructions to know if they have been triggered. This paradigm is an abstraction of the DMA design used on modern processors to allow direct communication between system devices and memory without intervention from the CPU for higher efficiency. Any of the registers that have been designated to be written exclusively by I/O devices are strictly forbidden from being modified through normal processor writes (in the same way the zero register is forbidden from being overwritten in the original regfile design). This decision limits the number of registers allowed to be general-purpose in the operation of the MIPS game script, but not enough to hinder our implementation or necessitate frequent memory operations to store and retrieve variables.

Challenges Faced

Laser Diode

We initially purchased a pair of commercial laser pointers off of Amazon with the intent of removing the circuits from their housing and using them directly on our gloves. We failed to realize that their housing was made of metal, however with quite a bit of effort, we were able to access the power contacts. When we applied 1.5V from the voltage supply, the laser worked as expected. We then attempted to integrate it into a circuit where its power supply was driven by an FPGA GPIO pin. We measured the internal resistance in the laser pointer and built a voltage divider to decrease the voltage supplied from the 5V of the FPGA down to 1.5V. When we implemented this, however, the voltage across the laser pointer did not exceed ~.6V, thus failing to turn it on. Even after applying the 5V directly from GPIO (with a small pull-down resistor), the laser still would not turn on. After sinking a large amount of time into debugging and testing, we returned to Amazon and purchased a pack of ten laser diodes intended for breadboarding. They worked immediately with our FPGA configuration.

Protoboard Debugging

After creating individual circuits on breadboards to gain proof of concept, we soldered the decoders and their corresponding OR and INVERTER gates, the photoresistor MOSFET circuits, and the flex sensor MOSFET circuits. We wrote a simple Verilog script designed to activate the targets one at a time, waiting for the corresponding photoresistor to be triggered (ie the target to be "shot") before iterating to the next active target. In performing this test, we observed that there were several targets that were being signaled as active when it should have been just one. After some visual and multimeter observations, we determined that one wire was soldered to the wrong row on the protoboard, an entire side of one of our hex inverters was not functioning as expected, and one of the decoders had a short between two of the outputs. We moved the incorrect wire and that removed one of the unexpected active targets. We then moved the inputs and outputs to the two bad inverters to another pair of inverters. This fixed another two of the unexpected actives. Lastly, in inspecting the decoder, we believed the short to be internal to the chip itself, as there was no clear connection between the pins on the protoboard. Some of the ribbon cable wires were fragile and broke near the solder point during troubleshooting, which added additional repair time.

Flex Sensors

For the majority of our design and implementation process, we were relying on flex sensors in each hand to run down a finger and detect when a hand was performing the trigger motion. These were in place of the eventual palm-mounted pushbuttons used in the final design. For all of our initial testing and initial protoboard creation, these sensors seemed reliable enough to deliver the results we were looking for. Unfortunately, near the very end of implementation and testing with actual game code, we realized that the sensors were not precise enough for our design, so to make the game more enjoyable to play, we replaced their functionality with that of the aforementioned pushbuttons. This meant that our final glove design was less appealing visually but more tactilely satisfying for the player.

