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 vehicle against the walls or obstacles as it travels down the tunnel. 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 the vehicle by using the left and right push buttons 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 push button pauses it. Different icons and speeds can be selected by toggling the switches on the board.

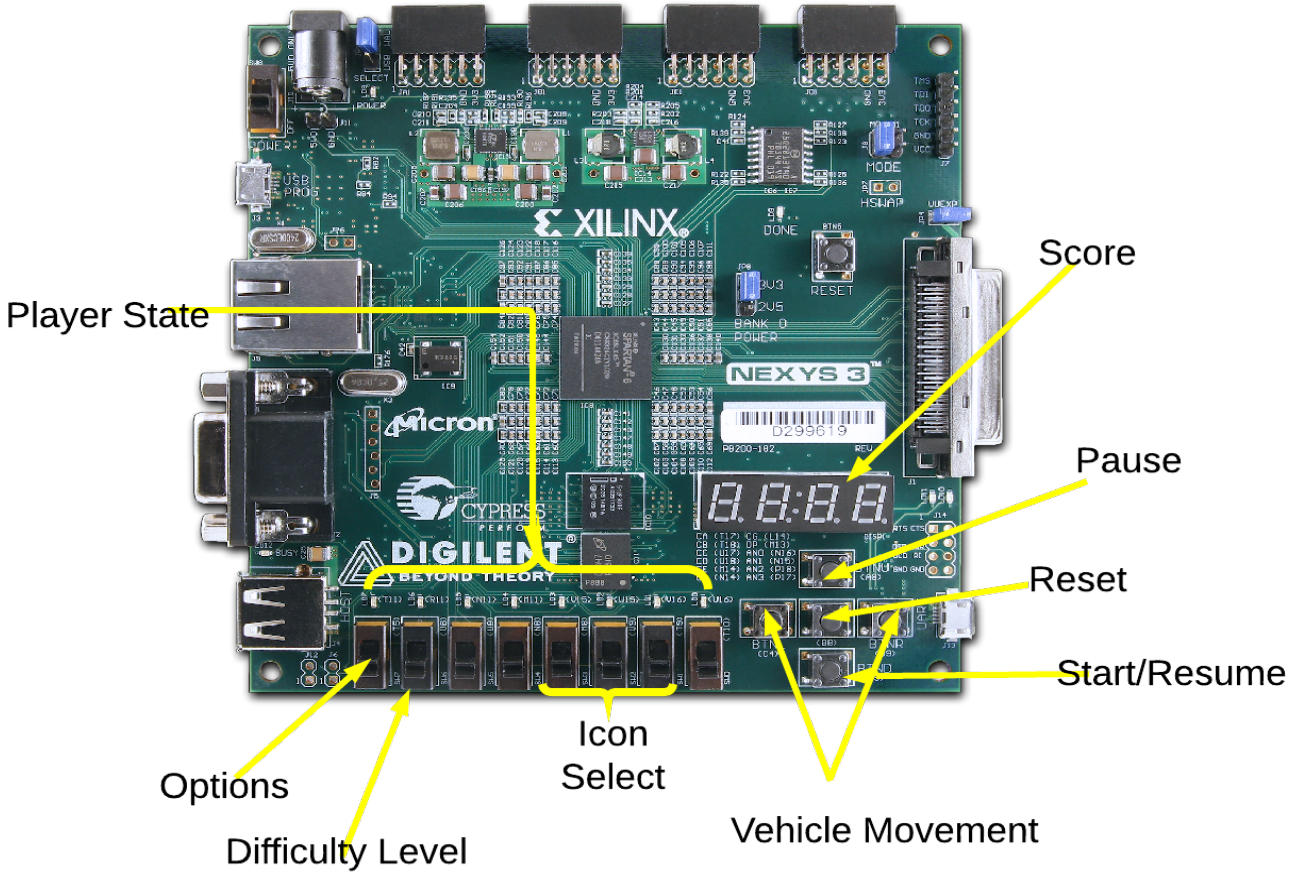


