Our teams project was the implement a Labyrinth game. The intention of the game is to navigate a ball around walls and hazards to a goal point, using the boards 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 the structure and interface.

CHECK BOT SIM THEORY OF OPERATION

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. The top level file AccelerometerCtl.vhd provides interface with our top level module via a SPI interface, an AccelX out, an AccelY out and Accel Mag out and Accel Tmp out. The later two outputs were not used in our project. The demo code itself depricates the AccelZ out signal, which would otherwise have been available for a more ambitious three dimensional game.

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 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 trade off 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.

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.

Here should be a block diagram.

Results:

The result of our efforts is a playable and fun game requiring some skill, but using an intuitive interface. We werent 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.