# ENGS 31 15X Final Project Report

# Tron Game

**Jonathan Huang & Edrei Chua**

*26 August 2015*

**Abstract**

The goal of our project is to create the popular video game, Tron, using theories of digital design. Tron is a two player game based on the Walt Disney Productions Motion picture *Tron* released in 1982. Each player controls a snake, and the goal of the game is to avoid colliding into the player's snake, the opponent's snake or the boundary. Our design consists of two 4-button Pmod external board used for directional controls, a Nexys 2 FPGA board running on Spartan 3E (4fg320), and a VGA monitor. There are three major steps in creating the game: a) programming the VGA controller; b) devising a way to store the pixels information in a frame buffer using a block RAM; c) implementing the logic for the game controller via a state machine. We have successfully accomplished each of the above three tasks and have presented the game to our classmates on 24 August 2015. Our design utilizes 6 counters, 95 registers, 4 FSMs, 1 dual port block RAM, 1 ROM, 4 accumulators, 19 comparators and 9 XORs. The critical timing path for our design is 11.886 ns. We applied what we have learnt over the past few weeks into creating the game and this helped to solidify our understanding of the digital design process.

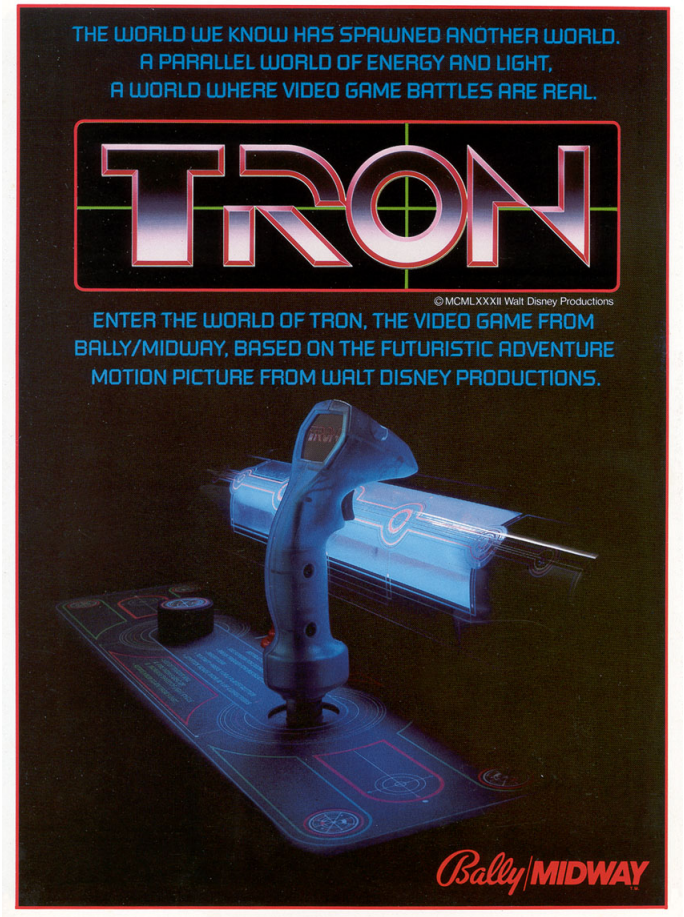


Fig. 1: This is the cover for the video game Tron manufactured and distributed by Bally Midway in 1982

# Contents

# 1 Introduction

Tron is a two player game based on the Walt Disney Productions Motion picture *Tron* released in 1982. Each player controls a snake, and the goal of the game is to avoid colliding into your own snake, the opponent's snake, or the boundary. The goal of this project is to design the game using VHDL. There are 3 main challenges in coming up with the design:

1) **Creating the VGA controller**: this requires creating two counters for horizontal and vertical sync. The main challenge is in counting to the exact number of clock cycles needed. We tested the VGA controller using a test pattern.

2) **Dealing with memory storage in a frame buffer**: this requires creating a block RAM that facilitates the reading and writing of pixel information from the game controller to the VGA. Due to memory limitation of the Nexys 2 board that we are working with, it is also necessary to scale down the number of pixels from 640 x 480 to 320 x 240. Another challenge is to devise a way to properly address a memory location using the xy co-ordinates of the pixel location.

3) **Design of the game controller**: the challenge is in co-ordinating the update of each player's trail to memory and to the VGA screen since it is only possible to update a single pixel to memory at a time. It is also necessary to figure out how to detect collision.

# 2 Design Solution

## 2.1 Specifications

Our design consists of a VGA display, a Nexys 2 FPGA board (which also contain the reset button for our game) and 2 x 4-pushbutton Pmod connected to the FPGA and used for directional control.

1) VGA Display
2) Main Control: reset button on Nexys 2 board
3) Player Control: left, right, up, down for each player (2x 4-pushbutton Pmod)
4) Nexys 2 FPGA board running on Spartan 3E

Fig. 2 below shows a conceptual block diagram for our design. The reset button, the game controller, the frame buffer and the VGA controller together forms the FPGA entity of the design. The FPGA is also connected to an external VGA display and 2 external Pmod components used for directional control.