Figure 1: Player Controls

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

2 Implementation

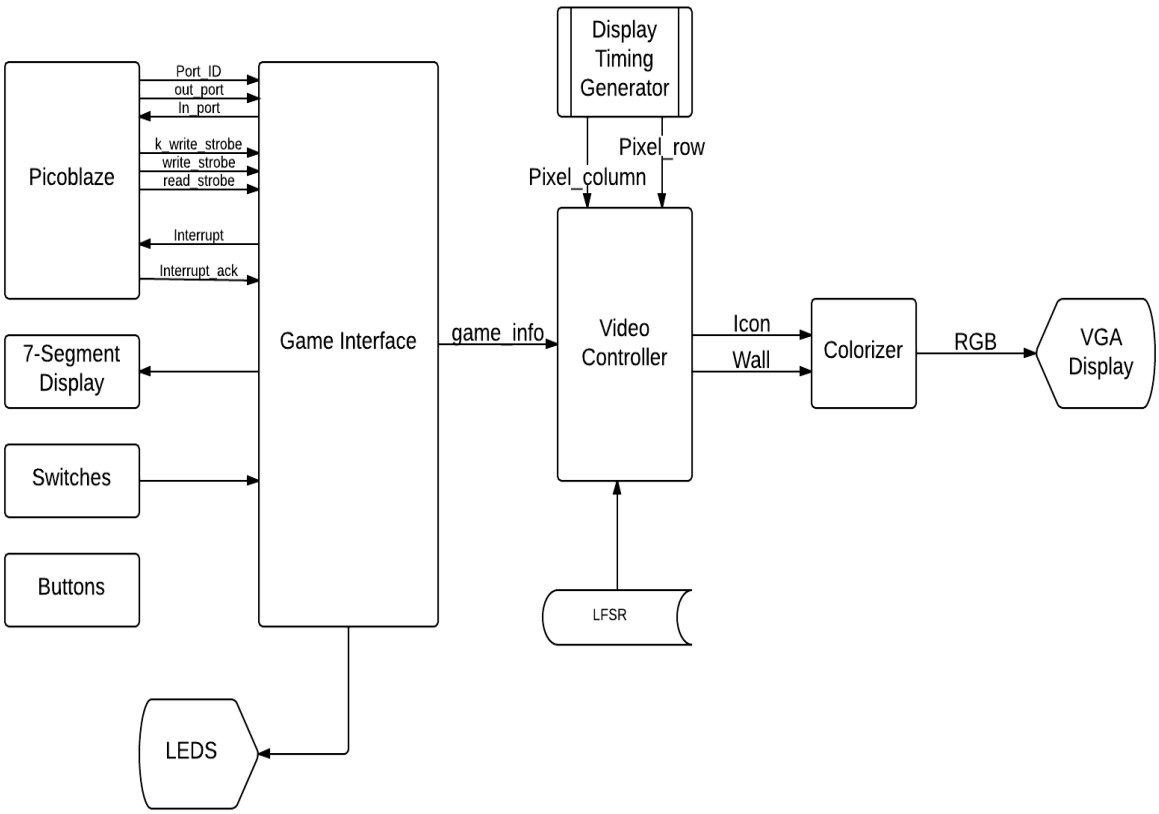


Figure 2: Game Play Block Diagram

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 receives 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 order seen in Figure 4. Each button check performs approximately the same function. The push button

