Aradhana Adhikari
Victoria Ouyang
Davis Tran

6.111 Proposal DDR Whack-A-Mole

1.0 Project Overview

For our final project, we intend to build a DDR-style Whack-A-Mole game. The setup of the game is to have a 3 x 3 square board with 8 buttons (the center square is for the player to stand on) that each correspond to a location on the screen where a mole could potentially pop up. The goal of the game is for the player to step on the corresponding button when a mole pops up in a certain allotted amount of time. Each time the player successfully whacks a mole, the scoring mechanism will increase. If a mole is missed, the player will lose a life, and when all lives are lost, the game is over. There will be five main modules in this game: foot buttons, game logic, graphics, sound, and memory.

2.0 Block Diagram

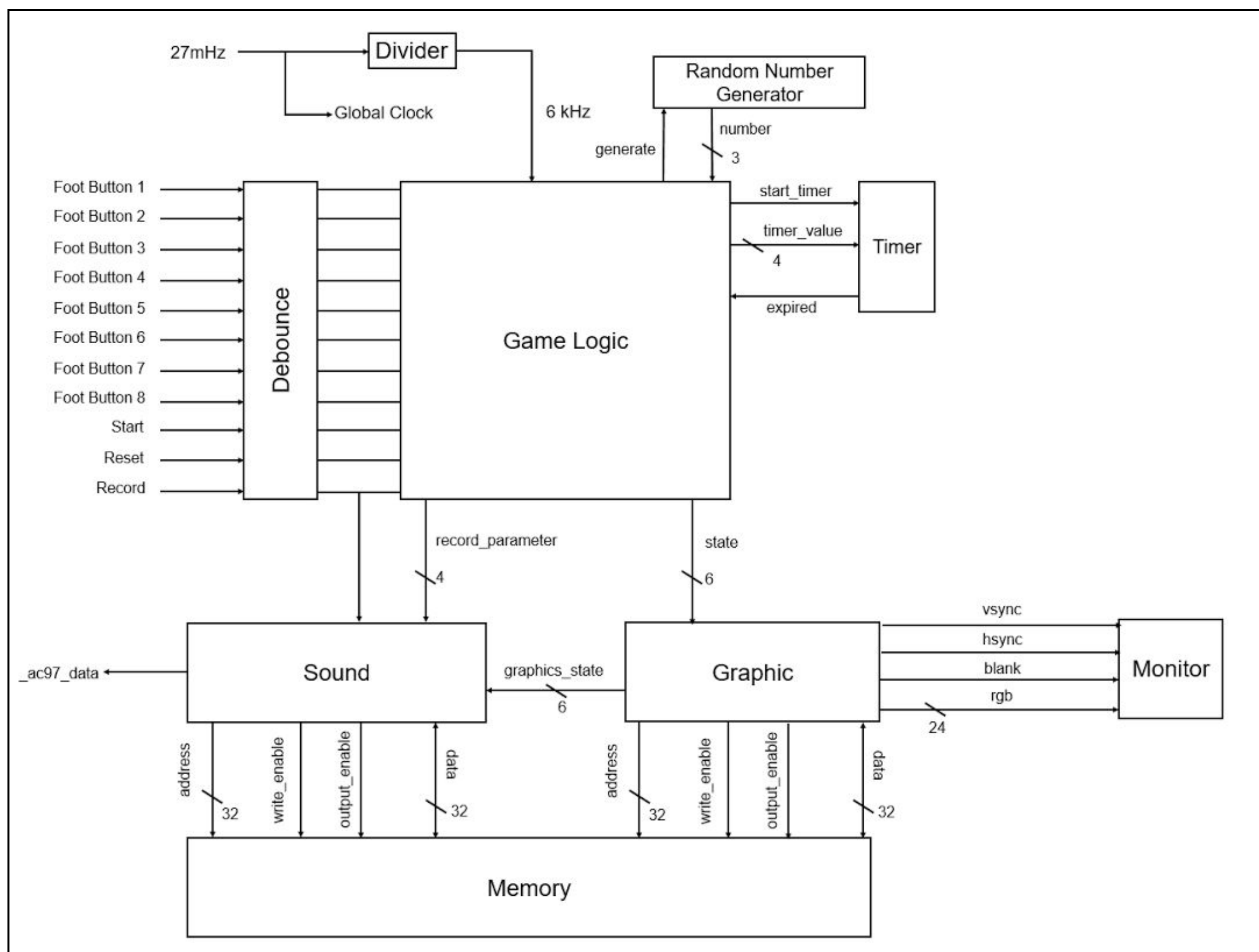

Figure 1: Block diagram of Whack-A-Mole Game System.