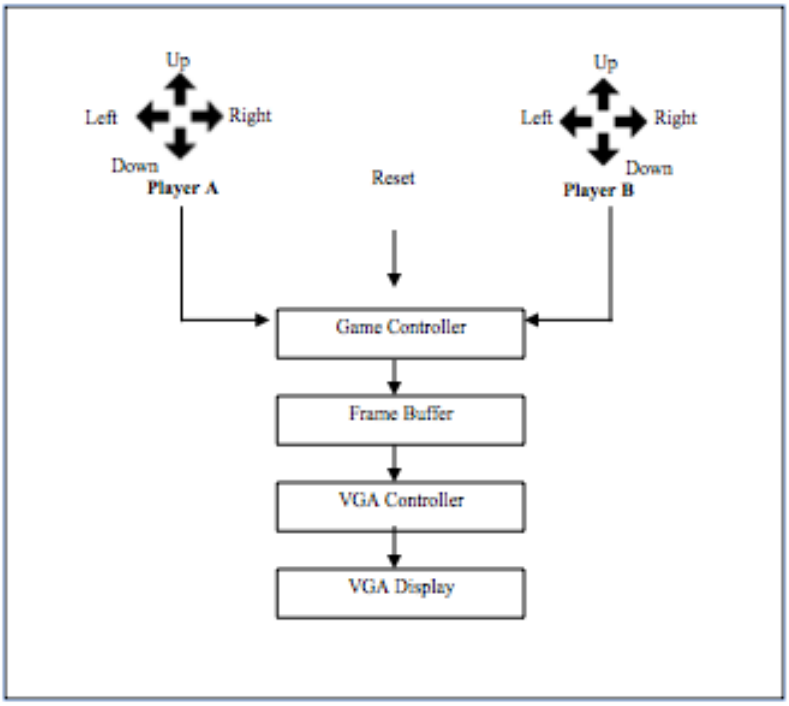


Fig. 2: This is the conceptual block diagram for our design

Fig. 3 shows a top level block diagram of our circuit, and the detailed functionality of each component will be elaborated in Section 2.3. It can be noted that each of the external buttons is connected to a debouncer, whose output is in turn connected to the Game Controller. The Game Controller also writes to and reads from the MemoryRAM. The MemoryRAM stores the pixel information and can be read by both the GameController and the VGAController. It also determine the output color that is sent to the VGA. Finally, the VGA controller controls the Hsync and Vsync and determines which pixel location is being written on the VGA monitor.

Fig. 4 shows a diagram of the control panel, with the location of the important buttons labeled. Each player holds a directional controller board that allows for up, down, left and right motion. There is a reset button on the FPGA. The FPGA is also connected to a VGA monitor via a VGA connector.

