ECE 350: Digital Systems Final Project Report

Gouttham Chandrasekar, Martin Muenster

April 24, 2019

Duke Community Standard

By submitting this LaTeX document, I affirm that

1. I understand that each `git` commit I create in this repository is a submission
2. I affirm that each submission complies with the Duke Community Standard and the guidelines set forth for this assignment
3. I further acknowledge that any content not included in this commit under the version control system cannot be considered as a part of my submission.
4. Finally, I understand that a submission is considered submitted when it has been pushed to the server.

Contents

Introduction

Our Final project was a culmination of everything we learned this semester in ECE 350. The labs taught us how to interface the FPGA and possible components to our project such as the VGA and Controller. The lectures and assignments taught us how to create a pipelined processor which we then tweaked and used to create our game. In the process of brainstorming, for the end product of the project, it was important to make sure that the project would utilize main parts, circuitry (hardware), IO interface with Verilog and the processor. After a couple weeks of brainstorming, we came up with PONG HERO! Not to be mistaken by pong or guitar hero, PONG HERO is a game that combines mechanics from both games to create a unique and fun experience. Our game comes complete with 2 3D printed, PCBed guitars an FPGA and VGA screen for absolutely no extra charge. We heavily used our processor and wrote most of our logic in MIPS. Behavioral Verilog was used exclusively for displaying our objects on the screen.

Game Functionality and Rubric

Rubric

Task	Points
Basic Pong Implementation	10
Basic Guitar Hero Implementation	20
Guitar Controllers (PCB)	20
Visuals	20
(Single Player Mode) Psuedo Random AI	10
(Multiplayer Mode) Powerups/Interaction / Pong/Guitar Hero Integration	10

Task Explanation

Basic Pong Implementation

- Ball, and Paddles movements are smooth. Collision logic between the different components working as expected. (Ball bounce off paddles and top/bottom walls. Ball respawns if it passes the side walls.
- (Game Mechanics) Player loses a life if the ball gets past the paddles.

Basic Guitar Hero Implementation

- Notes moving from a starting at one of three guitar strings towards a target. Multiple notes travelling at the same time. 1 Button corresponds to each string on guitar. Use buttons to hit notes.
- Notes are moving smoothly on the screen and disappear when they are hit.
- Notes disappear when they go out of bounds.
- Some form of keeping track of score.
- 3 types of successful note-hits. (Perfect - 3pts, average - 1pt, imperfect -0) Notes speed up as time progresses

PongHero Integration

- Pong is played in the bottom half of screen. Guitar Hero in the top half of the screen.
- Basic AI used to control pong paddles automatically.
- A missed note results in AI pausing the paddle stopping for a period of time. Goal to hit notes perfectly to keep paddles moving and stay in the game for longer than your opponent.

Guitar Controller

- Designed by us and 3D printed in 5 segments each.
- Uses three push buttons to detect notes and two limit switches to detect strums.
- Includes a PCB designed and printed by us.

Visuals

- Pong Hero logo, as well as a welcome screen and a game over screen.
- Guitars on top match the designs and color of our physical guitars.
- Colored note images travel down the string, a fire effect is added after a certain amount of time.
- Number of lives and powerup status displayed on-screen.
- Circles light up on screen when respective note is strummed.

Psuedo Random Single Player Mode

- Implemented in MIIPS. Randomizes when the AI in single player misses a note.

MultiPlayer Mode (Interaction between the players)

- Use perfect hits to rack up points. These points can be used to activate powerups
- Powerup1: Speed up the ball when it goes towards the opponent
- Powerup2: Reduce opponents paddles reaction time.
- If you have more points than the other player, you receive a small speed boost.

Overall Project Design

Architectural Pattern

The flow of data and separation of dependencies, present in the source code of this project, allow an operation that follows a Model-View-Controller work-flow. In this case, the MIPS assembly code acts as the model, calculating a variety of values and maintaining the flow of the game. These values are then passed to the skeleton (the controller in this scenario) and rerouted to different modules. The skeleton holds instances of a processor, regfile, and vga_controller passing values from one to the other, as previously described for the movement of data. The 'view' is the vga_controller, which contains logic to multiplex between different pixel color values. The color of the pixel, depending on the location for it on the screen, is a function of timing, location data from the processor, and additional register values from the processor's regfile.