3.0 Project Implementation Details

3.1 Foot Buttons

The foot buttons module handles the signals coming from the DDR pad that we will build, and sends information to the game logic module. We intend to build our own simple DDR pad since it would be easier to obtain the signals from it compared to a commercial pad available for purchase. Details about the mechanical aspects are shown below. Each button has two states, represented by 1-bit on/off. *Foot Buttons 1 - 8* are passed through a debounce module and sent to the game logic module. To test this module, we can have each button assigned to an LED on the FPGA, and when any button is pressed, the corresponding LED should light up to verify functionality.

The foot buttons will be arranged in a 3 x 3 square with no button at the center. The board will be supported by 2 by 4 wooden posts arranged like a Tic-Tac-Toe board. The center square will be a fixed wooden plate while the eight other squares are connected to the foot buttons via springs. The wooden Tic-Tac-Toe board provides a hard stop when the player depresses the plates. The system is depicted below in Figure 2.

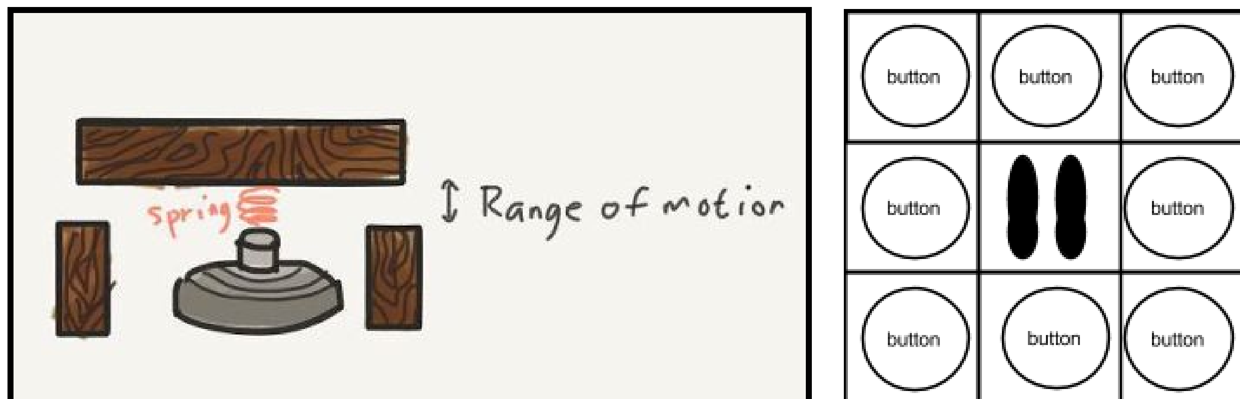

Figure 2: Left: Cross-sectional view of construction for one space in the 3x3 Grid. Right: Overhead view of the board. The player stands on the middle square.

3.2 Game Logic

The game logic module handles debounced input from the foot buttons and actively updates the game state. A 1-bit *start* button input begins the game. As the game is played, the player's score and lives are stored as registers. The score is incremented when a player successfully steps on a mole. Moles are generated on one of the eight stompable squares. The mole location is generated by sending a 1-bit *generate* signal to obtain a 1-bit location *number* from the random number generator module. The mole is displayed for a predetermined amount of time on an external monitor at the determined location. When the mole is created, the game logic module outputs 1-bit *start_timer* and 4-bit *timer_value* signals to a timer module. The mole is removed when the timer module asserts a 1-bit *expired* signal or the player steps on the button

corresponding to that mole. If the player did not step on the mole before the *expired* signal was asserted, the lives counter is decremented. If the player successfully whacked the mole, their score is incremented. A game over screen will display when the player runs out of lives.