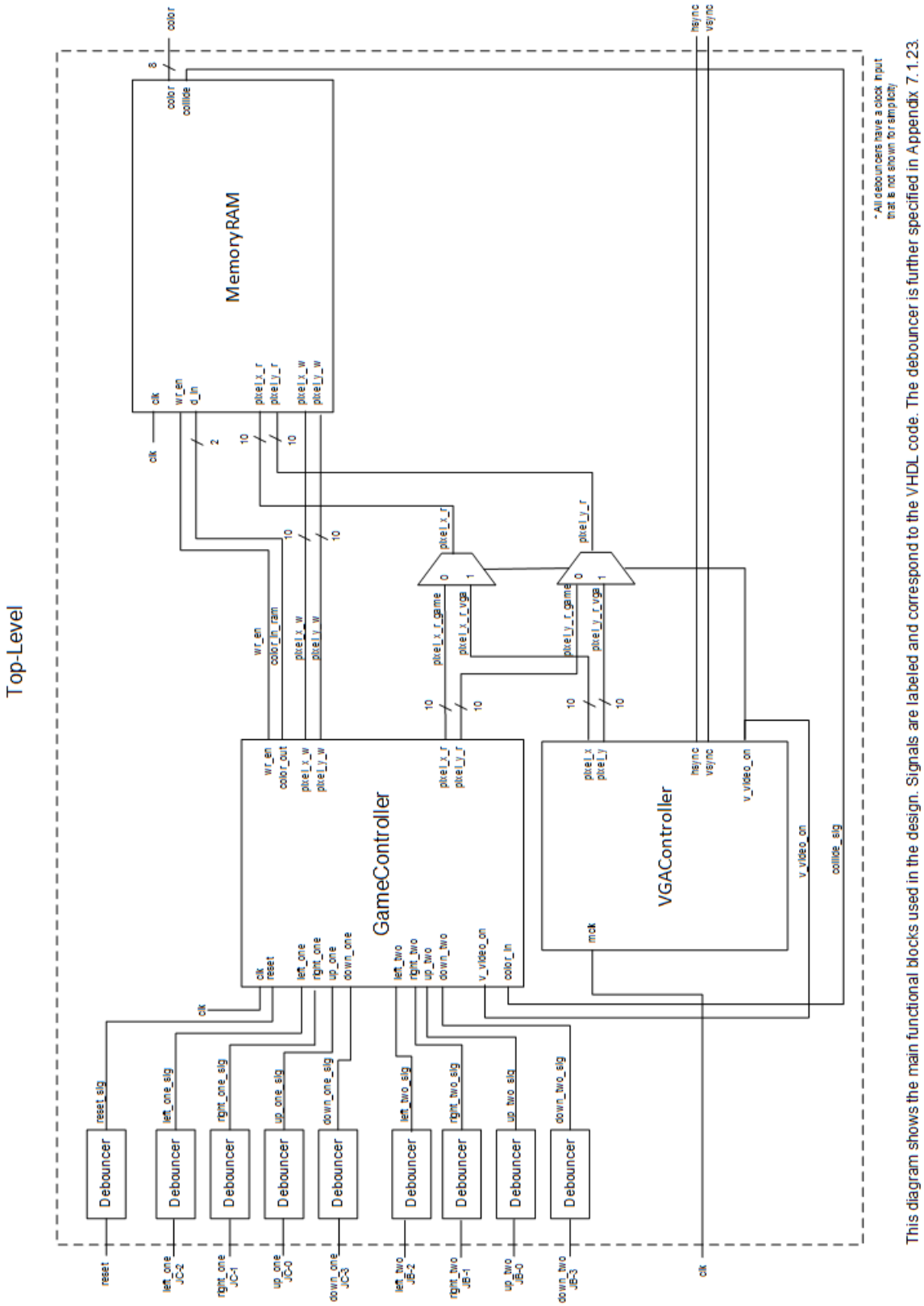


Fig. 3: Top level block diagram

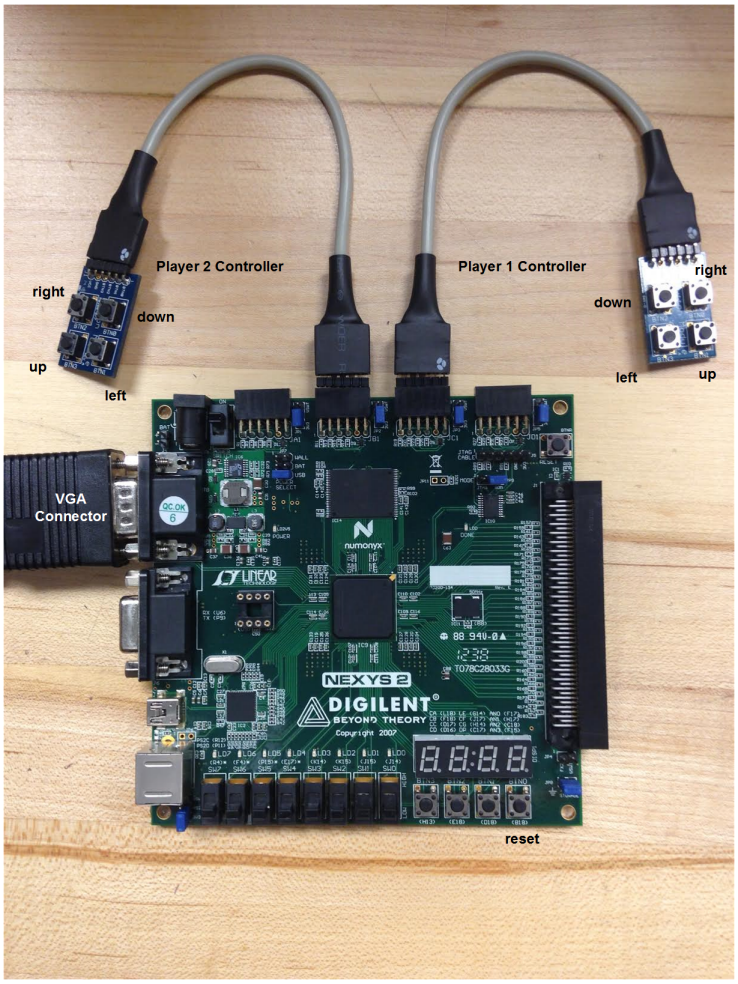


Fig. 4: Front panel

## 2.2 Operating Instructions

To set up the game, simply connect two 4-button pmod chip to J-B and J-C pin of the Nexys 2 board. Also, connect a VGA cable from the Nexys 2 board to the VGA monitor. The front panel in Fig 4. gives an overview of how to set up the circuit while the top-level diagram in Fig. 3 gives a more detailed illustration with the port number labelled. Connect the Nexys 2 board to the computer and upload the programming bit file to the FPGA.

The user-interface of the game is also relatively straightforward:

1) To reset the game, press the reset button on the FPGA board

2) To play the game: Use the directional controls, which are buttons, to control your snake.

Both players will see their snakes moving in the VGA monitor and can used the buttons to control the direction of the snake. The same rules as the classic Tron game apply. The snake follows the direction of the head, which is controlled by the player, and leaves a trail behind. Player 1 wins when the head of Player 2s snake hits the body either snake or the boundary and vice versa. If a player hits a wall, that player loses. A tie happens if both players heads collide.

In our game, Player 1 is controlling a blue snake while Player 2 is controlling a green snake. In the event of a collision, the head of the losing Player will turn red to signify the end of the game. The players can press the reset button if they wish to play again. Fig. 5 below shows a demonstration of our game.

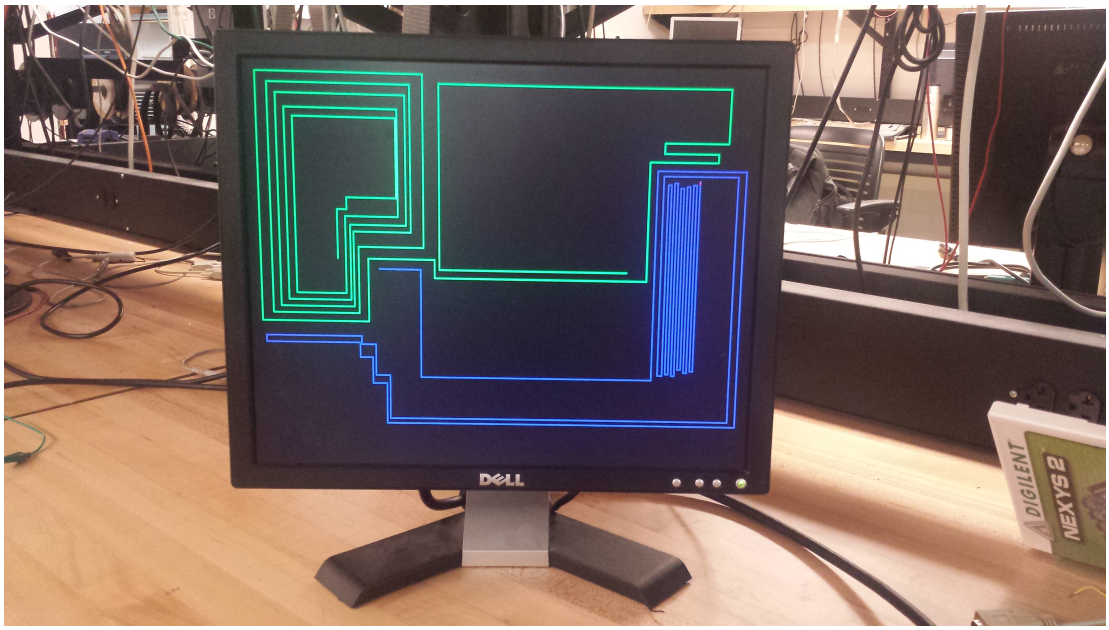


Fig. 5: A photo of the game

## 2.3 Theory of operation

**Top Level Design**

As can be seen from the top level block diagram (found in Appendix 7.1.1a), each of the buttons (up , down, left, right, reset etc.) are connected to a debouncer monopulser to ensure that external inputs are synchronized so to avoid metastability issues. These external controls (up_one, down_one, left_one, right_one, up_two, down_two are connected as inputs to the game controller. The game controller is connected to the

MemoryRAM, which stores the pixel information in memory. The VGA controller controls the Hsync and Vsync signal and decides the coordinate to write on to the VGA screen.

