Portland State University

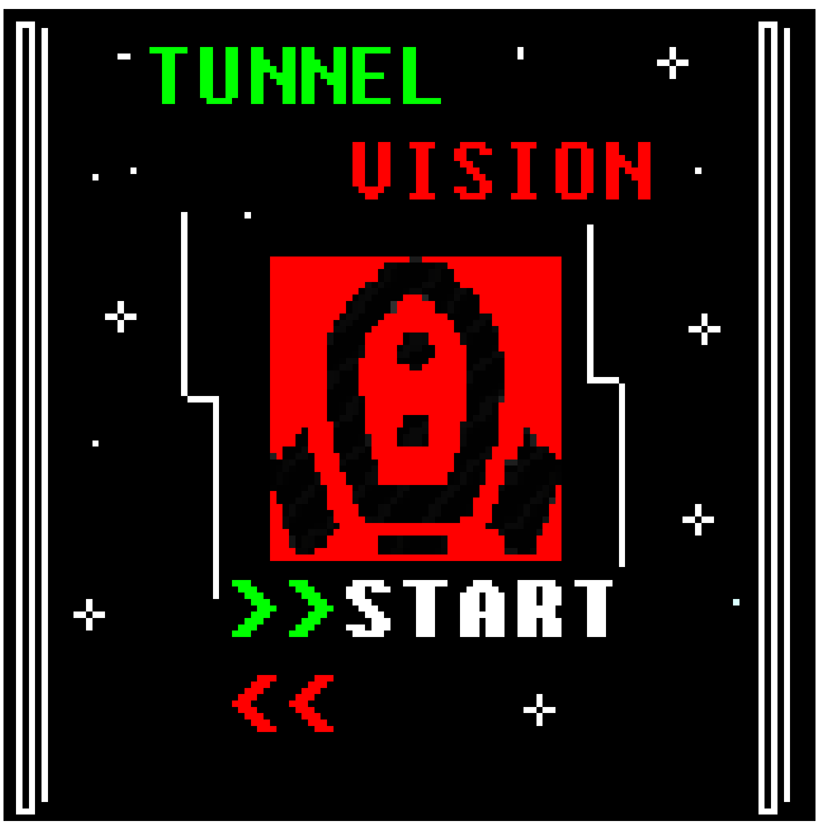
SoC Design With FPGAs

ECE540

# Tunnel Vision

Erik Rhodes Bhavana Dhulipala Rohan Deshpande Nikhil Patil

March 20, 2014



# Contents



# 1 Introduction

**Tunnel Vision** is a racing game that can be played on the **Xilinx Nexys3 FPGA** board and be displayed on a VGA monitor.

## 1.1 Gameplay

The player tries to avoid hitting the walls as it travels down the tunnel by moving the vehicle left and right. The space in between the walls steadily decreases until the player hits a wall or obstacle. The score is based on the amount of time the vehicle remains “alive”, and is displayed on the 7-segment display.

## 1.2 Controls

The player can move his vehicle by using the left and right pushbuttons on the Nexys3. When the game is over, hitting the middle button will reset the course. The top button starts the game and the bottom pushbutton pauses it. Different icons and speeds can be selected by toggling the switches on the board.

## 1.3 Features

**Tunnel Vision** features both starting and ending screens. The courses are generated randomly through a pseudo-random number generator. Additionally, the LEDs are lit with certain patterns depending on the action the player is taking. If the player selects the harder difficulty, the score is