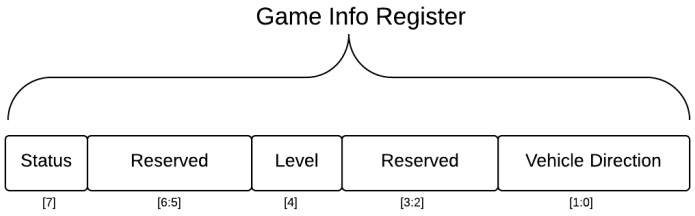


Figure 3: Allocation of bits in game_info register

input is checked, all other bits not pertaining to the specific button are masked off, and the 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

```
1  FETCH    s2,   SP_BTN              ;load saved debounced button
2  AND      s2,   MSK_BTN_UP          ;mask with up button
3  COMPARE  s2,   MSK_BTN_UP
4  JUMP     NZ,   chk_no_btn          ;go to next phase if no up button
5  LOAD     s3,   PAUSED_STATE        ;else pause and send out that register
6  STORE    s3,   STATE
7  FETCH    s3,   SP_GAME_INFO
8  LOAD     s4,   GAME_PAUSED         ;update game_info paused
9  AND      s3,   s4
10 STORE    s3,   SP_GAME_INFO
11 LOAD     s5,   FF                  ;output LED pattern
12 OUTPUT   s5,   PA_LEDS
13 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.

Listing 2: Update the state and game info

```
1  CALL          update_score
2  LOAD   s3,    ACTIVE_STATE
3  STORE  s3,    STATE
4  FETCH  s3,    SP_GAME_INFO
5  LOAD   s4,    GAME_ACTIVE
6  OR     s3,    s4
7  STORE  s3,    SP_GAME_INFO
8  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.

