ECE 540 Final Project: Labyrinth Game Using FPGA and Accelerometer by

Colten Nye, Hoa Quach, and Mark Ronay

Our team’s project was to implement a Labyrinth game. The intention of the game is to navigate a ball around walls and hazards to a goal point, using the board’s accelerometer as control. The ball moves in conjunction to the degree and direction of the tilt of the board.

The design we used to implement our game was done entirely in hardware. There are three ROM files, one for the map, one for the ball and one for the win screen, which are tied together by locks of combinatorial logic. The overall structure of our design can be broken into three parts and owes much of its form to our division of labor.

The block diagram below describes the structure and interface.

1. **Block Diagram**

Accelerometer Controls

SPI

Accelerator Arithmetic

ADXL362

Score/Timer

7 Seg Display

Accelerator ticker for ball control

Ball behavior

Maze wrapper

Maze ROM

Binary to BCD

VGA

dtg

Colorizer

Ball icon ROM wrapper

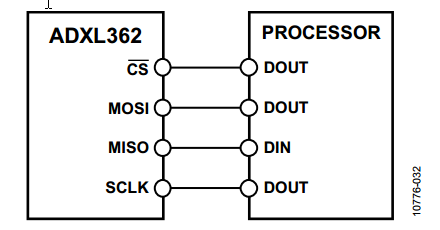
Ball ROM

img ROM wrapper

Img ROM

1. Accelerometer:

We were aided in our efforts to get useful output from the accelerometer by the demo files included with the Nexys4 board. These files, used unaltered in our final design, are AccelerometerCtl.vhd, AccelArithmetics.vhd, ADXL362Ctrl.vhd and a SPI interface called SPI.vhd. They were all written in VHDL, which hindered our ability to alter the files in our time frame. Also we wanted to treat them as external soft IPs and don’t want to make any changes beside the absolute necessaries to enable functionalities. The top level file AccelerometerCtl.vhd provides interface with our top level module an AccelX out, an AccelY out and Accel Mag out and Accel Tmp out. The latter two outputs were not used in our project. The demo code itself deprecates the AccelZ out signal, which would otherwise have been available for a more ambitious three dimensional game. The AccelerometerCtl interfaces with the ADXL362 accelerometer via a Serial Peripheral Interface (SPI) and a 512 bit FIFO.



A lower level module, AccelArithmetics.vhd manipulates the raw accelerometer output. The default output over the SPI wire comes in 12 bit signed twos compliment binary strings. AccelArithmetics module does several things to this output: It inverts X and Y, it truncates the output to 9bit twos complement strings (in order to fit in the demo program VGA display box), and returns the total magnitude of the acceleration by summing the squares of the 3 returned vectors and then square rooting them with an IP block.

Our initial strategy for controlling the accelerometer was to replace the AccelArithmetics with a module that would output specific movement modes, corresponding to 8 discrete speeds and four directions for our ball to take. However considerable time was spent trying to rectify reset polarity levels through the levels of hierarchy, which was very confusing because we were passing through an unfamiliar VHDL level. We decided our limited time budget could be best allocated if we kept all the original VHDL code and manipulated its output elsewhere in our program where we had complete control over the hierarchy. The tradeoff for doing this was that we were saddled with unneeded signals and IP from the AccelerometerCtl top level module.

accel\_threshold\_ticker.v

This module is responsible for translating accelerometer input to movement signals appropriate for our ball. It recieves as input the accelX and accelY signals, which come in the form of 9 bit twos compliment binary strings. The module takes the top bit to determine if X and Y are positive or negative, then compares the lower 7 bits to predefined magnatude thresholds.

The thresholds are used to define a new top count for a tick generator – which will write out ball control signals at increasing rates for increasing tilts. After some play testing we settled on 4 closely spaced speeds, though this can be changed to adjust the play feel and challenge of the game. Care had to be taken to stagger X and Y output signals to meet requirements set by the collision checker, which cannot check move legality in both directions at once. There are two clocks, one for X and one for Y, staggered by 100 clock cycles. This stagger is large enough to ensure the collision checker has time to process a signal, and small enough to be unnoticible to the player.

The pseudo code for the ticker is as follows:

Ticker algorithm

If clk\_cnt equals count top\_x

Assert tick\_x

Set top\_x to new\_top\_x

If x is positive & increment is asserted & tick\_x is asserted& tick\_ y is not asserted

increment x

If x is negative & increment is asserted & tick\_x is asserted& tick\_ y is not asserted

