ECE385

Fall 2015

Final Project

Slime Games in System Verilog

Saurav Puri (scpuri2), Nachiket Joshi (nsjoshi4)
Section: AB8 at 9:00AM – 11:50 AM, Wednesday
TAs: Xinying Wang, Jong Bin Lim

Contents

1 Purpose of Circuit

The purpose of this circuit is to simulate a fun and common internet game where two players interact with a ball. We usually see these type of games created in JAVA or Flash, so we want to try and create it using FPGA. The game we decided to create is similar to the popular game "Slime Games." Slime Games was a collection of games where one or two players would play sports (basketball, volleyball, soccer, bowling, cricket, and different variations) with very limited graphics. The player models are semi-circles and the ball is very basic.

We decided to make basketball and wallball. Since we were limited by the number of keyboard inputs, due to the amount of workload, we modified the Slime Games mechanics to have one player with full motion, and a second player who's objective is to block the other player from scoring.

2 About Slime Games

Slime Games was one of our favorite games growing up. It was fun because of its simplistic but competitive nature. We first found slime games on a popular flash games website Miniclip. The games are based around very simple movements and trying to hit the ball depending on the sport. The following images are taken from our game, and illustrate the gameflow and sprites.

3 Description of Circuit

The circuit runs on an Altera Cyclone IV e FPGA. The system can be broken down into a few major components: Qsys for circuit design, System Verilog Code on Quartus and C Code on Eclipse. These components sections will be broken down further to their corresponding functions and modules later.

We designed our players and ball to each act independently of each other first. The first goal was to have both players be able to move in response to the keyboard and the on-board FPGA keys. The ball would independently move without reacting to any user input. We also had to implement gravity for all objects in the game, since the game is designed to simulate real-life sports in a very basic manor.

Figure 1: The l-o-n-g caption for all the subfigures (FirstFigure through FourthFigure) goes here.

4 Code

4.1 Code in Quartus

SystemVerilog code is split up into modules, which all interact with each other in a main, overarching, module. Descriptions of each module used

in the project follow. The behavior that the modules combine together to provide will be detailed in the following section.

4.1.1 Module: gamelogic

This module is the finite state machine responsible for the transitions between the various states in the game. In addition to handling the multiple states of the game, this module used to track the score and reflect it to the hex displays on the FPGA through the HexDriver modules, however, this implementation was later moved into the puck module, as it was easier to compute.

4.1.2 Module: puck

This is the ball used for basketball. Collisions between the basketball and all the other objects on the screen occur within this module itself. The ball knows the location and size of both players on the field and will deflect accordingly. In addition to collisions, the ball knows when it is within the scoring region and used to output a signal to the gamelogic module which will update the score and send the updated score to the appropriate locations. The score updating has now been moved to this module, because it was easier to keep track of its behavior.

4.1.3 Module: wallball

This module is very similar to puck, in that the reflections are the same. The only difference between the two modules are the zones where the player can score. Here, wallball can only score when it hits the rightmost wall and the score will reset when it hits the ground.

4.1.4 Module: ball

The naming of this module is a holdover from the time when we were using this code for lab 8. In actuality this is the module for the main player. This module contains logic that moves the character around within certain bounds as well as limitations imposed on the player based on the position relative to the boundaries of the stage. The player is prevented from wrapping around the stage to the other side or dropping through the floor to reappear from the ceiling.This player is also the one that shoots the ball, but the collisions are not calculated within this module.

4.1.5 Module: slime2

This is the other character model, the one that acts as a defender. For the most part this module is identical to the other player module, except for how the character moves. Whereas the first character model is able to move across most of the play field, this character can only jump up and down to three different heights. This module is also does not detect collision and is detected in puck module.

4.1.6 Module: lab8

This is the top level module. It instantiates all other modules and also connects Quartus to Qsys and Eclipse.

4.1.7 Module: sprite_table

This module contains the multitude of sprites that we use throughout the game. The sprites are mainly simple text prompts saved as 3 dimensional arrays. They are then passed to the colormapper module which will control when the sprite is displayed.

