ECE 540 FINAL PROJECT REPORT

Fall 2018

Project:

Toadly aMazing

Submitted by:
Jonathan Anchell
Chelsea Brooks
Michael Bourquin

Table of Contents

Project Summary

For our final project, we created a video game maze using the Nexys 4 board and a VGA screen. The game is simple: steer your character towards the finish line and don't touch the walls. However, the game isn't controlled by push-buttons, as in the former half of project 2, but by the accelerometer built into the Nexys 4 board. The game has three mazes. Each maze harder than the level before. If the user touches the wall they must start over at the beginning of the maze in that level. The user has 3 lives in each maze. If the user touches the black wall three times, the game is over and the game goes back to level 1. If the player gets the red line, the maze shows a screen that indicates the level is complete. Once the player wins level3, the screen says "you win"

Team Contribution

Hardware - Michael and Chelsea

Software - Jonathan, Michael, and Chelsea

World Maps/Icon - Chelsea

Project write up – Jonathan, Chelsea, and Michael

Hardware

To get the robot to move smoothly on the screen using the accelerometer, a driver module for the accelerometer chip was was written and instantiated at the top level. The outputs are passed through to MIPS so that the software can read the x, y, z values. Also, instead of using the rojobot from project 2, a simplified maze bot was designed that has set x and y speeds plus a direction control. This was needed so that the robot does not need to turn before going a different direction, but instead turns "instantaneously". It is directly controlled by the software. There is a level selector which is also controlled by the software. All new or modified(from project 2) source source code can be found under */source/hardware.

The following is a breakdown of the hardware additions for this project. Each will be described in more depth in the following sections. Contributors are shown in parenthesis.

Top Level Design (Everyone):

- Accelerometer (Engineer: Michael Bourquin, Contributors: Jonathan Anchell, Chelsea Brooks)
  - Driver Module
  - Synchronizer
  - Testbench
- Maze Bot (Jonathan Anchell and Michael Bourquin)
- Level Select (Chelsea Brooks)
- MIPS System (Everyone)

Accelerometer:

The on-chip accelerometer (ADXL362) was used for the purposes of this project and two modules were derived using the datasheet. The SPI_driver module outputs a roundDD signal telling the spi_sync module that x,y, and z have been updated and the current temp values are valid on the line. The roundDD

signal is held high until the driver begins writing new values to the temp x,y,z signals (approximately 8 SPI clock cycles), leaving plenty of time for the values to synchronize. When roundDD is high, output values from the SPI driver are saved into the MIPS sys registers. The spi_sync module also holds the reset signal low when a reset is detected giving SPI clock time to latch in the reset signal. This must be done because the SPI clock is much slower compared to the MIPS sys clock. The SPI clock runs at 5.23 MHZ while the MISP sys clock runs at 50MHZ. An overview of the accelerometer hardware can be seen in figure 1 below.