Circuit Diagrams

Finger Trigger Circuit

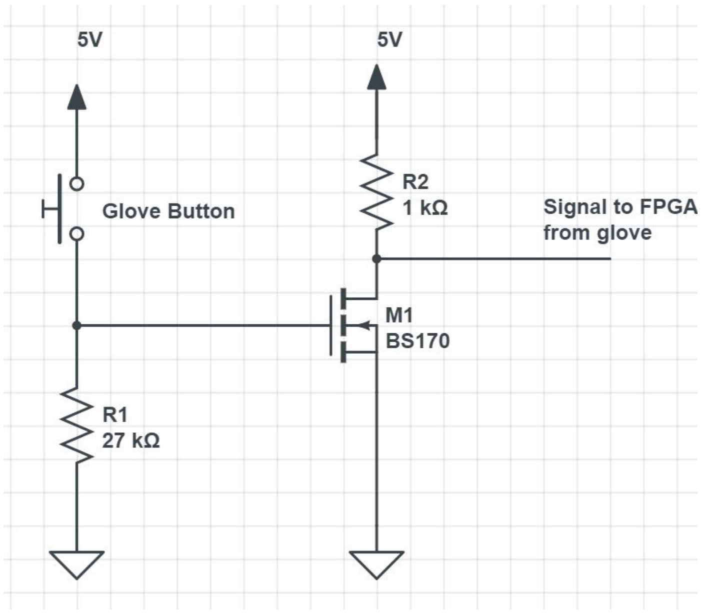


Figure 1: Glove Button Trigger Circuit

The above circuit is used for each glove's input method. Its purpose is to output a HIGH or LOW signal based on the palm-mounted pushbutton input. This circuit is functional but much more complex than it needs to be because originally, a flex sensor was in place of the pushbutton, requiring much more thought and calibration to go into the circuit implementation. Conveniently, the circuit still functioned correctly after the replacement occurred, meaning that it would have been ultimately unnecessary for us to simplify this circuit that was already soldered into a protoboard. R2 acted as a balancing resistor when this circuit was used for a flex sensor. Adjusting its value allowed us to set the point at which the flex sensor would be bent enough to warrant a shot being fired. We settled on 27k ohms after testing several different values.

Photoresistor Receiver Circuit

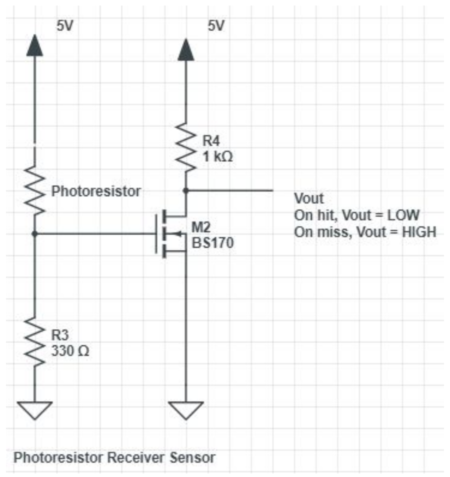


Figure 2: Photoresistor Receiver Circuit

The circuit above is nearly identical to that of the Finger Trigger Circuit. Its purpose is to output a hard HIGH or LOW value depending on the photoresistor. We designed the circuit such that when hit by the laser, the photoresistor value changes drastically enough that the voltage applied to the MOSFET exceeds the threshold voltage. Again, R3 acts as a calibrating resistor in that we tested various values until the circuit's sensitivity to the photoresistor values with ambient light versus the laser met our needs.

Laser Enable MOSFET Circuit

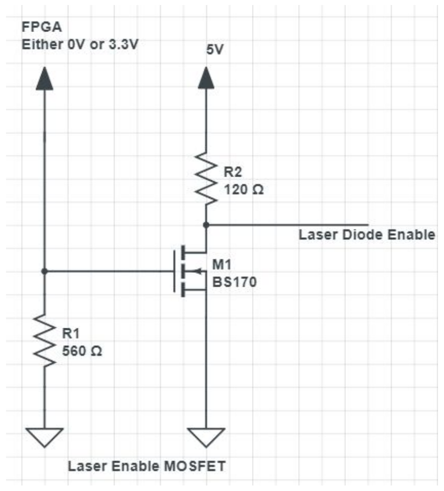


Figure 3: Laser Enable MOSFET Circuit

The above circuit was implemented to give the FPGA control over the laser's enable pin. Originally, the laser was connected directly to the FPGA output, but the voltage was too low and the resulting light was too dim. Now, the FPGA pin can output a HIGH, thus grounding the laser diode and turning it off, or output a LOW and allowing the diode to receive power. R1 was selected because this value allowed for the FPGA's output to easily exceed the MOSFET's threshold voltage when outputting a HIGH. R2 was selected because when the diode is enabled, the light is bright enough for the photoresistors but when off, a very dim light is still seen which can be used by a player to aim.

Testing

Circuits

We built several smaller circuits to gain proof of concept for each of our designs. After verifying that they worked on the breadboard, we then transferred them to a breadboard-style proto board. Before connecting it to power, we performed visual inspections for shorts and solid connections as well as performed continuity tests with the multimeter. After confirming that nothing was shorted, we connected it to power and then connected it to the FPGA. When it wasn't working quite as expected, we used a multimeter to check pins for problems. See "Challenges Faced" and then "Protoboard Debugging."

Verilog

We created many small Verilog files to test specific basic communications between the hardware and code of our project. Later, we would test more robustly with an assortment of MIPS files in our final design.