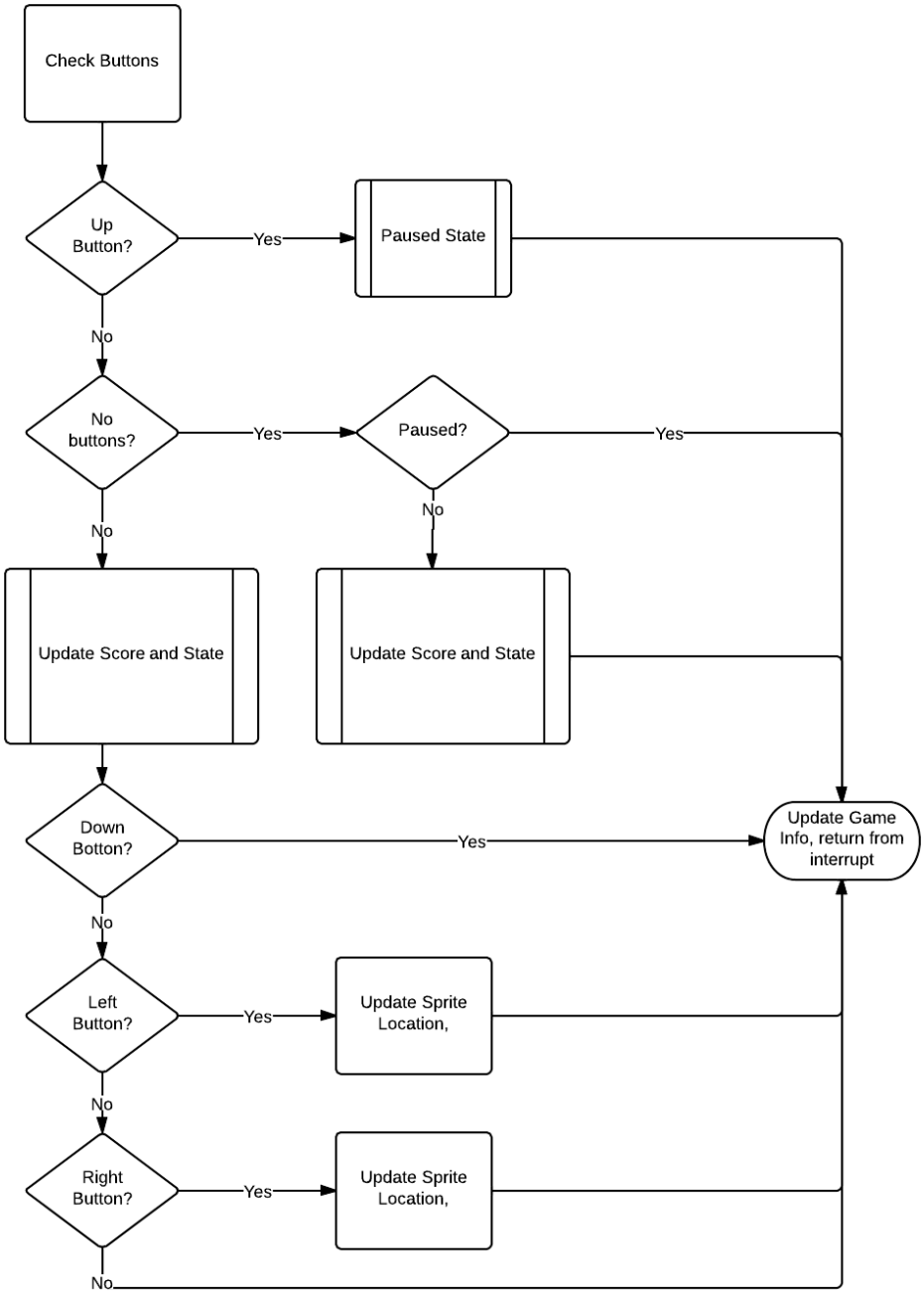


Figure 4: Flowchart of Picoblaze Logic

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, implemented in hardware, managed the icon switching and generation, the width and location of the walls, the obstacle creation, and the collision detection.

3.1 Walls

The tunnel walls consist two red lines separated by a predefined width. The walls directions are generated randomly using a linear feedback shift register. The video controller takes 2 bits from the LFSR and moves the walls left, right, or keeps them at the same position. Since there are 4 possible numbers, the last possibility defaulted to keeping the walls at the same position.

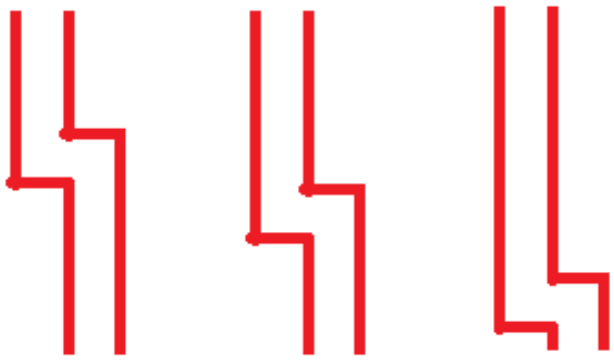


Figure 5: Wall Location Transitions

The tunnel width, which is a function of time, decreases continuously as the user is playing the game. It is controlled by a flag that is set whenever the counter reaches a specific number. The wall speed is also monitored by a counter, which decides when to alter the wall location. Figure 5 shows the transitions the wall would make based off one random value.

Edge detection was performed in order to ensure the walls did not move out of bounds. The coordinates of each side of the wall are checked, and if they are on the edge, the wall simply changes its next location to be closer to the middle.

3.2 Icons

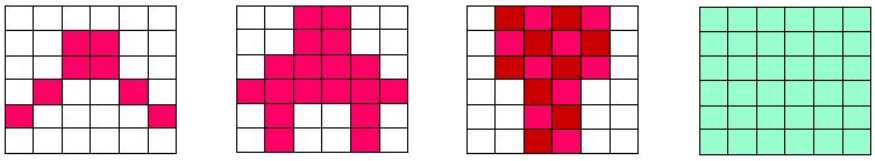


Figure 6: Icons the user can select

There are currently four icons which can be selected by different switches on the board. The icons, just like the obstacles, background images, and walls, were created by directly coding the images into bits (Listing 3) instead of reading the image from a `coe` file. When the `pixel_row` and `pixel_column` signals reach the location of the icon, they are displayed on the screen. At all other times the icon acts transparent. The icons are "animated" by interchanging different colored icons of the same shape. The hammer in Figure 6 is an example of this.

