**ECE 385**

Fall 2014

Final Project

**Tron**

Caleb Klaus, Evan Dotterer

AB1/Friday 8AM

Benjamin Delay

## Purpose of Circuit

In this lab we implemented a simple game based on Tron. 2 players move around on the screen until somebody hits a wall or a spot where any player has previously been. There are 2 levels with 2 AI in the second level that chase around the players making it more difficult. A yellow ECE 385 logo appears at the end of the game in honor of this amazing class.

## Written Description of Circuit

Initially the two players each display an 8 by 8 pixel box on the screen, with no movement. Players are allowed to move in either the vertical or horizontal position, with diagonal movement prohibited. The game starts when one player starts moving. Movement is controlled by inputs from a PS2 keyboard, and code for the PS2 keyboard was given. Player 1 controls are W for up, A for left, S for down, and D for right motion. Player 2 controls are I for up, J for left, K for down, and L for right motion. Once a player hits one of these keys to start moving in a certain direction, that player will continually move in the same direction until another key is pressed. Each player’s path is drawn with the same color of that player’s head. Level 1 ends when a player hits his or his opponent’s path or one of the walls. When this happens, a sprite draws out the words level 2 on the screen. Once the Reset button is hit, the game enters levle2. In level 2, the game can also end when the AI catches up to the head of one of the players. The AI does not leave a path, and can travel through player’s paths, but moves at half the speed of the players, adding an additional challenge to each player.

The game board is implemented by a 60 by 80 array, with each position corresponding to an 8 by 8 pixel block on the screen. Wall positions are pre-determined and correspond to the value 2’b11 in the array. Player 1 and Player 2 positions and their paths are represented in the array as 2’b01 and 2’b10 respectively. 2’b00 is an empty space, which can be filled at any time. The array is continuously updated as player 1 and player 2 move around on the game board with lengthening paths. Because we are using such a big array which is continually being updated with hardware constraints this resulted in very large compile times around 30 minutes, but we think this is the best way to implement this game.

The yellow LEVEL 2 and ECE 385 logos which appear before level 2 and at the end of the game are implemented with 2 separate sprite tables, a 6 by 24 array; each position corresponding to an 8 by 8 pixel block on the screen. Whenever we are in between levels or at the end of the game, the output of the sprite module takes priority for drawing the screen, and the signal which keeps track of whether we are at level 1 or not decides which sprite is drawn on the screen. When this signal is 1’b1, the level 2 logo is drawn; otherwise the ECE 385 level is drawn.

The AI works by comparing the difference in X positions and Y positions with the corresponding player. If the difference in the X positions is greater, the AI will move in the X direction towards the player. If the difference in the Y positions is greater, the AI will move in the Y direction towards the player. The AI will never hit a wall because it is impossible for each player to go past the wall. There is a power-up available for the AI in the second level, where each player has the opportunity to speed up the opponent’s AI for 300 clock cycles, roughly 5 seconds. This is activated by the E and O keys for player 1 and 2 respectively. When activated, a counter is loaded with the value 300 and counts down every frame clock cycle to 0. The speed of the AI is thus equal to the speed of the player during this time, instead of just half the speed. Another signal keeps track of whether this power-up has been used; limiting each player to just one power-up use per level.

We will not include a wave simulation of our design as I believe this project is much better understood by actually playing the game and visually seeing things happening rather than looking at some wave signals on paper. So since we have our game working completely and is purely graphical, there is no reason for us to include a wave simulation.

## State Diagrams and Tables

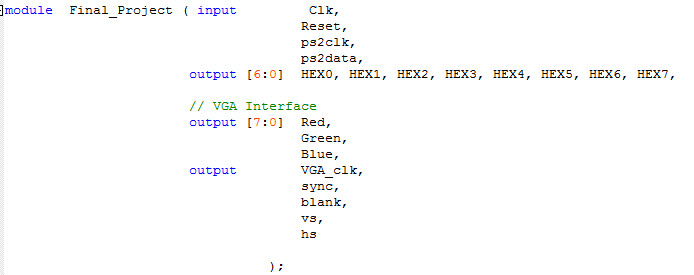
We did not use any state machines in our design, but a general flow of our design is as following. After the project.sof file is loaded onto the FPGA board, level1 begins. When level 1 ends after a collision, we go to a screen with the words LEVEL 2 being displayed. When the Reset button is hit, level 2 begins. When level 2 ends after a collision, we go to a screen with the words ECE 385 being displayed. When the Reset button is hit again, we go back to level 1. This cycle then continues.