Of particular interest are the two multiplexers whose output pixel_x_r and pixel_x_r are connected to the MemoryRAM. The multiplexers determine the memory location to be read from the MemoryRAM to extract the color needed by the VGA monitor. When v_video_on = 1 i.e. the video is on, the read address will be from the VGA controller; the VGA monitor will update the screen. When v_video_on = 0 i.e. the video is off, the read address will be from the Game Controller; the game controller will read from memory to determine whether or not a collision has taken place.

**Game Controller**

The game controller part of the design consists of a a state machine and two directional state machine (one of each player). A high level state machine is presented in Appendix 7.2.1a. The detailed state machine of the game controller can be found in Appendix 7.2.1b while the state machines (one for each player) for direction control can be found in Appendix 7.2.1c and Appendix 7.2.1d. The datapath for the controller can be found in Appendix 7.1.2b.

The datapath of the game controller consists of 2 position counters that keep track of the x-y coordinate of each of the player. It allows the increment and the decrement of the x-y coordinate of each player's position based on the input up, down, left and right. The position counter also includes a pos_ld input, which loads the starting x and y position of each player at the start of the game. The game controller also contain a background counter thats counts through all the pixels, and is used during a reset. A slow down counter is used to count to an integer multiple of the period of v_video_on and serves as the enable for transitions in the state machine. This is necessary in order to slow down the speed of the snake to match approximately that of a typical human reaction time.

Apart from the 3 counters, the game controller also uses several multiplexers to communicate information with the memory block. A Write Mux takes player_select as its select bit and is used to determine which player's information should be updated to memory. A Color Mux takes reset_game, collision parameters (collison_one and collision_two) and player_select as select bits and is used to determine the color to write to memory (RED for collision = 1, BACKGROUND for reset_game = 1 and PLAYER_2_COLOR for player_select = 1 and PLAYER_1_COLOR otherwise). Combinational logic which involves two 2-input OR gate is used for collision detection. A collision for a player happens when either the pixel to be written to

memory is not a background pixel, or is a pixel that is along the border.

**Memory Block**

The memory block (functional block can be found in Appendix 7.1.2c) takes as input the pixel co-ordinates (pixel_x_w and pixel_y_w) and the color (color_out) from the game controller. It converts the pixel coordinate into a read address that can be referenced by the block RAM. The read address is a simple concaternation of pixel_x_w and pixel_y_w. The least significant bit of pixel_x_w and pixel_y_w is dropped in order to scale down the number of pixels stored in memory from 640 x 480 to 320 x 240 due to memory limitation of the Nexys 2 board. This is essentially the same as grouping four adjacent square pixels as a single square pixel.

In addition to writing to memory, the memory block also allows the game controller and the VGA controller to read from its memory through the input pixel_x_r and pixel_y_r. This is required to write the pixels to the VGA screen and to provide a feedback loop to the game controller to determine collision. Since the block RAM cannot read from and write to memory at the same time, a wr_en signal from the game controller's state machine is used to determine which operation to perform at any one time.

**VGA Controller**

The VGA controller contains two counters - the horizontal counter and the vertical counter as can be seen in Appendix 7.1.2d. The horizontal counter will enable the vertical counter when it counts to the last column. Together, the horizontal counter and the vertical counter will assert v_sync and h_sync high or low depending on the vertical count and horizontal count.

**Debouncer Monopulser**

The Debouncer Monopulser consists of a typical debouncer (prof Hansen's code) connected to a monopulser, which is a state machine that ensures that the electrical input only goes high for one clock cycle even when the button is pressed for several clock cycle. The drawing for the debouncer can be found in Appendix 7.1.2e while the state diagram for the monopulser.

## 2.4 Construction and Debugging

We started by building the VGA controller and test to see if it works by using a test pattern. At the beginning, the VGA controller did not return a display at all. This was because the clock frequency was not set to 25 MHz. Changing the clock frequency to 25 MHz using a clock buffer solves the problem. Another

problem that we later faced was that the h_sync and v_sync signal were off by a few clock cycles. We compared the values of the h_sync and v_sync with the values in the lecture notes again and corrected the mistake.

After we ascertained that the VGA controller worked using the test pattern (Appendix 7.3c), we moved on to constructing the MemoryRAM. We spoke to Professor Hansen and he advised us to perform a simple concaternation of the x coordinate and the y coordinate to generate the memory address. We also dropped the least significant bit of the x co-ordinate and y co-ordinate it order to scale down the number of pixels from 640 x 480 to 320 x 240. We tested the MemoryRAM by attempting to write to memory and read from memory to the VGA monitor. The test code for the memory test can be found in Appendix 7.3d. We attempted to draw a few vertical and horizontal lines on the VGA just to make sure that the MemoryRAM is working. We realised that while we are able to draw the vertical lines, the horizontal lines drawn would flicker a lot. Eventually, we realised that it was because we are not addressing the memory properly when dropping the least significant bit and are grouping the pixels the wrong way. The horizontal pixels appear larger than the vertical pixel, and this offer us the clue that we are addressing the block of pixels incorrectly. In addition, we realised that we were attempting to update the memory when v_video_on = 1, which explains the flickering.

Appendix 7.3 a and 7.3 b shows the successful test pattern generated from the VGA controller test and the Memory test respectively.

The final step would be to build the Game Controller. Our initial state machine is different from the one presented in this report because originally, we thought of have a separate state machine for each player and to coordinate the update to memory by toggling between the two different state machine. The logic gets complicated really quickly, and we thought of building a single state machine with 16 blocks of states, each block representing one of the 16 combination of direction ($2^4 = 16$ since there are two players and each player gets to choose between four directions up, down, left and right).

After talking to Professor Hansen, we decided to have two position counters to keep track of the x and y co-ordinate of each player. A game control state machine will select which player's pixel to be updated in memory. Two secondary state machines are also created to control the update of their respective position counters. Collision happens when a player attempts to write to memory a pixel that is either not a background pixel or is a pixel along the boundary. The high level state machine for our new design after consulting Professor Hansen is presented in Appendix 7.2.1a.