4.1.8 Module: ColorMapper

This module performs the object and shape rendering as well as the coloring for the many objects and sprites we have. The module takes input from the FSM based on which state it is in. Based on the state, it draws specific sprites to the screen.

4.1.9 Module: VGA_controller

This module communicates the Quartus to the VGA monitor that will be used to display the game. It outputs a horizontal sync and a vertical sync timing signals which is sent throughout other modules to make sure everything is synchronous.

4.1.10 Module: HexDriver

This module takes a 4 bit input and outputs a hex character on a display on the FPGA. We use this output for two purposes: keeping track of scoring and seeing the code of the key pressed on the keyboard. The score of the

basketball game was output onto HEX6 and HEX7 while wallball was output onto HEX4 and HEX5.

4.1.11 Module: hpi_io_intf

This module is a communication module to Qsys CY7C67200 interface. This is a handshake type of procedure between Qsys and Eclipse, and the output is then sent to Quartus. The HPI in simple terms, reaches into a few select registers which then reach into memory. This is how keycodes are stored and retrieved in memory.

4.1.12 Module: lab8_soc.v

This module is the System-on-a-Chip descriptor for the NIOS II CPU and SDRAM. This module outputs the memory values for other modules to use and HPI values to the hpi_io module.

4.2 Qsys Code

4.2.1 jtag_uart_0

The jtag_uart_0 is a method to communicate between Eclipse and the FPGA. The jtag_uart_0 adds the ability to debug using C code rather than Quartus, which also enables us to use print statements which output to the Eclipse terminal.

4.2.2 PIO

- *Keycode* - outputs the users keyboard input. This input is then sent to Quartus.
- *otg_hpi_cs* - outputs a chip-select signal which is used to select which HPI register is used.
- *otg_hpi_address* - outputs the address of which register is being used.
- *otg_hpi_data* - is the 16 bit in-out port that is used for data transmission between HPI and SDRAM
- *otg_hpi_r* - read signal

- *otg_hpi_w* - write signal

4.2.3 nios2_qsys_0

This is the Nios II embedded processor. It is a simple CPU that can be programmed using high level languages. We used C in specific. The Nios II is connected to Eclipse through Quartus so that we can control the CPU through control signals.

4.2.4 SDRAM

The SDRAM is a synchronous dynamic RAM. It is a quick access type of memory and is the only on-chip type of memory we use. Eclipse is able to access the SDRAM using addresses because we have full control of the size and location through Qsys.