Assembly Code

Wait and Initialize

The MIPS script first waits for the register that stores the button output to go HIGH. Once it does, all registers that are used for game control are reset (ie score, score threshold, etc). Next, the game timer is started and two targets are started (see "Target Started" for more details).

Game Loop

Game Timer and Button Check

The program checks if the game timer is up. If it is, the program loops back to the "Wait and Initialize" section and does not do anything until the button is pressed to start the game. If the reset button is pressed, the program jumps back to the "Initialize" section and starts the game over.

Target Hit Check

The program next checks if target 1 is hit by inspecting the value in the $t1hit register. If it is hit, it is checked whether a civilian or villain is hit and the score is either decreased or increased respectively. Additionally if it is hit, a new target is started (see "Target Started" for details). This same process is then repeated for checking if target 2 is hit.

Timer Check

Next, the timer registers for each of the active targets are inspected. If the timer is up, a new target is started (see "Target Started" for details). After checking both, the program loops back to the start of the "Game Loop."

Target Started

A pseudorandom number is generated in the range [0,9] (see “Changes Made to Processor” and then “randn” for more details). This value is then compared to the active target number as well as the target number that is being replaced. If it matches either of these, a new random number is generated. This is to ensure that the active target isn’t selected twice and that the new target is different from the one that is being shut off. If the random number is not either of the current target numbers, a check is performed to ensure that two civilians are never active at the same time. This is done by checking first if the new random number is a civilian. If it is, the already active target is checked if it is a civilian. If it is, a new random number is generated and the process is repeated until a valid random number is generated.

Once a valid random number is generated, a timer is set for the newly active target. The amount of time that the target is active for is dependent on the player’s score. The max time a target can be active for is 5 seconds. Every 1000 points the player earns, the time a new target is active for is decremented by 1. This ensures that the game increases in difficulty as the player progresses.

Improvements and New Features

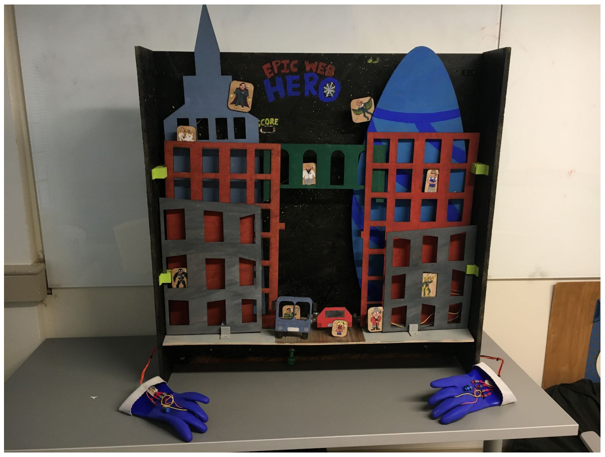
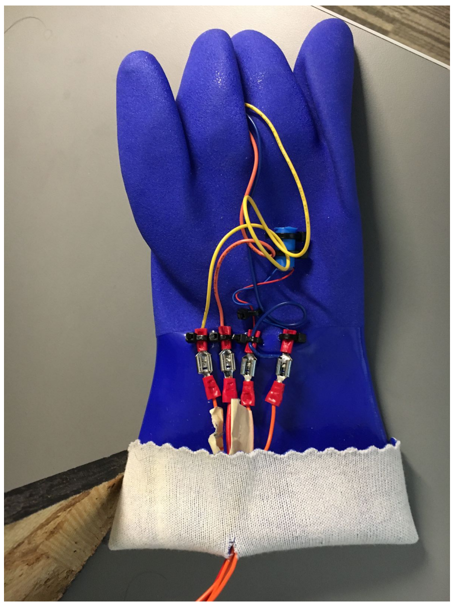
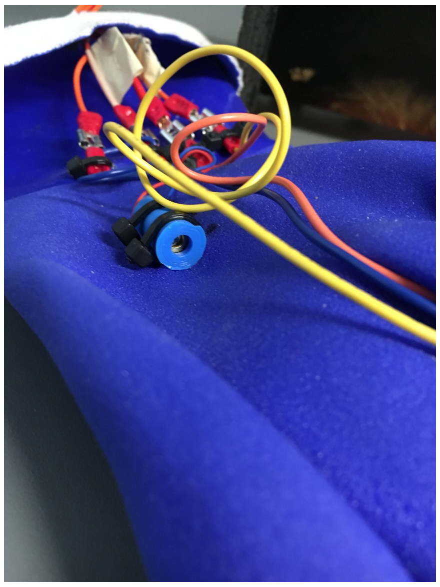
Sounds Effects