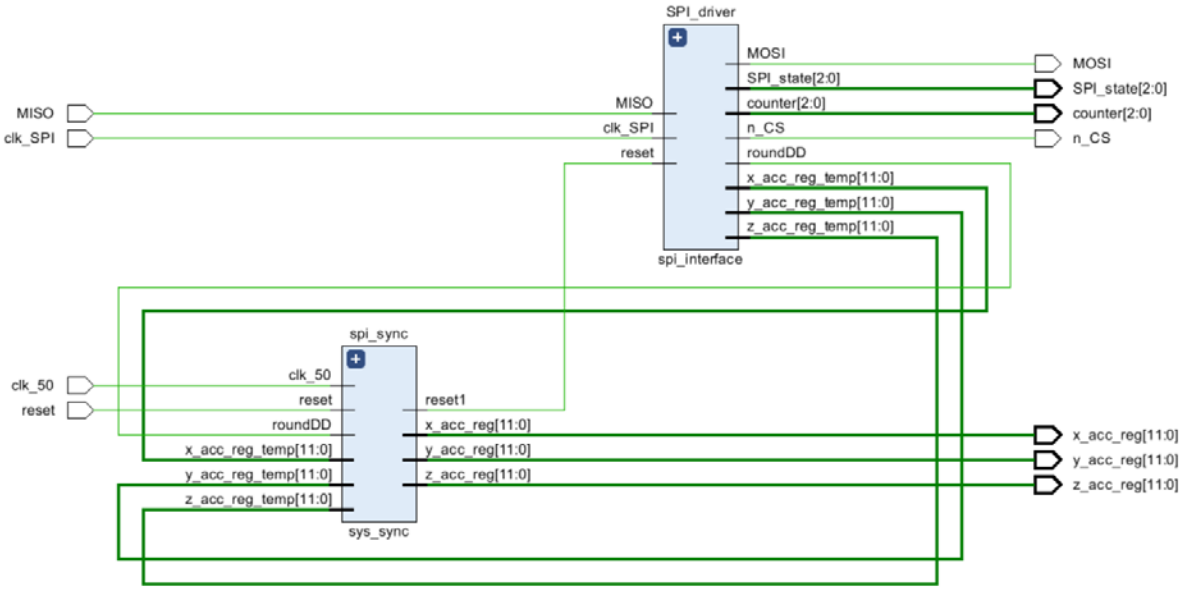


Fig. 1: Accelerometer Top Level

Driver Module:

The accelerometer driver module uses an FSM to read and write to the accelerometer chip. According to the datasheet there are 3 phases to each read and write. Each phase takes eight SPI clock cycles. They are the following in order.

- Instruction
- Address
- Data In/Out.

The accelerometer reads data on the positive edge and writes out data on the negative edge. The CS(Chip Select signal) is latched low to start a read or write and then latched high when finished. There must be time given before writing out the first instruction bit and the CS signal going low. There also must be time given for the CS signal to be held before starting another read or write. As the accelerometer chip writes out data on the negative edge, the driver module reads incoming data on the positive edge. It writes out data on the negative edge. An eight bit width register (accel_data) is used for data writes and saving incoming data reads as data bursts are 8 bits in length. It is re-initialized to zero at the end of every read/write.

Once a phase has begun, a counter is used to determine which bit to either save or write out in the accel_data register. Once an entire read is complete, the accel data register is saved to the appropriate bits in the x,y,z registers using a case statement. A register keeps track of which address is currently being read from and is the case statement input. On a reset, a write instruction is done to start the accelerometer

in measurement mode. This is followed by continuous reads from the x,y,z registers until another reset. Every time x,y,z is read, the roundDD signal is held high (signaling valid accelerometer values) until the next instruction phase is complete (about 8 clock cycles). Once the next data in phase is complete, the x,y,z registers will no longer be valid as first; only the first 8 bits are saved into 8 of the 12 bits of one of the x,y,z registers. They will not be valid again until the final bits in z are written.

Registers: Xlow(8bits), Xhigh(4 valid bits), Ylow(8bits), Yhigh(4 valid bits), Zlow(8bits), Zhigh(4 valid bits)

Instructions: Read and Write

From the above it can be seen that the accelerometer registers are a total of 12 bits in width each. Values are in two's complement with the MSB indicating negativity.

Synchronizer:

The accelerometer synchronizer module saves the accelerometer driver outputs every time they are updated and are valid (meaning they are not being updated in 8 bit bursts currently). When roundDD is high, the accelerometer driver outputs are valid. On detection of a system reset, the synchronizer also holds an output SPI reset register low for 64 clock cycles, giving the SPI module plenty of time to latch in a reset.

Testbench:

A testbench for the accelerometer driver was written previous to physically testing the system in order to make sure that the driver module matches the timing diagram from the ADXL362 datasheet as well as the chip select latch timing constraints. This saved time as it was much easier to see what was happening internally and also caused it to work on the first actual run.

The waveform output of the testbench can be seen in figure 2. MISO is the data input from the accelerometer chip (used on a write). It was tested with a constant 8 bit output of 0xFF. This waveform shows up until the final 12 bits are stored in the Y register. There is the initial write (measurement mode) followed by a read of X and Y. A read of X takes 16 clock cycles as both Xlow and Xhigh must be read to get the full 12 bits. All values are read in the default +-2g measurement mode. This is the most precise mode available. The yellow line shows the completion of the initial write.