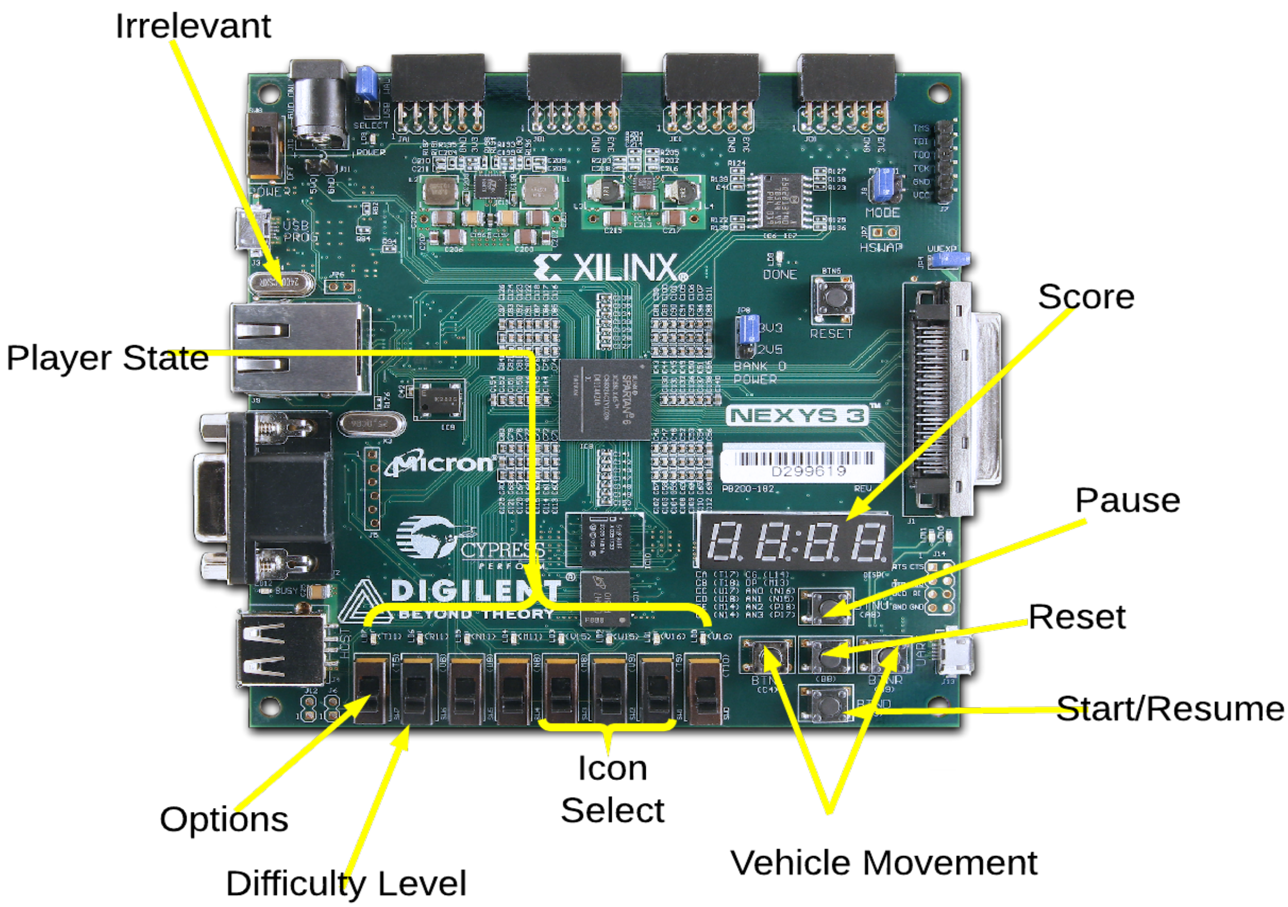



*Figure 1: Player Controls*

incremented at a faster rate and with a multiplier, awarding them a higher score for the same distance travelled.

## 2 Implementation

The information controlled in the **PicoBlaze** core is sent to the `Video_controller` module via the `game_interface` module. This information is constructed in an 8-bit register called `game_info`. The allocation of bits is seen in Figure 3. Collisions, obstacles, and icon changes were controlled in hardware. The video controller received the inputs of game_info register and the `LFSR` module, shifted the wall appropriately, and used the collision logic to stop the game and display the **Game Over** screen.