## Design Statistics

|  |  |
| --- | --- |
| LUT | 50,188 |
| DSP | 0 |
| Memory (BRAM) | 0 |
| Flip-Flop | 9805 |
| Frequency | 103.84 MHz |
| Static Power | 104.67 mW |
| Dynamic Power | 309.81 mW |
| Total Power | 464.56 mW |

**Design Module Information**

// Top-level entity that combines all of the other modules and allows the circuit to function



//This module controls the movement and positions of player 1 and AI 1. Player one movement is updated whenever a player hits a corresponding key, and AI movement is updated based on the difference in the X and Y positions of the player and AI. Player 1 position is updated every clock cycle, while AI position is updated every 2 clock cycles, resulting in the AI moving at half the speed of the player, except when the power-up is active.



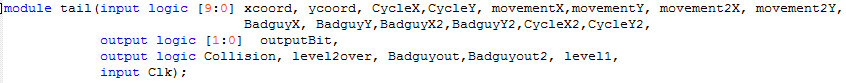
//This module controls the movement and positions of player 2 and AI 2. Player one movement is updated whenever a player hits a corresponding key, and AI movement is updated based on the difference in the X and Y positions of the player and AI. Player 1 position is updated every clock cycle, while AI position is updated every 2 clock cycles, resulting in the AI moving at half the speed of the player, except when the power-up is active.



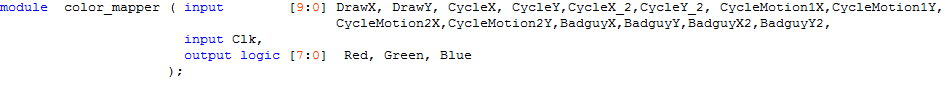
//This module contains the sprite tables, and the output depends on signal level1. When this signal is 1’b1, the level 2 sprite is outputted. Otherwise the ECE 385 sprite is outputted.



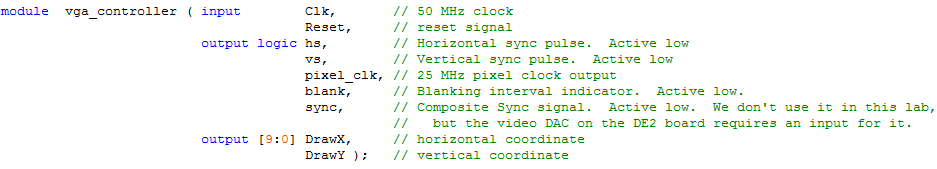
//This module is responsible for keeping track of which level we are on, as well as whether a collision has occurred. The game map array is also held in this module. Every time a player gets to a new position, we update the array with that player’s position. Every Player 1 position is updated with 1’b01, and player 2 position is updated with 1’b10. Wall positions hold the value 1’b11. Every time we attempt to update the array, we check to see whether that position already occupies a value other than 1’b00, which corresponds to an empty space. If it is occupied by a value other than 1’b00, it means a collision has occurred, and the collision signal is set to 1. The collision with the AI is also checked in this module by comparing the player and corresponding AI positions. If they are equal, collision is detected and game ends. Since each position in the array corresponds to an 8 by 8 pixel block, we had to divide the player and AI positions by 8 to get the correct game map array position. When Reset button is hit, all movement is set to zero, which causes a new map array to be declared, overwriting the old game map array.



//selects which colors need to be drawn on the screen, and then draws it out. If there has been a collision or level 2 is over, then the output from the lightcycle (sprite) module is drawn on the screen. Else if we are at level 2 and at the AI position, the AI is drawn. Otherwise the background is drawn, which is stored in the array that is constantly being updated, drawing out the walls and player paths.



//This module takes care of the vga display stuff and was given to us.



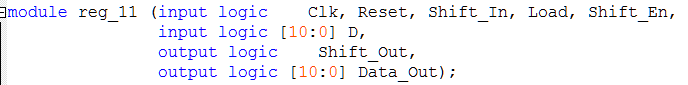
// outputs the input as its hex value to the seven segment display



//PS2 keyboard module which was given to us. This outputs the keycode which was pressed by the player.



//This module is for use with the keyboard module and was given to us. It is a 11 bit register with shift enable.



//This module is also for use with the keyboard module and was given to us. It is essentially a one bit register.



## Conclusions

This final project was a good way to wrap up this class, as it gave us freedom to explore an area that interested us while giving valuable experience with SystemVerilog coding. We feel like our programming skills as well as project skills greatly increased after working on this for over 30 hours in the last week. The greatest difficulty was debugging when compile time was around half an hour. This greatly limited the amount of times we could test our circuit causing a lot of frustration at times. We understand this is a hardware constraint and there is nothing we could do about it, but it would be nice if there was a way to test it without doing a full compile. Coming up with the AI was fun as we had never created an AI before and seeing everything work together at the end was pretty rewarding. It is a shame that we won’t be able to play the game anymore after turning in our FPGA boards, but we feel like our knowledge and experience gained during this project was definitely worth all the time spent on it. Overall we thought this was a very good course and we were glad to have the opportunity to take it.