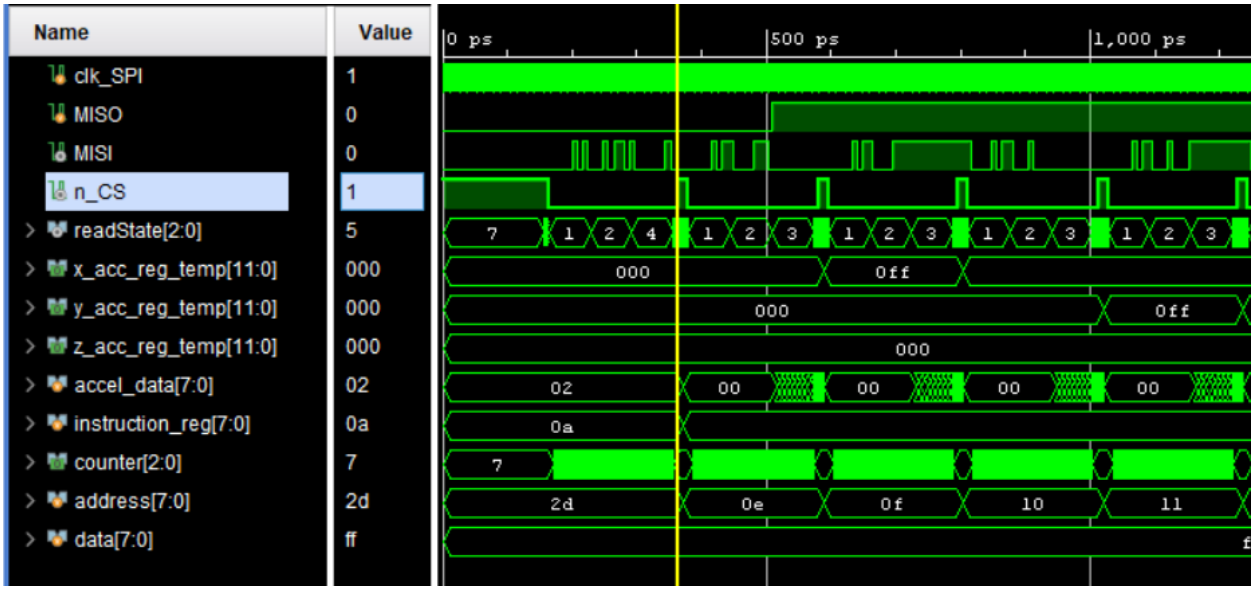


Fig 2: Accelerometer Driver Testbench

Maze Bot:

The toad icon is modeled in hardware via this module. The module keeps track of the current location of the bot on the screen and updates the position based on speed and direction inputs from the MIPS system. The clock is used to simulate movement. There are three different speeds and each has a different counter limit. The smaller this limit, the faster the robot will move across the screen. The robot can move faster in X than Y and vice versa as each have a separate speed input.

When the robot hits a wall, it enters a deadlock (wait state) where it cannot move and it is not visible on the screen. This acts as an interrupt for the software to do other things when the user has either hit a black wall (loses a life) or hits a red wall (wins the level/game). There is a soft reset signal that the software can write to, which resets the bot into its initial state.

Level Select:

There are three maze levels in the game, but we made seven total world maps that can be selected. In hardware, we added a new signal called LSEL. The hardware receives the LSEL signal in the GPIO module from software. The LSEL then travels back to the top level module. Inside mfp_nexys4_ddr.v we wrote a case statement that takes in the LSEL signal and decides which world map outputs to the colorizer and maze bot. The input to all the world maps are the same, the only difference are the outputs to the colorizer and maze bot location.

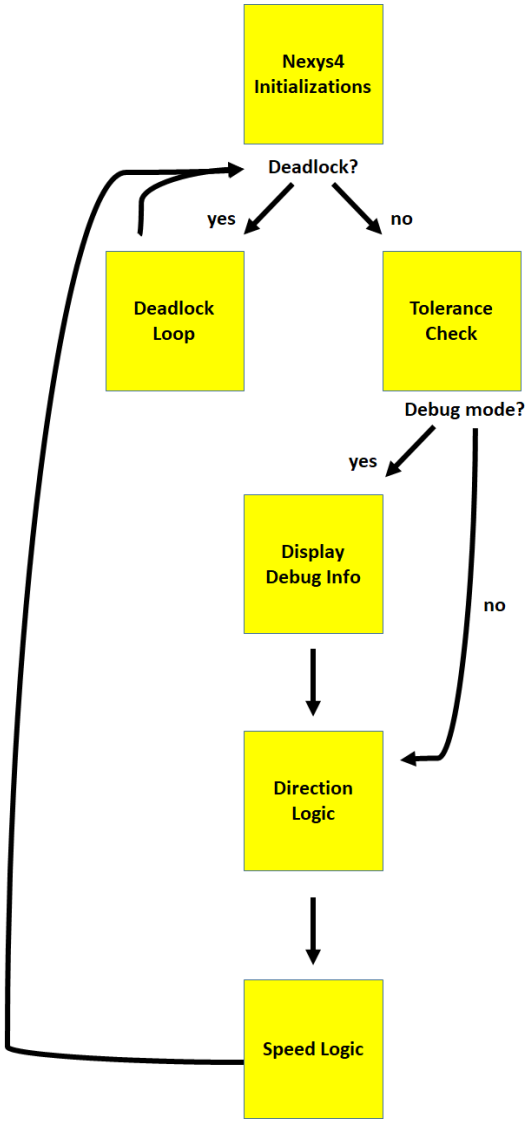
MIPS System Slave Additions:

The following additions to the GPIO slave were added to this project. The wires are passed through all the levels of mhp and passed to their parent modules when needed. The bot control bits were modified with each X/Y speed control taking 2 bits. Direction was unchanged taking the LS 3 bits. The lower 6 bits of bot info now output wall detection bits(3) and the internal bots direction.

Read Only: Accelerometer X, Y, Z Values
Read/Write: Level Select, Bot Control(Maze Bot), LEDS(was previously write only)
Write Only: Soft Reset(Maze Bot)

Software

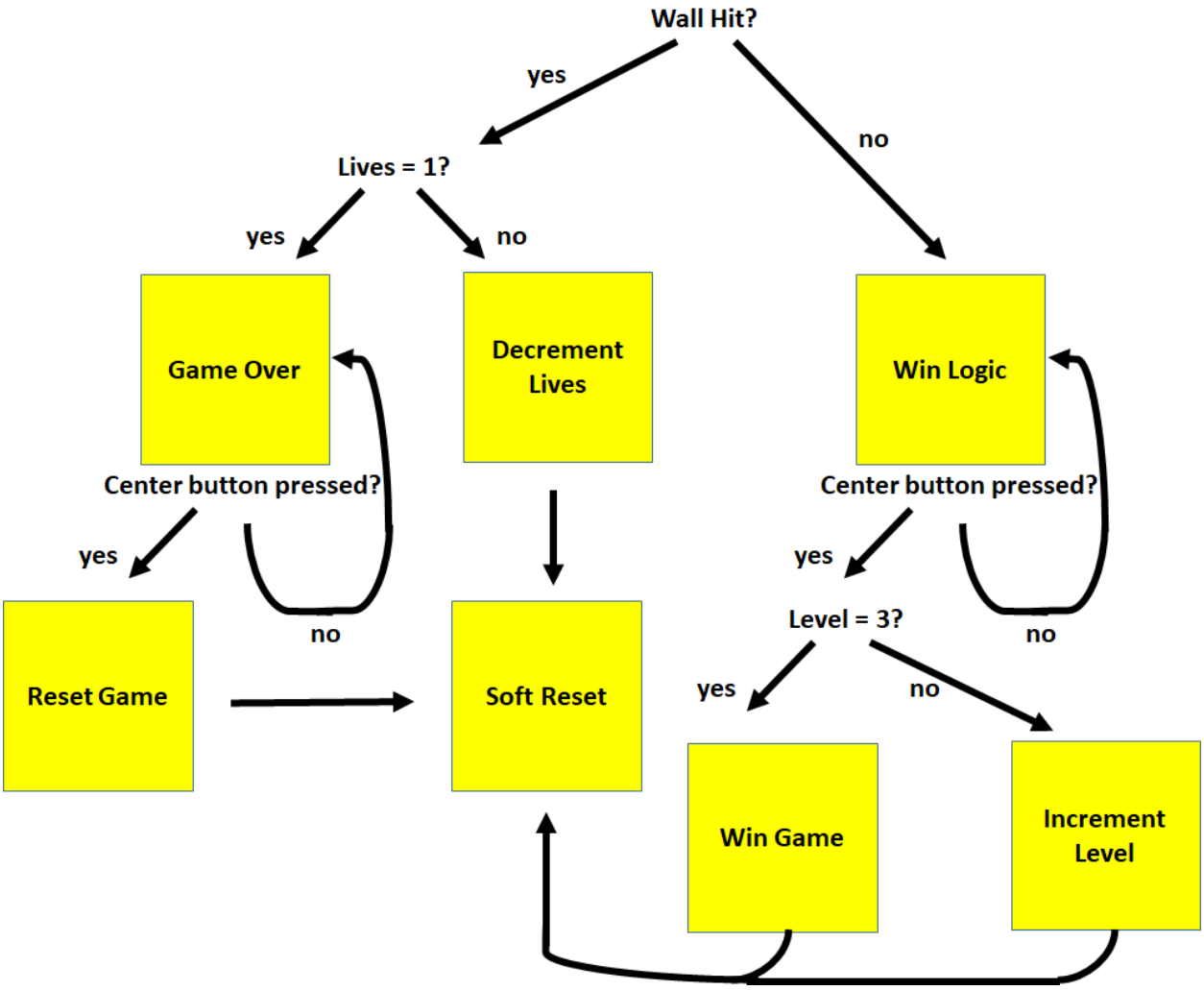
The MIPS assembly code runs in a large while loop. A high-level flow-chart of the program is shown below:



The program begins with a set of initializations that occur before the while loop. This includes enabling and zeroing both seven-segment displays, turning off all LEDs, writing the level 1 screen to the display,

and creating global variables for current acceleration, lives and health. After the Nexys4 input registers and global variables have been initialized, the program enters a forever-while loop.

The first check in the forever while loop is for deadlock. Deadlock occurs if the user crashes into a wall - either at the end of the level to advance to the next level or win the game, or into a maze wall to lose a life. A flow chart of the deadlock loop is shown below:



The event type that caused the wall hit is returned in the bot info register. If the user hit a wall, they either lose a life or, if all their lives are gone, are shown the game over screen. Screens are changed by writing pre-assigned numbers to a dedicated port in hardware. If the user has run out of lives, thy are stuck in the game over screen until they perform a soft-reset. A soft-reset has three components. First, it sets the icons direction to a default value of North. Second, it sets all icon speed values to zero so that the icon does not instantly begin moving once a new level is started. And third, it writes a 1 to the hardware soft reset input. This tells the hardware that a soft reset has occurred, and therefore to toggle its deadlock bit in the bot info register. This will keep the code from reentering the deadlock loop. If, instead, the user hits a maze wall but it is not their last life, a soft reset-occurs automatically without asking for user input.

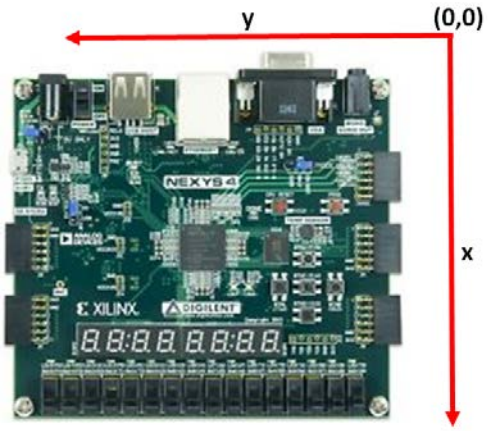
If the user hits a wall that is not a maze wall but is instead the end of the level, the codes enters the deadlock win logic. If the user was on level 3, the win screen is displayed. If the user is not on level 3, the next level screen is displayed. Both screens wait for user input with a soft-reset. If the user has won the game, number of lives and levels are reset.

After the deadlock check, the user enters the portion of the code responsible for measuring the x, y and z acceleration values from the accelerometer. These values vary from roughly 0-1000 milli-g (gravity). Values over 1000 are possible, but difficult to obtain unless you are tilting the board very quickly. To read the values, we simply have to read the values at three addesses passed from the Nexys 4. These numbers fluctuate, even if the accelerometer is sitting still. Therefore, a simple tolerance function was written to not update these values unless they were +/- 50 of the previous value.

After reading the acceleration values, the program will enter “debug mode” if switch 1 is set on the Nexys 4. Debug mode was incredibly useful for creating our project. It enabled us to view the acceleration values at any moment in time without having to set breakpoints. Among other things, this allowed us to select reasonable thresholds for direction and speed by testing in real time. Debug mode displays the absolute value of the x-acceleration on the bottom four seven-segment digits and the absolute value of the y-acceleration on the upper seven-segment digits. The sign of the tilt, negative or positive, is displayed on LEDs 0-3 which corresponding to the acceleration in x, y and z.