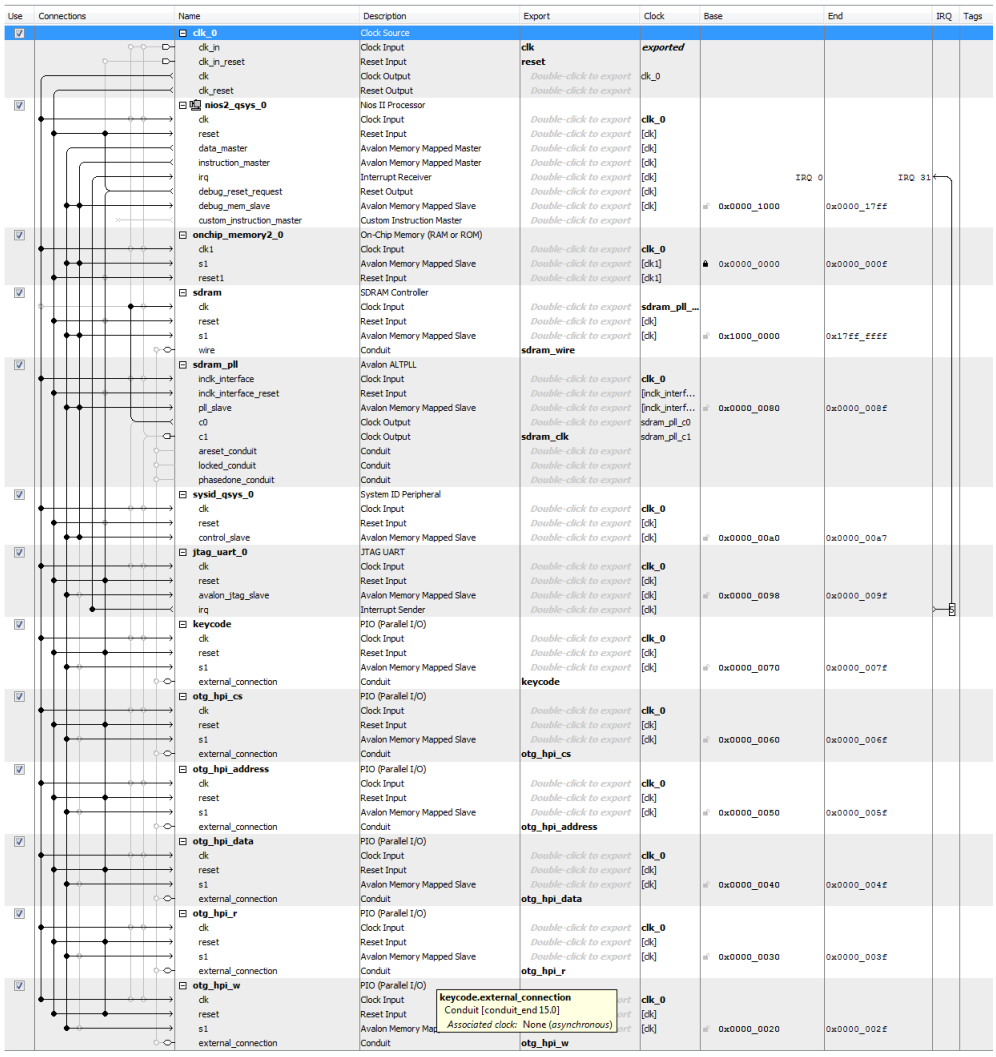

Figure 2: Qsys Schematic

4.3 Code in Eclipse

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

4.3.1 io_handler

- *io_read*: This function triggers the appropriate signals in C to activate the hardware to read a value from a specified register in the HPI.
- *io_write*: This function triggers the appropriate signals in C to activate the hardware to write a value to a specified register in the HPI.

4.3.2 usb_handler

- *usb_read*: This function calls *io_write* to write the address to access to HPI_ADDR and then calls *io_read* on HPI_DATA and returns the data read from that register.
- *usb_write*: This function calls *io_write* to write the address to access to HPI_ADDR and then calls *io_write* on HPI_DATA.

5 Circuit Operation

This section deals with the overarching behaviors of the circuit with reference to the modules used in providing those behaviors.

5.1 Behavior: Graphics

All the graphics used in this game were sprites. We wanted to maintain simplicity because Slime Games had very simple graphics. We also wanted to be original in our designs so we modeled everything ourselves.

The sprites were then printed to the VGA by the colormapper module. This module used an If-Elseif method to display the sprites. It also took inputs from the gamelogic module FSM to check which sprites should be shown.

5.2 Behavior: Player Movement

Player movement in done in two methods in separate modules. One player is controlled by the keyboard inputs while the other player is controlled by the keys on the FPGA. We split the controls because of the limited communication between the keyboard and Nios.

Restrictions on the player movement were required to reduce the number of bugs. We limited the players to stay on the screen and have a height limit. Originally the players were only able to jump once by a certain number of pixels, however this caused an issue with ball collisions, so we changed it to allow the players to infinitely jump while having the number of pixels lowered per jump. The second player does not have any side to side movement, but can jump at varied speeds. This is made so that the game stays balanced and is not too easy.

5.3 Behavior: Ball Movement

The ball movement is controlled in one main module. The main basketball is the puck module and the wallball is the wallball module. The ball movement has a lot of code because most of the calculations are purely done in these modules. The ball movement is completely independent of user input, except for when the player collides with the ball. The ball has a natural tendency to fall to the ground (gravity) which will be explained in the next section.

The ball movement is limited to stay within the screen, however sometimes glitches out for unknown reasons. When the ball happens to glitch out, it will immediately be respawned so the game is not ruined. The ball also will bounce downwards when hitting the left, right or top boundary. This is done because it makes the game a little more controllable. When the ball hits the ground, it will be immediately respawned. Currently there is no penalty for having the ball touch the ground, but we are open to any ideas that will not be too punishing.