To manage graphics, the game logic module sends a 6-bit *state* to the graphics module, which handles graphics and conveys information to the sounds module. When recording, the game logic module sends a 4-bit *record_parameter* signal to the sound module.

To test the game logic module, we can use buttons on the FPGA to represent the foot button inputs and send *state*, *record_parameter*, and *timer_value* signals to a hex display and all the binary inputs to leds. A test bench can also be created in ISE to simulate the game logic FSM. The test bench provides input foot button signals and the waveform outputs can be viewed in ModelSim.

3.3 Graphics

The graphics module will handle all monitor display aspects of the project. The module takes in an 6-bit *state* input from the game logic module and sends an 8-bit *address* along with 1-bit *write-enable* and 1-bit *output-enable* to the memory module. The memory module then sends a 32-bit *data* signal back to the graphics module. The data is a saved image of the specific graphics for a particular game state. We will start with graphics for moles and later incorporate more variety such as different types of moles that have different durations of appearance time, moles that start laughing when game is over, etc. With this data, the graphics module sends 1-bit *vsync*, *hsync*, and *blank* signals, as well as a 24-bit RGB value *rgb* to the monitor. The graphics of the moles will then be displayed on the monitor.

To test this module, we can use switches on the FPGA to represent the different states, and have certain graphics associated to each state saved in memory. For each state, we can check that the address being outputted to memory is indeed correct, and that the displayed graphics on the monitor is as expected.

3.4 Sound

The sound module will handle all audio aspects of the project. There will be a 1-bit *record* input going to the module which will be coming from a button. This will be used to record audio pieces that needs to be stored to memory (so that it can be used later in game). There will also be a 4-bit *record_parameter* input going into the sound module to indicate parameter selection (this will be used to figure out where in memory to store the recorded audio). For now, *record_parameter* is a 4-bit input, but this can be changed based on how many different audio recording the project will require. There will also be a 6-bit *graphics_state* input coming into the sound module from the graphics module. This input will be used in determining the memory location of the audio piece that needs to be played. Based on this information, an 32-bit output *address* will be sent to the memory module, along with 1-bit *write_enable*, 1-bit *output_enable*,

and the memory module will return the 32-bit audio *data* output as in input to this sound module. The audio coming from memory will be the data that is sent through to_ac97_data.

This sound module can be tested in two ways. One way to test is by creating testbench and simulating module with module inputs and checking to make sure the expected addresses are outputted through *address*. In addition to this, the overall system (which includes the sound module with the memory module) can also check by ear to check that correct sounds are being outputted based on the inputs.

3.5 Memory

The memory module will be used to store audio and images used in the project. Data retention without power is needed for this project (as music and graphics should not be reloaded every single time the game starts up). So, the plan is to use flash memory. This module will have inputs such as 32-bit *address*, 1-bit *write_enable* , 1-bit *output_enable*, and inout 32-bit data. The *data* will be used as an input in case of a write, and an output for read. This module should have at least two different modes: a write-mode for when data can be stored to memory, and a read-only mode for when a game is in play and modules will utilize the memory module to get necessary information. This module should also have a way of handling multiple modules trying to access memory, as the graphics module and sound module both require use of memory (potentially with an fsm).

To test this module, testbenches can incorporate different combinations of clearing memory, writing to memory, and reading from memory to make sure the read, write, and clear functions are working as expected. The memory module can also be tested in combination with the sound and graphics modules to check that the overall system is responding correctly given set inputs.

4.0 Extra Implementation

Once the basic functionality of the project has been implemented (graphics with one mole, with simple background audio, and correct game response to inputs), more complexity can be introduced to the project in the following ways:

1. Different sound effect depending on mole hit or miss
2. Mole graphic changes depending on hit or miss
3. Incorporating levels to the game play: multiple moles, faster moles
4. Making game more like DDR (full length music background with mole location used to indicate the “arrows” that needs to be hit), which needs entire system to be fast enough that system response to changing inputs are almost instantaneous
5. Scoreboard display based on game play time, (or counter on number of hits or misses if game more like DDR)

5.0 Project Modules Work Distribution

1. Foot buttons - Davis and Victoria
2. Game logic (and timer) - Davis
3. Graphics - Victoria
4. Sound - Ara
5. Memory - Ara