The vehicle's location is updated based on the `game_control` register coming from the Picoblaze soft core. Whenever the display timing generator has finished displaying a full screen and its coordinates are back at the origin, a counter is checked to decide whether the vehicle's location should change, regardless of the push button inputs. Only when this counter is zero will the location be updated. This effectively allows us to control the "sensitivity" of the vehicle by modifying the counter value.

Listing 3: Example Icon creation (hammer)

```
1      if(i == 1 && (j==0 || j==1 || j==2))
2        bitmap_bot_4[i][j] = 2'b11;
3      else if(i == 2 )
4        bitmap_bot_4[i][j] = 2'b11;
5      else if(i ==3)
6        bitmap_bot_4[i][j] = 2'b11;
7      else if(i == 4 && (j==0 || j==1 || j==2))
8        bitmap_bot_4[i][j] = 2'b11;
9      else
10       bitmap_bot_4[i][j] = 2'b00;
```

3.3 Collision Detection

Collisions are detected by comparing the coordinates of the wall or obstacle with the coordinates of the icon. If they are equivalent, the collision detect is asserted, which is checked every clock edge. Once this occurs, the game stops and a "Game Over" screen is displayed.

Listing 4: Collision Detection Logic

```
1 if ((Pixel_row >= locY) && (Pixel_row <= (locY + 3'h6)) && (Pixel_column >= locX)
     && (Pixel_column <= (locX + 4'h6)) ) begin
2   collison_detect <= 1'd1;
3 end
```

3.4 Obstacles

Obstacles randomly appear on either side in the tunnel and also employ collision detection. They take the icon's size (6x6) and extend the height to transform into a rectangular shape. The direction of the wall is also based on the LFSR, and is constrained to either be on the left or right side of the tunnel.

3.5 Colorizer

The `colorizer` module displays the icon and tunnel walls pixel by pixel. Table 1 summarizes the color scheme used for wall and icon.

4 Conclusion

4.1 Challenges

- **Debugging**. Finding errors in our code proved to be difficult due to the length of time synthesis takes. We were forced to view the output on the screen to resolved issues, which was not very practical. Debugging assembly language was also quite tedious, as there aren't any helpful warning or error messages.

Wall	Icon	Color	Purpose
00	00	White	Background
01	00	Green	Background (Trees)
10	00	Red	Obstacles & Walls
11	00	Gray	Background
X	01	Maroon	Icons
X	10	Cyan	Icons & Splash Screens
X	11	Magenta	Icons & Splash Screens

Table 1: Color Schemes

- **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.

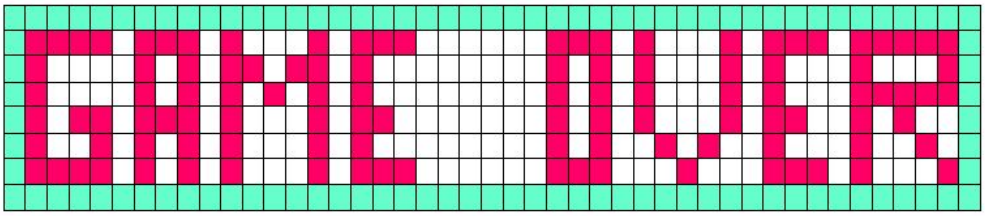
	Erik	Bhavana	Rohan	Nikhil
Game Logic	✓			
Video Controller	✓	✓	✓	
Icons and Walls		✓		
Graphics		✓	✓	✓
Random Number Generation			✓	
Documentation, Source Control	✓			

Table 2: Division of Tasks

4.2 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.

Creating **Tunnel Vision** was an enjoyable experience (for the most part). While we did face some roadblocks, we were pleased with the overall result. There are many different paths we can take in the future to add new features to this game.


GAME OVER