### 2.1 Software

In order to ensure the correct flow of game play, the push button inputs were checked in the other seen in Figure 4. Each button check performs approximately the same function. The pushbutton input is checked, all other bits not pertaining to the specific button are masked off, and the

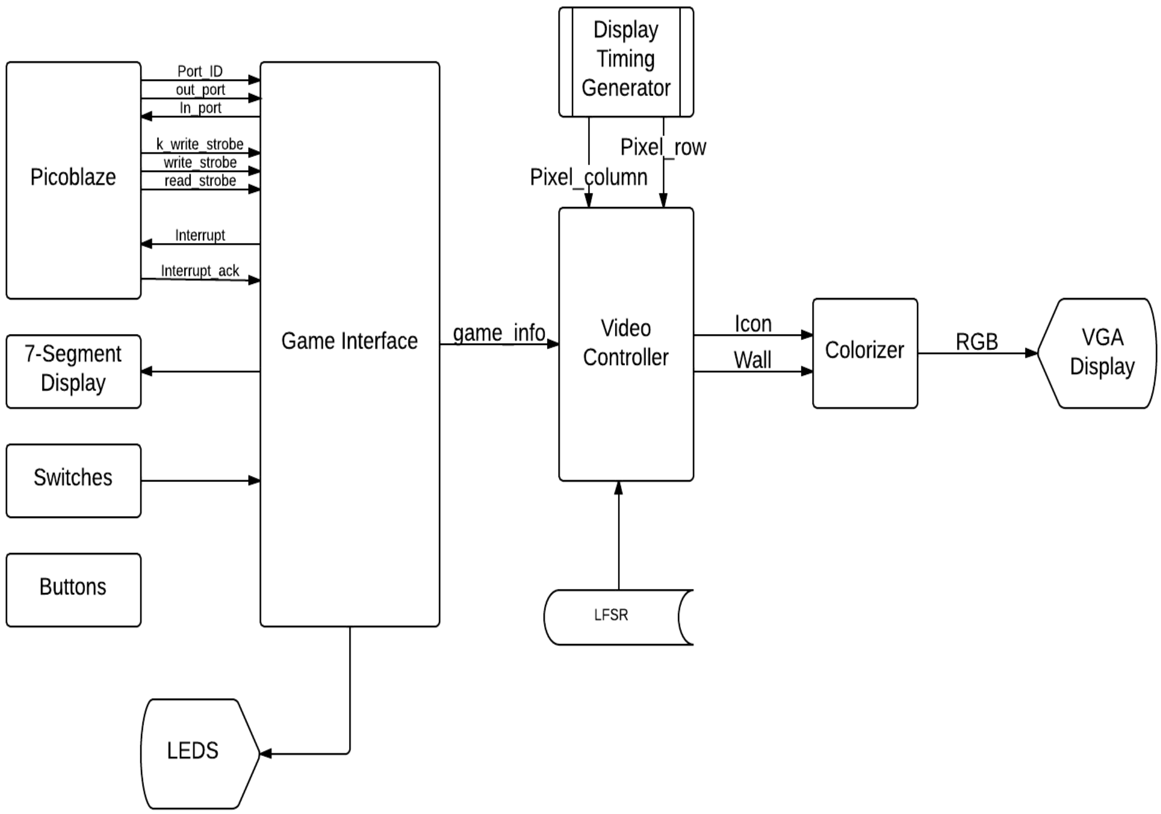



*Figure 2: Game play Block Diagram*

button value is checked. If it has not been pressed, the next button check is called. If it has been pressed, the state is updated accordingly and the LEDs output the desired pattern.

*Listing 1: Function checking if the top button has been pushed*

```
FETCH    s2 ,   SP_BTN          ;load saved debounced button
AND      s2 ,   MSK_BTN_UP      ;mask with up button
COMPARE  s2 ,   MSK_BTN_UP
JUMP     NZ,    chk_no_btn      ;go to next phase if no up button
LOAD     s3 ,   PAUSED_STATE    ;else pause and send out that register
STORE    s3 ,   STATE
FETCH    s3 ,   SP_GAME_INFO
LOAD     s4 ,   GAME_PAUSED     ;update game_info paused
AND      s3 ,   s4
STORE    s3 ,   SP_GAME_INFO
LOAD     s5 ,   FF              ;output LED pattern
OUTPUT   s5 ,   PA_LEDS
RETURN
```

If the game is active and another button than the top one has been pressed, the `active` function is called. This updates the state, score, and `game_info` output. In addition, when the left or right button has been pressed, the location of the vehicle is appropriately.