One of the first problems which we encountered was that a straight line was being drawn each time instead of a snake leaving a trail behind. We attempted to reduce the dimension of the boundary and observed that a shorter line was drawn. This tells us that our logic is sound and a line was indeed drawn; we concluded that the problem might be that the line was drawn too fast and we are not able to see it move at all. We created a counter that counts to a certain number of screen refreshes in order to slow down the updating of the position counter, and this allowed us to see the movement of the snake.

We also encountered the problem of not being able to update the trail of the player on the VGA. We noticed that one of the players did not move even though the player's pixel turns red after sometime, indicating a collision. We hypothesized that one of the player's pixel was not being written to memory even though the position counter for that player is incrementing. We narrowed the problem to the select bit player_select was always at the same value, choosing the first player over the second player. We fixed the problem by changing the output of one of the states to enable the update of both player's position.

Another problem that we faced later on was that the update of the trail for the snake was not a continuous line but instead, a dashed line. This happens because the update to memory happens every other pixel. We checked the parts of the code where we enabled the position counters and we noticed thatt we were enabling both counters using the same position enable even though we are only updating one to memory at a time. We created a separate enable for each position counter and that fixed the problem.

The final outstanding issue was that a blue or red line would flash at the bottom of the screen periodically. This happen even though we did not write anything to the bottom of the screen. Professor Hansen suggested that we might be writing to memory when v_video_on is 1. We looked through our code again and eventually realised that we did not set the range at which v_video_on is 0 at the right time in our VGA controller. With that, we have resolved the final bug in our code.

# 3 Justification and Evaluation of Design

Our design utilizes 6 counters, 95 registers, 4 FSMs, 1 dual port block RAM, 1 ROM, 4 accumulators, 19 comparators and 9 XORs in the FPGA. The detailed resource utilization chart is in Appendix 7.2.3.

In addition, the critical time delay for our design is 11.886 ns. It goes through seven levels of logic and

happens in the path starting from reading the Memory RAM to determine collision for player 2 and ends with the updating of the MemoryRAM with the color of either player 2 or the collision color. The detailed chart can be found in Appendix 7.2.4.

We also noted that there are seven residual warnings left over from synthesis, and we noted that they all harmless and can be ignored. The first 6 warnings are concerning the fact that the least significant bit of the vector bus for pixel_x_r and pixel_y_r is not used and the least significant and most significant bit of the vector bus for pixel_y_r and pixel_y_w are not used. This is because of the way we rescale the pixel size (grouping 4 pixels as one) due to the memory limitation on the Nexys 2 board. The final warning pertains to h_video_on not being used in the design. We only write to memory when v_video_on = 0 and never found the need to write to memory in the short duration when h_video_on = 0. However, we noted that it is a convention for the VGA controller to have a h_video_on output as it allows for the portability of the VGA controller code for future projects that might require updating during the time when h_video_on = 0. The list of residual warnings can be found in Appendix 7.2.5.

Our final design is better than our original approaches, which includes 1) creating two different state machine for each player and instatiating it in the top file; 2) having 16 blocks of states to represent the 16 different direction combination of both players. We managed to program the logic of the game using a state machine with fewer states than our original two approaches.

Our design also allows for easy modifications, such as adding extra features. For example, it would be relatively easy for someone to include an acceleration feature to our design by creating an input that would determine the number of V_video_on periods to be counted for the slow down counter.

# 4 Conclusions

Our original proposal for the Tron project involves additional features such as an acceleration mode and a scheme to record the score of each player. Due to time constraint, we were unable to work on the additional features.

Nevertheless, we successfully accomplished the task of desigining and building Tron within the time frame given and we presented the game to our classmates on 24 August 2015. Our design utilizes 6 counters, 95 registers, 4 FSMs, 1 dual port block RAM, 1 ROM, 4 accumulators, 19 comparators and 9 XORs. The

critical timing path for our design is 11.886 ns. This project has helped to solidify our understanding of the digital design process.

We advised groups that are considering a similar project to modularise the project into three major component: the VGA controller, the memory RAM and the game controller, and to build the project in that sequence. In addition, it is necessary to test each module and ensure that it works before proceeding to building the next module.

We advised all future groups to make sure that they have detailed state machines and datapaths for the module they are building before they even started coding. Digital design is different from software programming; it is essential to have a clear understanding of the combination and arrangement of macro units (FSMs, registers, counters, adders, comparators, RAM etc) to ensure that the design can be synthesized. It is frustrating to debug a code when you don't have a clear picture of the state machine and datapath involved.

# 5 Acknowledgments

We would like to acknowledge Professor Eric Hansen and the lab instructor Dave for providing valuable feedback during the design review. We are also grateful to Professor Hansen, Dave and the TAs for offering insightful opinions during the debugging process.

The task of designing, programming, debugging and writing the report is evenly divided between the both of us since we believed that learning is maximised when we both engage in all aspects of the project.

# 6 References

Wikipedia Page on Tron Game. Accessed on 26 August 2015. Retrieved from:
`https://en.wikipedia.org/wiki/Tron_%28video_game%29`