5.4 Behavior: Gravity

Gravity is controlled by having a constant movement downwards when a check condition sees if the ball is in the air. We split the position changing function to include two variables, motion and input. Motion incorporates

the native movement of the ball, so boundary conditions and gravity, and input incorporates the collision checks with the player models.

6 Problems and Solutions

Throughout the course of the final project, we encountered several obstacles that hindered our progress. In this section, we detail some of the problems we faced and the workarounds and solutions that we implemented to overcome them.

6.1 Multiple Keyboard Inputs

Initially we had planned on using the keyboard to control both characters on the screen. One player would control using the arrow keys, and the other would use WASD. We came across an issue where we could not send more than two simultaneous key presses to the NIOS II controller and the subsequent control schemes. Our trial and error process was as follows:

- We'll have the USB driver write multiple times to the base address and hope that the NIOS II will pick up the full 2 bytes of keyboard data we wanted to read properly. This did not work.
- We'll try increasing the size of the appropriate PIOs to 32 bits and hope the NIOS II will deal with the change in the manner that we want in an appropriate manner. This did not work because NIOS II only had ports that were 16 bits wide, we did not have control about how much data could be passed.
- The final solution was to change how we're looking at the problem and see if we can avoid the issue coming up in the first place. We then changed our game from a 2 player basketball to a 2 player hit and block type of basketball.

Our solution was to take a step back and think about how we could turn this to our advantage. We couldn't come up with a solution that would allow up to four simultaneous keystrokes at a time, so we turned to the FPGA to supply us with extra inputs. Of the four push buttons on the FPGA, we were only using one: This was a hard reset for the entire system. Since we had three push buttons at our disposal, we decided to change the game from two

players moving about the entire screen to a penalty shoot out style of game where one character could move around most of the screen, but the other character would be limited to only being able to jump to various heights and attempt to block the ball.

6.2 Collisions and Reflections

Our original plan for handling the reflections was two-fold: first detect that there was a collision between the ball and a player where a reflection would take place, and then perform the appropriate changes in motion to the ball. We quickly encountered a major issue where we couldn't figure out how to implement the math necessary to calculate the appropriate reflections to the degree that we would have liked. Originally, it was going to be as in physics where the reflection occurs tangent to the curved surface at the point of contact. This idea was a little too ambitious, so we had to change that idea. Our compromise was to split the player into three zones and have set rebounds for each zone. The middle third of the player would reflect the ball directly back up, maintaining the motion of the ball in the X direction. For the left and right sections of the player, we reversed the Y motion and sent the ball off with a fixed speed in the appropriate X direction. This was primarily used for testing and initial game play. Subsequently, when most of the bugs were ironed out, we increased the number of zones to better simulate the reflections off a curved surface. The zones were increased to four different faces on each side of the slime. This approach simplified the math where we just divided the slime semi-circle into equal zones, and each zone had a different reflection.

6.3 Implementing Gravity

Gravity was probably the hardest part of this project. It was difficult because we were trying to reduce the number of variables per object, but in the end, this started to overwrite different motions from collisions and player inputs. Our solution to this error was to make two different changes of the objects positions. One was _Motion and the other was _Input in both x and y directions. _Motion dealt with the gravity and boundary bouncing. While _Input dealt with all the user inputs and bounces off the players model.

7 Borrowed Work

There is no way that we could have gotten everything done on our own without the efforts of other people.

Apart from the custom sprites for the basketball hoop and the title sprite, all fonts used are based off of the font file provided.

Our project uses the lab 8 coursework as a base upon which we created Slime Games.