If the user did not select “debug mode”, “game mode” is entered. “Game mode” only differs from “debug mode” in what is displayed on the Nexys 4. Instead of acceleration values, current level is displayed to the seven-segment high digits and current lives are displayed to the seven-segment low digits. LEDs are not used in game mode.

Regardless if the user entered debug or game mode, the next phase in the forever-while loop is movement logic. In order to understand movement logic, it is important to note the sign of the acceleration values on the seven segment display.



Once we were able to determine which directions were positive in the x and y direction and a reasonable acceleration threshold was selected to begin movement (in debug mode), it was a straightforward task to write pseudo-code for the movement logic. The pseudo-code we used in our final implementation is shown below:

```
x_pos = positive(x_value) # convert to positive value if negative
y_pos = positive(y_value)


if (x_pos>300 && y_pos<300):
    if x_value is negative:
        orientation=North
    else if x_value is positive:
        orientation=South
else-if (x_pos<300 && y_pos>300)
    if y_value is negative:
        orientation=East
    else if y_value is positive:
        orientation=West

else-if (x_pos>300 && y_pos>300)
    if x_value is negative:
        if y_value is negative:
            orientation = NORTHEAST
        else if y_value is positive:
            orientation = NORTHWEST
    else if x_value is positive
        if y_value is negative:
            orientation = SOUTHEAST
        else if y_value is positive:
            orientation = SOUTHWEST
```

This code is all very straight-forward in pseudo-code, but was extremely messy in assembly. In order to calculate the positive x and y values, a simple function was created to calculate the two's-complement.

After the movement calculation, the speed of the icon was determined. Speed values were set according to the following table:

		x acceleration			
		<300	**>300 & < 400**	**> 400 and < 500**	**> 500**
y acceleration	**<300**	x stopped/ y stopped	x slow/ y stopped	x medium/ y stopped	x fast/ y stopped
	>300 & < 400	x stopped/ y slow	x slow/ y slow	x medium/ y slow	x fast/ y slow
	> 400 and < 500	x stopped/ y medium	x slow/ y medium	x medium/ y medium	x fast/ y medium
	> 500	x stopped/ y fast	x slow/ y fast	x medium/ y fast	x fast/ y fast

As shown by the table, x and y speeds to not need to share the same value. It is possible for the icon to move faster in the y-direction than the x-direction, for example, if the icon is heading northeast.

Pseudo-code for the speed was also very simple:

```
if (x_pos < 300):
    x_speed = STOP
if (x_pos >300 && x_pos < 400):
    x_speed = SLOW
if (x_pos >400 && x_pos < 500):
    x_speed = MEDIUM
if (x_pos > 500):
    x_speed = FAST

if (y_pos < 300):
    y_speed = STOP
if (y_pos >300 && y_pos < 400):
    y_speed = SLOW
if (y_pos >400 && y_pos < 500):
    y_speed = MEDIUM
if (y_pos > 500):
    y_speed = FAST
```

After direction and speed values were calculated, they were passed back to the hardware through the bot control register - the last step before the forever-while loop repeats.

World Map/Icon

The world map and icon were designed in https://www.piskelapp.com/ . We exported the design as a .c file. The c file was then converted into a .coe file using a perl script. With the .coe file, we used the random memory generator to convert the 7 maps. The level complete map shows up after you finish level 1 and 2. At the end of level 3, the screen displays you win. If you hit the black line aka wall of the maze, the game over screen is shown. The debug map was made to debug the screen transitions so we did not have to complete every maze every time. We chose toad from mario as our icon. The icon was also designed and exported from piskel. We use the internal memory for the icon instead of the random memory generator.

World Maps

Level 1

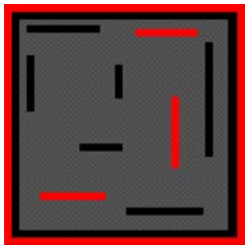
Level 2

Level 3

Level Complete

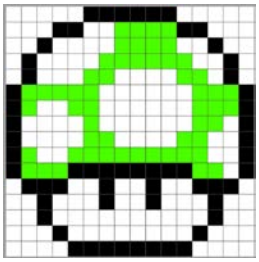
Game Over

You win



Debug

Icon



Challenges

- Working with different versions of Vivado
- Getting the speed to work as wanted. Took trial and error to find the right 3 speeds we wanted to use.

Github Link

https://github.com/codexhound/ECE540FinalProject