# 7 Appendices

## 7.1 System Level Diagrams

### 7.1.1 Front panel

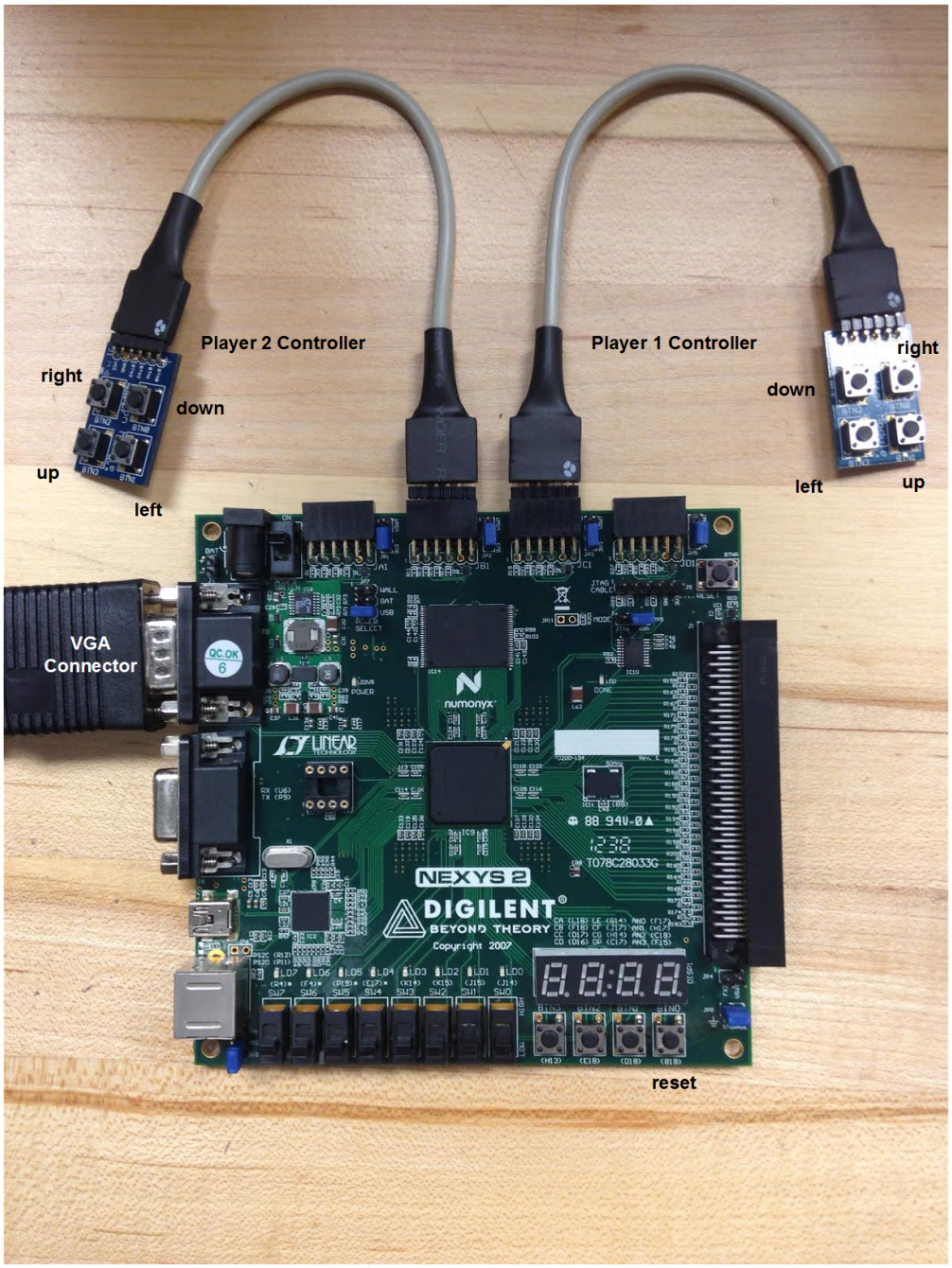


Fig. 6: Front panel

### 7.1.2 Functional Block Diagram

**a) Top level block diagram**

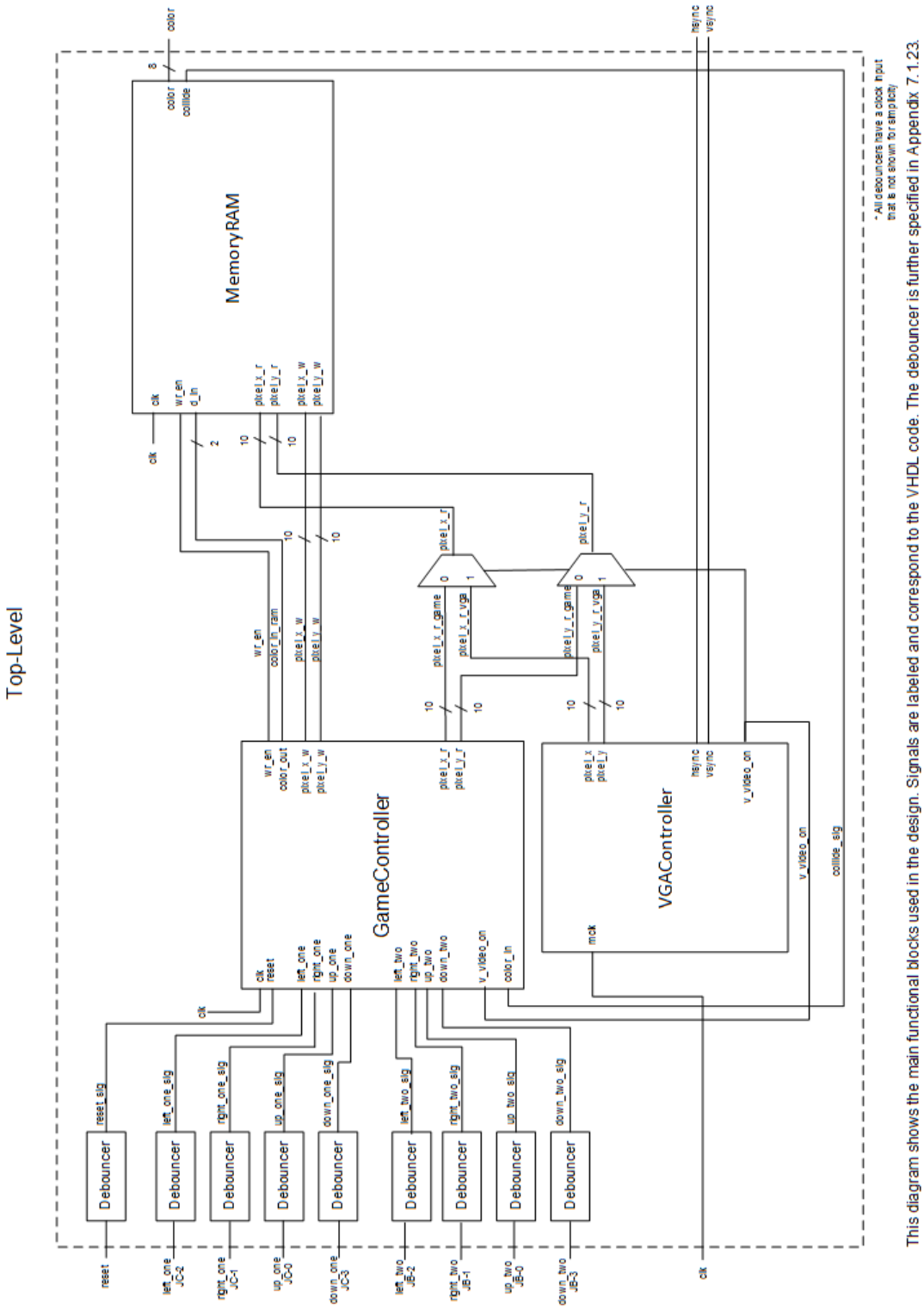