8 Timing and Map Report

main_clk	85.32 MHz
altera_clk	133.69 MHz
LUT	5863
DSP	40
Memory (BRAM)	55296 bits
Flip-Flop	2385
Static Power	102.42 mW
Dynamic Power	38.03 mW
Total Power	239.22 mW

9 Block Diagrams and State Machines

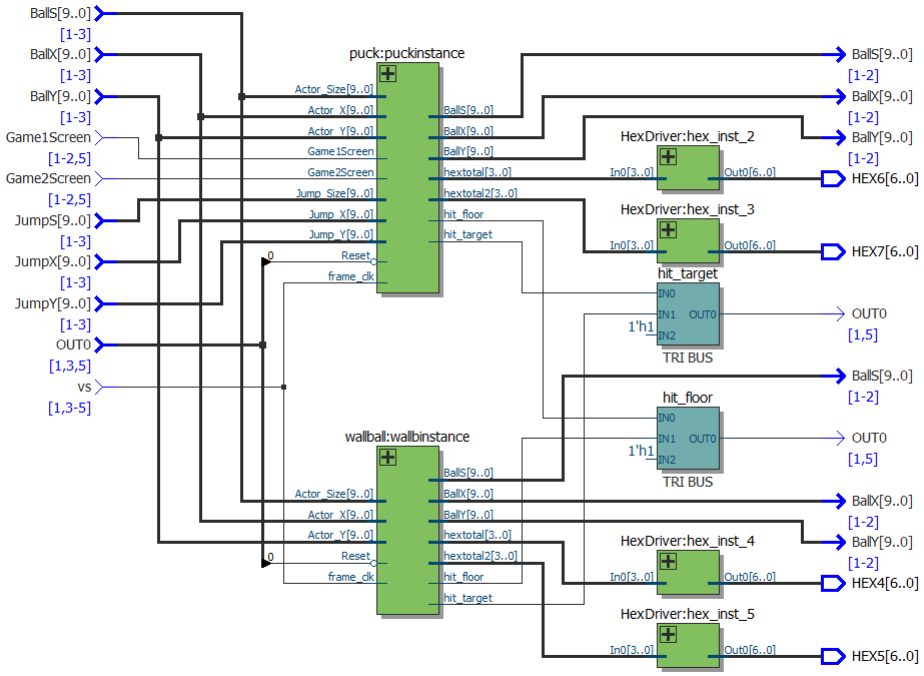

Figure 3: Block Diagram of Top Level Module (lab8)

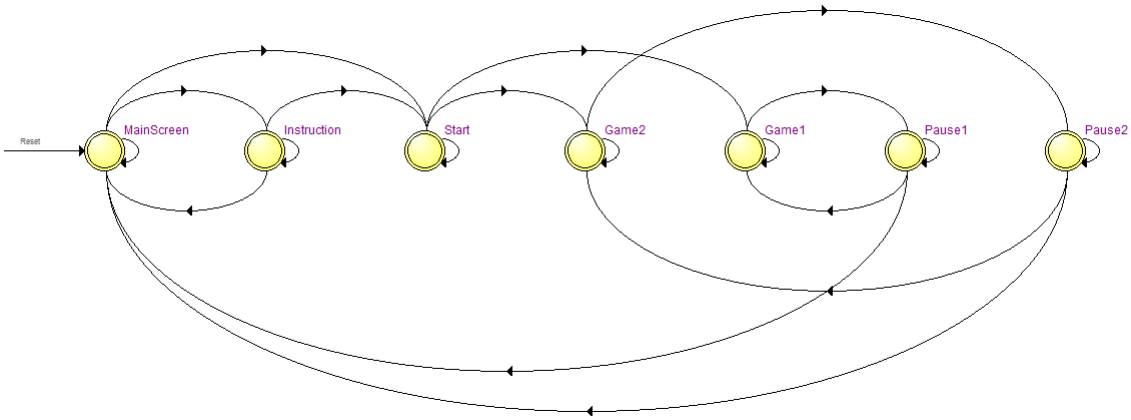

Figure 4: State Diagram of Game Logic

10 Conclusions

It always feels like this project is nowhere near completion, but we did not have nearly enough time to implement everything that we wanted. Ideas such as audio, better graphics, fewer glitches, multiple levels, and AI. But after finishing everything that we did, we do not feel like there is anything lacking. The game is fun to play and debug. The glitches look more like game mechanics rather than errors in design. Before beginning this project, it seemed like it would just be a mess of 2000 lines of code; however, we learned so much more about FPGA, Quartus, Qsys, and Eclipse than we expected. Even though the softwares would act funky and crash plenty of times during compilation, we learned how to persevere through these quirks. In the end, we did meet our goal to make a fun game and closely match Slime Games with our own twists.