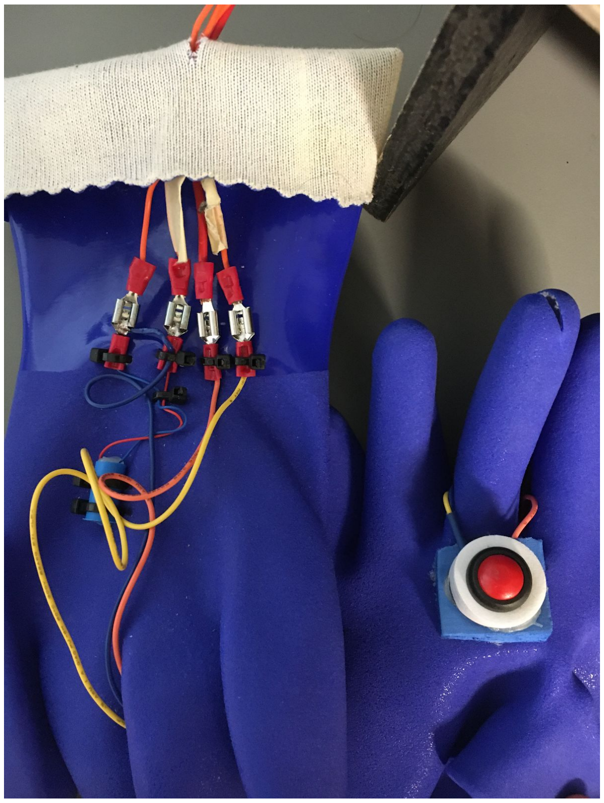
Part of a fun game experience is the sound effects that accompany various events. Currently, the only way to know if a target is hit is if the light turns off, which could also just mean the target was switched, and a change in score, which can be hard to notice while playing. Adding sound effects, possibly when the laser is fired and/or when villains or civilians are hit, could make the game more engaging and intuitive. We are not currently using dmem, so there is plenty of memory to store short sound effects.

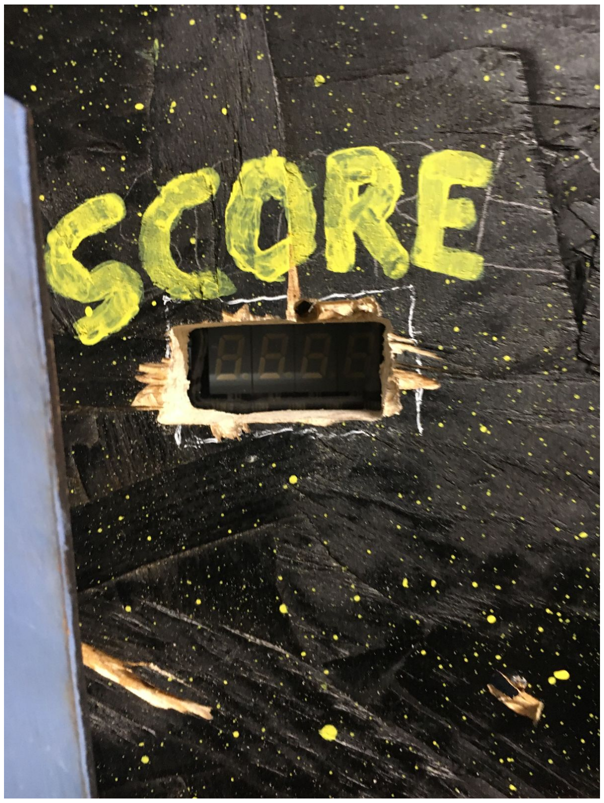
Accuracy/Streak Bonuses

The FPGA receives signals from the flex sensors every time a shot is fired. Given more time, we could use that information along with signals for when a target is hit to keep track of a player’s accuracy. With this, we could add bonuses for example if a player hits three enemies in a row. It would also be possible to generate statistics at the end of the game, as is sometimes seen in video games. These statistics could then be displayed to the user on a monitor.