Fig. 7: Top level block diagram

**b) Game Controller block diagram**

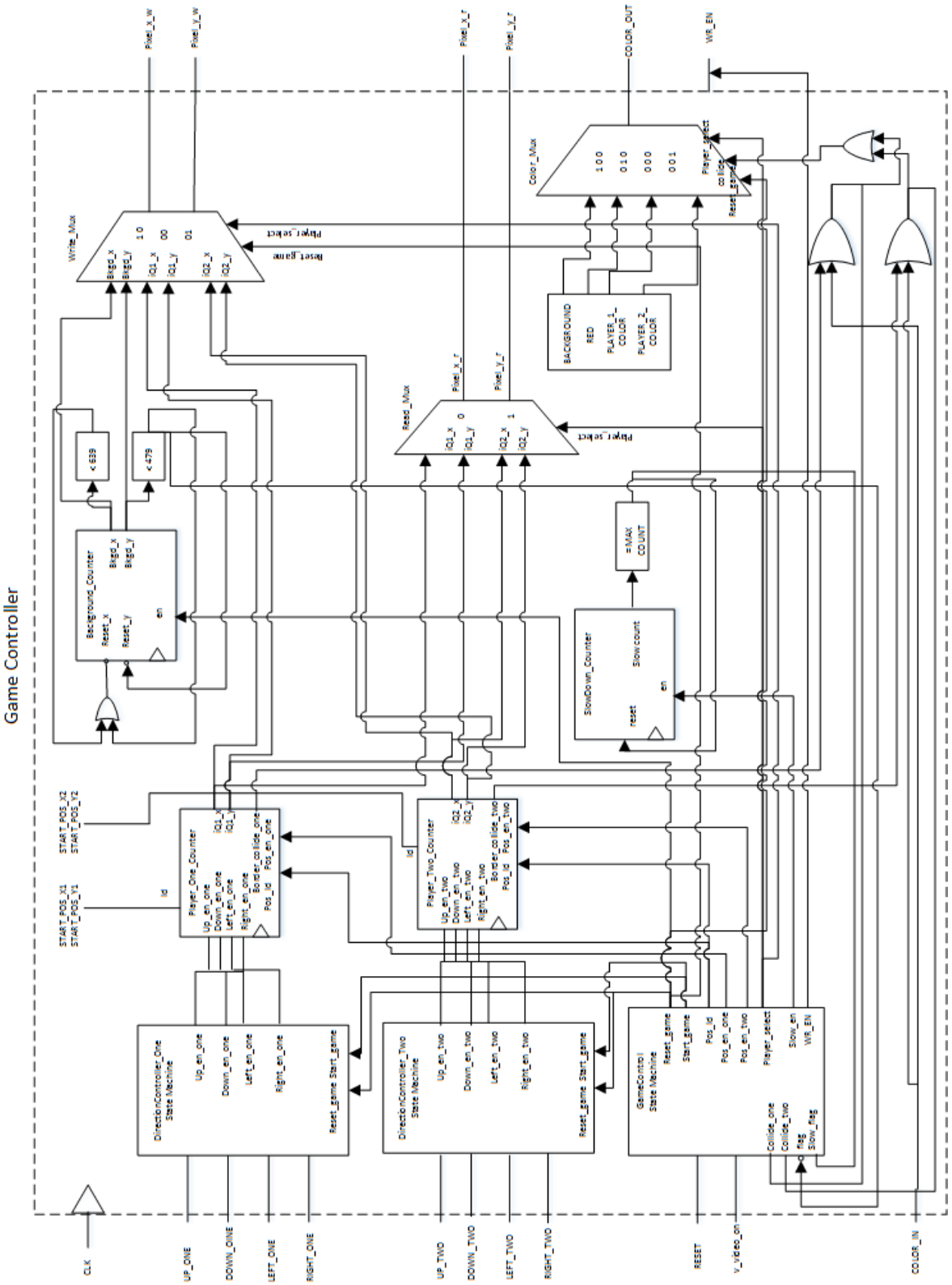


Fig. 8: Game Controller block diagram

**c) Memory block diagram**

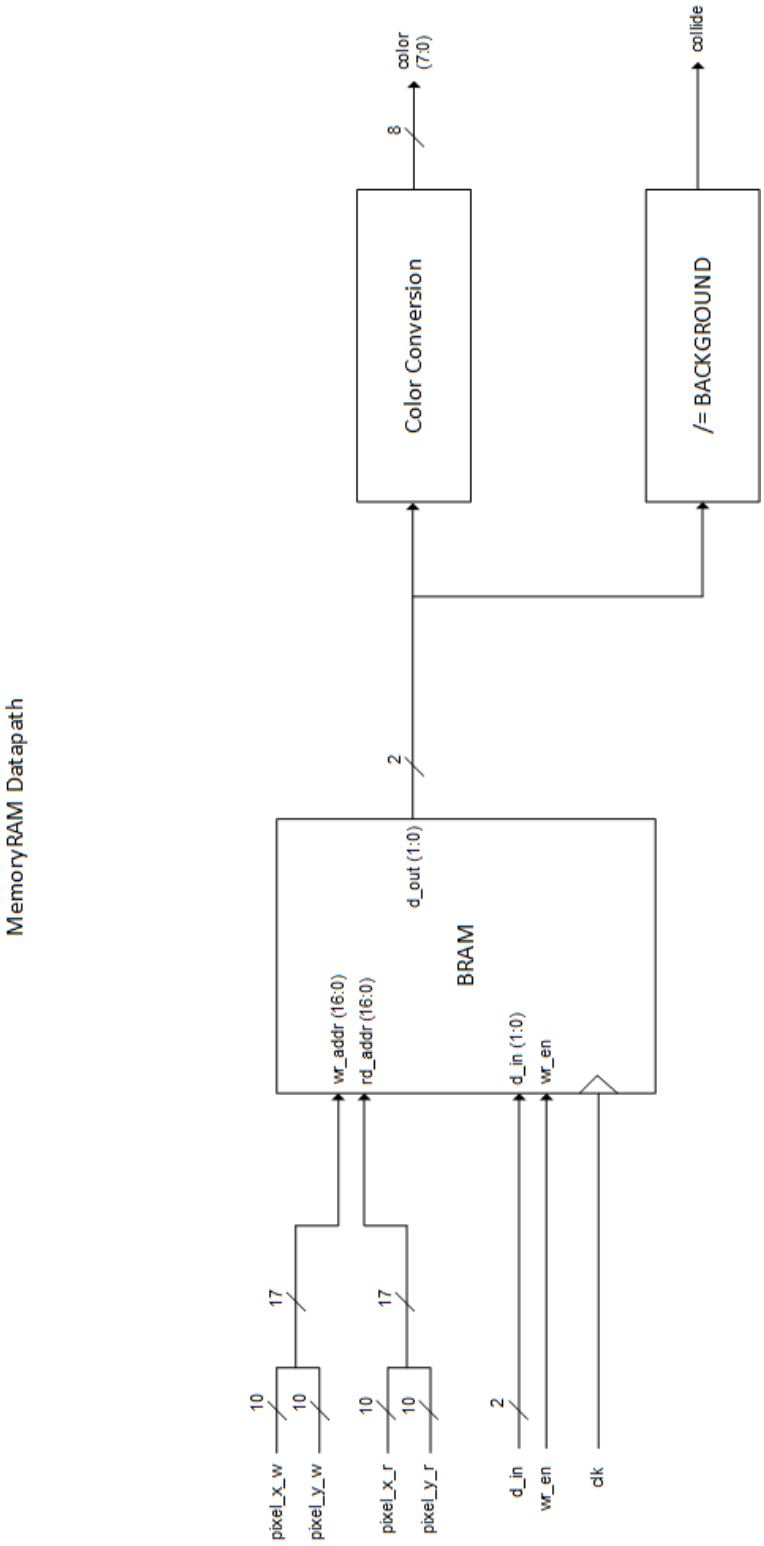