Model / MIPS Code Design

In this section, I will explain general details about our approach to writing MIPS code for our game. The mips code that we wrote controlled all of the mechanics of the game. Initially, it was difficult to get started because as we were writing MIPS code for the game, we were slowly uncovering all of the bugs in our processor design. It was a slow and grueling process initally getting a ball moving on the screen because we weren't ever sure if our MIPS code was incorrectly written or if our processor was designed poorly. After a couple days of debugging, we had a fully working processor. From then on, we focused on writing modular and readable MIPS code. We commented our code religiously and had bookmarks in our code everywhere to make it as readable as possible. We also organized our dmem.mif file and our registers s.t. we would never run into problems with overwriting important registers. This made our code easier to debug as our codebase expanded.

In this section, I will explain our dmem organization. In dmem, I allocated locations 1-100 for constants such as vga screen dimensions and paddle/ball size and initialization values such as where the paddle/ball starts at the begining of the game. Dmem location 100-199 store values that change as the game progresses. Finally, dmem locations 200-299 store note color (which string the note appears on), and the gametime of the notes (when they appear in the song).

The registers were organized to save as much register space as possible to maximize the scalability of our project. We knew that being liberal with our register could only limit the scope of our project so we created a register storage design to maximize the amount of game-info we could send to our vga-controller. In our design, registers 1-5 were trash registers, 6-10 more important trash registers, 11-14 were game items, (paddle, ball), 15-19 were jal registers, and 20-23 stored registers information. Finally, register 29 and 31 stored return address information for functions. This left us with 5 extra usable registers. Each note register stored 2 notes, location and hit information and perfect hit information. Each game item register stored location and velocity x[31:21] y[20:10] xvel[9:5] and yvel[4:0].

The code is segmented into 5 sections, "choose screen type, bufferloop, parse registers, gameloop (edit registers), store registers. The registers that stored game item information was passed into the vga-controller so that the pictures can be displayed in the correct location. The processor runs on the 50 Megahertz clock

cycle which is much faster than the processor. To prevent our gameloop code from running too fast, we have a bufferloop that counts to 1 million for each gameloop cycle. This slows down our gameloop s.t. it updates around the same pace as our vga screen updates. The splash screen, and the game over screen is also a loop that keep running until the user requests to be in another state. The parse register section of our code parses the input information (guitar inputs) from the user for every game loop cycle and stores each property into a unique dmem address. That way, I don't need to worry about isolating the properties I need from the registers using sll and sra. This improved the debugging and readibility of our code tremendously. The gameloop first updated the vga controller register, stored inputs into dmem (from the guitar input register, incremented gametime, figured out the notespeed based on the gametime, moved notes, activated powerups, moved the ball, and then moved the paddles. Each action was isolated into seperate functions. We also modularized our code s.t. we could easily add more notes and objects into our game. This is because we had a seperate function for each unique game object.

View / VGA Controller

The view for our project can be seen in the image above. We knew that there was only a limited amount of memory that we could use in our project without connecting an SD card. Therefore, we reduced the amt of space required for our graphics by limiting the number of colors in our design. Additionally, all of our repeated images (each paddle, all the notes) all pointer to a singular picture. This was a pretty tricky task to figure out but we will able to successfully implement it. Also, our main PONGHERO was hand drawn. We had moving images, different designs for each screen and special effects such as fire around the notes when they moved at a certain velocity.

Controller / Guitar

The controller, which was the Guitars were each 3d printed, spray painted and the circuits used to control the guitars were PCBed. We used buttons for the different string colors and a couple limit switches for the strum. The controller was designed in Fusion and the PCB was design in Eagle. We then connected the output wires to the FPGA to control the two players.