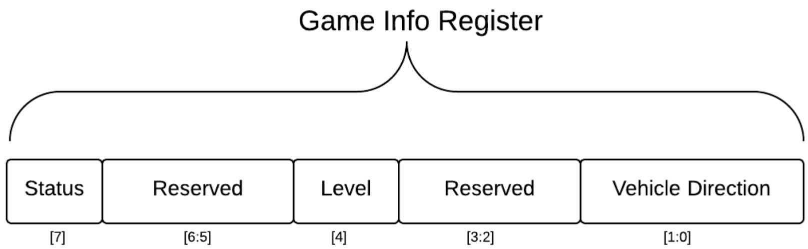



*Figure 3: Allocation of bits in game_info register*

*Listing 2: Update the state game info*

```
CALL   update_score
LOAD   s3 ,   ACTIVE_STATE
STORE  s3 ,   STATE
FETCH  s3 ,   SP_GAME_INFO
LOAD   s4 ,   GAME_ACTIVE
OR     s3 ,   s4
STORE  s3 ,   SP_GAME_INFO
RETURN
```

In order to show the score over 4 digits, the `update_score` function increments the least significant digit until 10 is reached. At this point, the next significant digit counter is called and the previous value cleared. This algorithm continues throughout all four digits. A separate function, `display_score`, was created to actually load the stored score values to be displayed. This structure was chosen to allow for the score to still update even if the 7-segment display is used for another output. Future modifications to the game may make use of this function.

The difficulty level of **Tunnel Vision** can be changed by toggling switch 4 (Figure 1). When the user has switched levels, the speed of the game increases. This information is also sent in the game_info register (Figure 3). The speed control is doubled, but the score increases at a 5:4 proportion. This scaling is accomplished by modifying the counter in the `game_interface` module that controls the speed of the interrupts. The speed is modified in the hardware portion.

# 3 Video Controller Implementation

The video controller module managed the icon switching and generation, the width and location of the walls, and the collision detection.

## 3.1 Wall movement

### 3.1.1 LFSR

### 3.1.2 Edge Detection

### 3.1.3 Width

The video controller takes 2 bits from the LFSR to generate a random direction that the walls will shift. If the value was 0 or 3, the walls did not shift at all.
Edge detection logic......