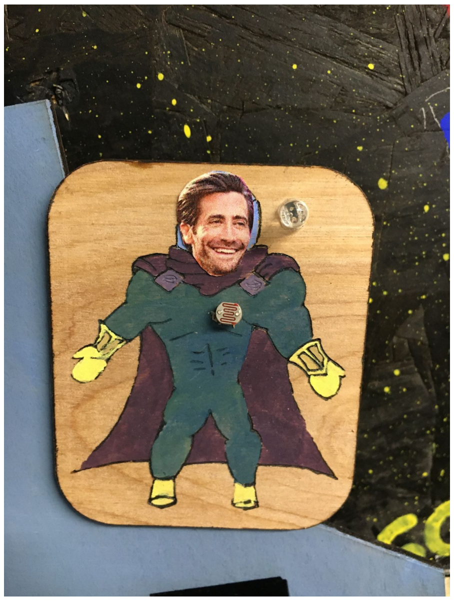
Pictures

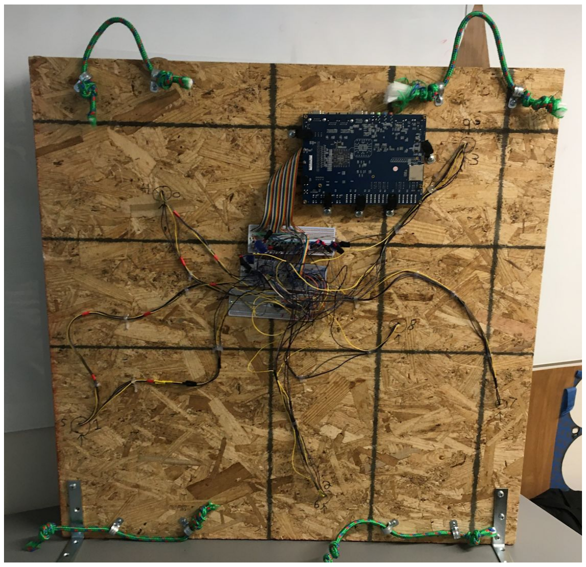
 Full cityscape set with the gloves out front	Cityscape partway through being built
 Back hand of the glove with the wire connections and the laser diode	 Laser Diode on the back of the glove with its 3D printed casing

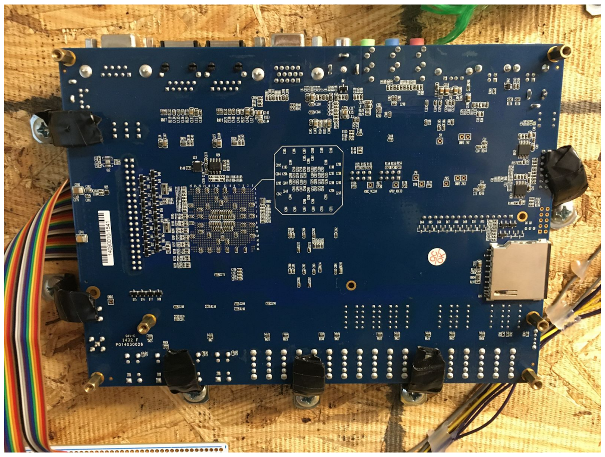
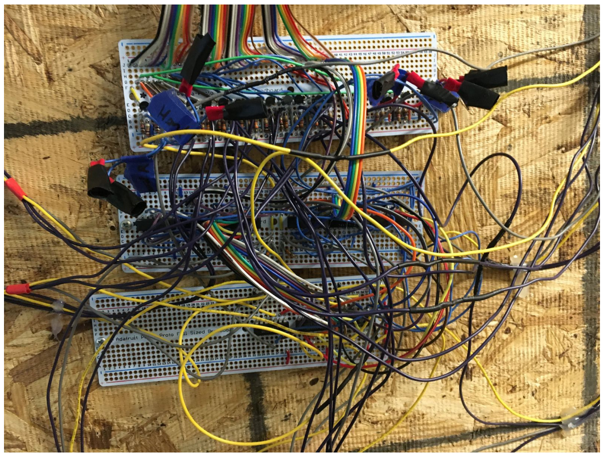
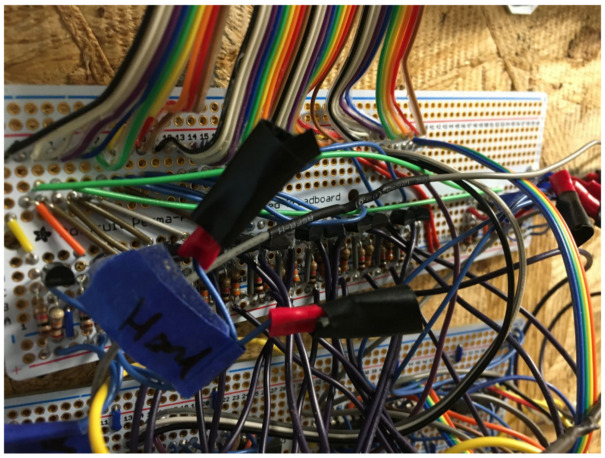
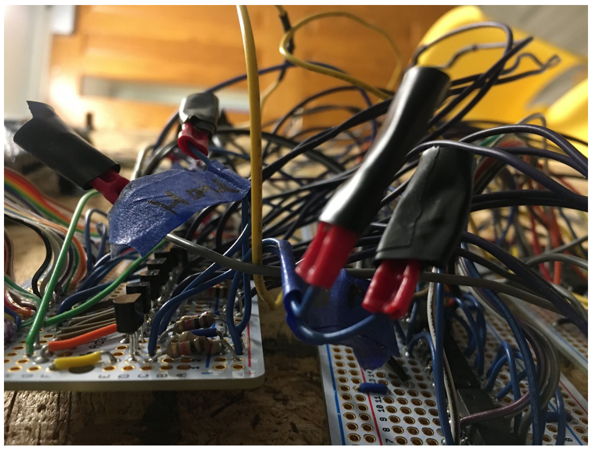