decrement x

If X axis tilt magnaute is below threshold 1

Do not increment X

Else if X axis tilt magnatude is below theshold 2

Set new\_top\_x to count top\_1

Else if X axis tilt magnatude is below theshold 3

Set new\_top\_x to count top\_2

Else if X axis tilt magnatude is below theshold 2

Set new\_top\_x to count top\_3

Else

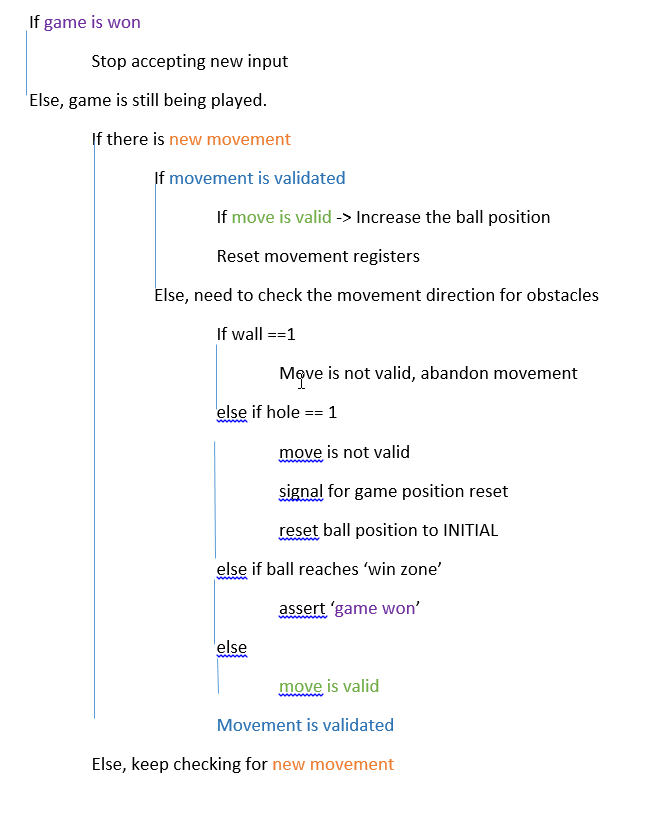
Assert tick\_X when clock counts to count top\_4

Assert increment

Repeat algorithm for Y

ball\_vid\_buttons.v

Keeps track of a virtual ball. Receives direction signals from the ticker, calculates if that move can be made, and what impact that move has on the game (hit a hole, made it to the endzone). Each move is made one pixel at a time, in one of the four cardinal basic directions. In order to determine if a move can be made, the module must check every pixel along the face of the direction to move in. If none of the pixels are a wall, the move is valid. If any of the pixels results in a hole or the endzone, appropriate measures are taken. This module also holds the ROM that contains the maze information.



1. **Visual ROM & VGA implementation:**

Much of the visual organizations and functionalities are reused from Project 2. The Maze ROM with its wrapper are inside the ‘Ball’ module to be convenient for the ball to check its position on the maze. The icon and winner image are instantiated inside the VGA subsystem.

We wanted to have a 640x480 pixel ‘fullscreen’ graphic for the entire visual system but because a block RAM with Xilinx IP generator is limited to 262144 deep (18bit address) and 640x480 requires 307200 address, we had to compromise with the maze being 1/2 the resolution (320x240). The maze is then scale up 2x by shifting the address in to the ROM right 1 posistion (so that the address has to increases by 2 to index to the next pixel). The other ROM for the icon and win-image smaller and could be stored at full resolution (15x15 for the ball and 240x 240 for the win-image)

The images used in the program are adapted from stock images or drawn by hand. The images iwere edited using Microsoft Paint to make an image with somewhat solid lines and colors. This is especially important for the maze since the bot uses the color to check for obstacles and requires sharp edges with little or no anti-alias. The images are saved in 256 colors BMP file format.

We utilized a simple MATLAB script to generate the initial draft for the ROMs’ coefficient (coe) files (http://www.mathworks.com/matlabcentral/fileexchange/12437-bmptocoe). This script takes in .bmp file and generates coefficient files which use 8-bit pixel width (8 bit RBG color). Bit [7:6] are blue, [5:3] are green and [2:0] are for red. The resulted coefficient files are then edited to give a more accurate colors on the Nexys4 12bit color scheme.

The ball icon is drawn in the same fashion as in Project 2. Whenever the vertical sync and the horizontal sync signals are matched those of the ball’s x-y coordinates, the wrapper would index in to the icon ROM to get the RBG value for the VGA.

The congratulation/win image is also work similar to the ball icon. The image is 240x 240 pixels stored in a single port ROM. Instead of checking for a match of the horizontal and vertical sync signals as in the ball icon (since the image does not move), the image is only displayed when the ball reaches the ‘win’ condition and asserts the ‘win’ signal. When the wrapper receives this signal, it indexes into the ROM and sent out the pixel information as a function of sync signal + offset (to center the image)

Results:

The result of our efforts is a playable and fun game requiring some skill, but using an intuitive interface. We weren’t able to implement any of the reach goals outlined in our original proposal, however the finished product stands up very well on its own. We have four speeds which, after some play testing, we tied to four close together and shallow tilts. It turns out nobodies intuition is to turn the board entirely on its side to effect a move, and so high degrees of speed and corresponding tilt thresholds were deemed unnecessary. However, they could easily be added if the game were to be expanded with new levels. We have a single maze, with a single win condition and one start position. However, by the nature of the design additional levels could easily be added. There are four types of surface with which the ball interacts, ‘wall’, which blocks movement, ‘hole’ which triggers a game over signal and sends the ball back to start, open floor, over which the ball may freely traverse in response to board tilt, and ‘goal’ which triggers the win condition - a display of a gratifying icon of success.

Because our ball collision algorithm is fairly general it can be modified to accommodate different triggers for different map features without too much effort.

Individual member contributions.

The team was very cooperative in sharing responsibilities on initial discovery and debugging. During the main phase of the project Hoa Quach took primary responsibility for the VGA and score modules, along with some behavioral coding. Mark Ronay took primary responsibility for the Accelerator outputs and translating them to movement modes. Colten Nye took primary responsibility for ball behavior and collision checking along with compatibility between Accelerator and Collision checker.