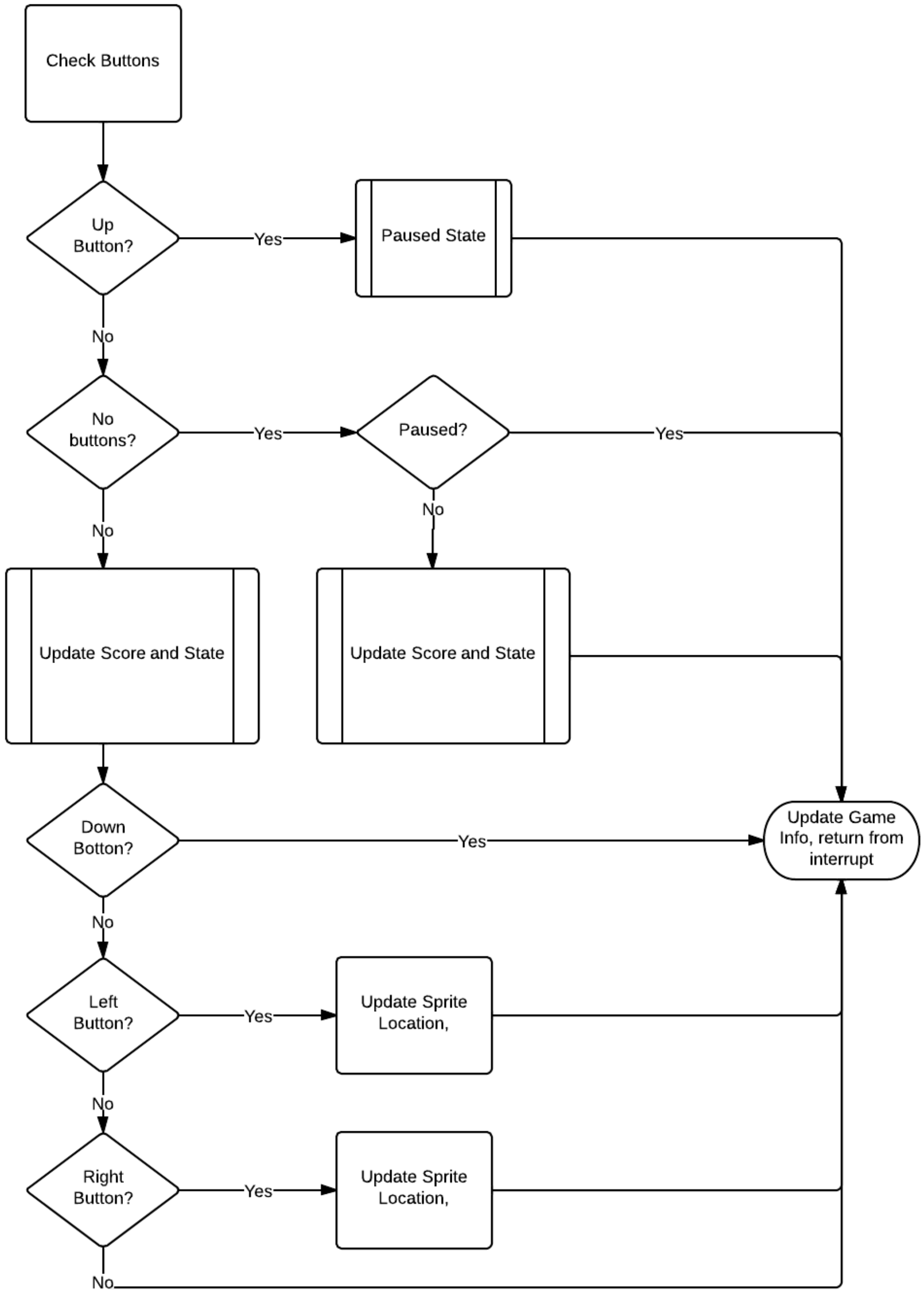



*Figure 4: Flowchart of Picoblaze Logic*

### 3.2 Colorizer

### 3.3 Collision Detection

### 3.4 Icon

## 4 Conclusion

Length of time, github, results, etc.

### 4.1 Challenges

- **ROM for map that dynamically changes**. We attempted to add memory in order to change the collision detection and dynamically switch wall coordinates, but this was not able to be implemented.
- **Integration between hard-coded images and script generated images**. We had trouble getting the different icons and splash screens created with Perl script to work with our current icon implementation. Ultimately, these graphics are only used in the report and final presentation.

### 4.2 Time Invested

| | Erik Rhodes | Bhavana Dhulipala | Rohan Deshpande | Nikhil Patil |
|---|---|---|---|---|
| Game Logic | ✓ | | | |
| Video Controller | ✓ | ✓ | ✓ | |
| Icons and Walls | | ✓ | | |
| Graphics | | ✓ | ✓ | ✓ |
| Random Number Generation | | | ✓ | |
| Documentation and Source Control | ✓ | | | |

*Table 1: Division of Tasks*

### 4.3 Future Work

- **ROM Map:** The current implementation of wall generation does not use any memory. If this functionality was converted to be stored in a ROM, different wall widths could be displayed in one frame, and collision detection could be done based off of the coordinates themselves.
- **Improved Graphics:** Creating more icons and backgrounds would improve the game's aesthetic appeal.
- **Powerups and Bonuses:** Certain objects could give the player an extra life or a slower game speed.
- **Multiplayer Mode:** Two different tunnels could be generated, allowing multiple players to compete against each other.