Both gloves, the back of the one on the left and the palm with the one on the right. Notice the button trigger on the right



The score and the 7-segment LEDs showing through a hole cut into the display


A target with a photoresistor over the heart and an LED in the upper right corner


The back of the display with the protoboards, the wires feeding to and from all the LEDs and photoresistors, and the FPGA

 The FPGA mounted to the back of the set. The black latches holding it in place are able to rotate to allow for easy removal	 The protoboards mounted to the back of the display (from top to bottom, FPGA interface and MOSFET circuit board, decoder board, OR gate board)
 The protoboards mounted to the back of the display	 A different angle of the protoboards mounted to the display

Materials and Resources Used

Materials

- Altera Cyclone IV E FPGA
- USB A-to-B cable
- FPGA power cable
- Quartus Prime software and computer
- Full-length protoboards (3)
- Lead-free solder (a lot)
- Hookup wire (a lot)
- 40-pin ribbon cable
- Heatshrink
- Crimp connectors (32)
- BS170 MOSFETs (14)
- Photoresistors (10)
- Quad-OR Gate ICs (3)
- Hex Inverters ICs (2)
- 4 to 10 Decoder ICs (2)
- Resistors of varying values (a lot)
- RGB LEDs (10)
- Laser diodes (2)
- 3D printed laser diode casing (2)
- 1” Plywood (4’ x 3’)
- ¼” Plywood (a lot)
- Self-tapping screws (a lot)
- 3D printed custom brackets (6)
- Angle brackets (6)
- 1-Hole Conduit Straps (13)
- Acrylic paint
- Spray paint
- Hot glue
- Superglue
- Electrical tape
- Masking tape
- Painter’s tape
- Zip ties
- Rope
- Foam shapes
- Gardening gloves (2)

Tools

- Soldering irons
- Desolder pump
- Solder tip tinner
- Heat gun
- Hot glue gun
- Crimpers
- Wire cutters
- Wire strippers
- Pliers
- Helping hands
- Screwdrivers
- X-Acto knife
- Hacksaw
- Filers
- Sandpaper
- Laser cutter
- 3D printer
- Multimeter (testing)
- Power supply (testing)
- Permanent marker
- Pencil
- Paint pen
- Paintbrushes
- Rulers
- Drills

Facilities

- ECE 350 Lab
- The Foundry at Gross Hall
- Innovation Co-Lab
- Rubenstein Arts Center Co-Lab