Fig. 9: Memory block diagram

**d) VGA controller block diagram**

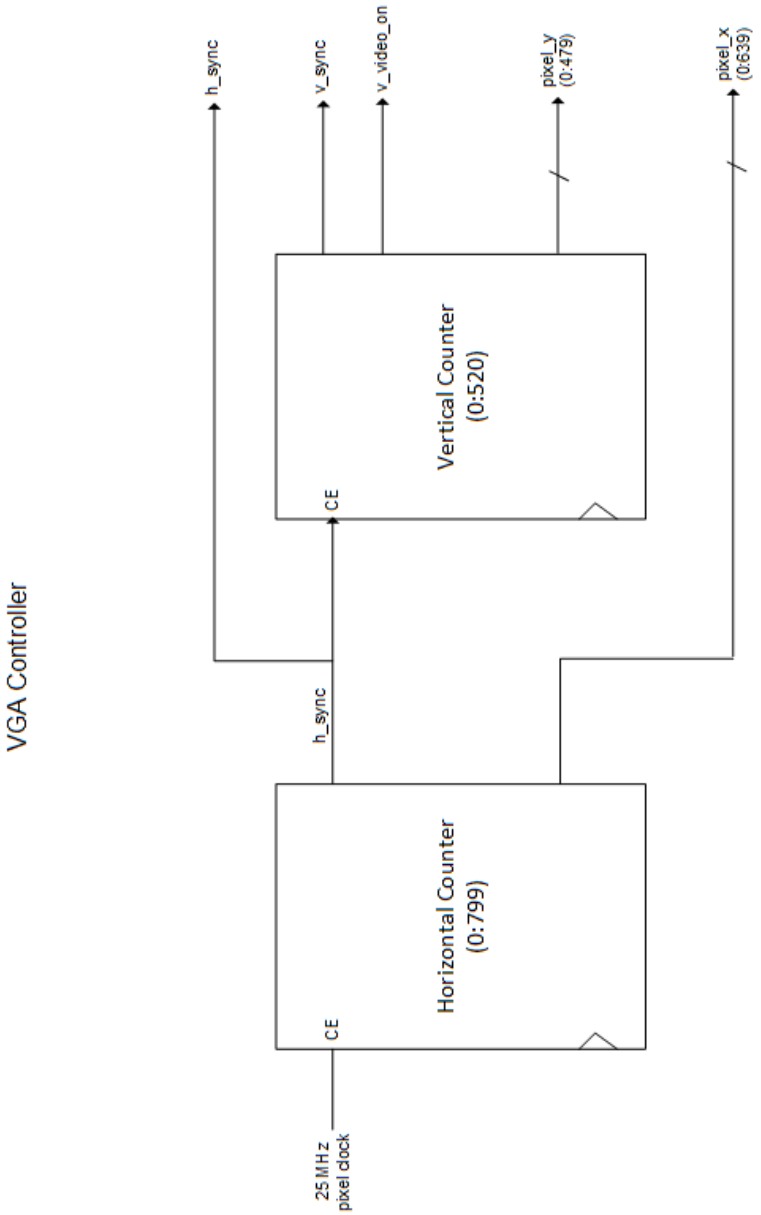


Fig. 10: VGA controller block diagram

**e) Debouncer Monopulser block diagram**

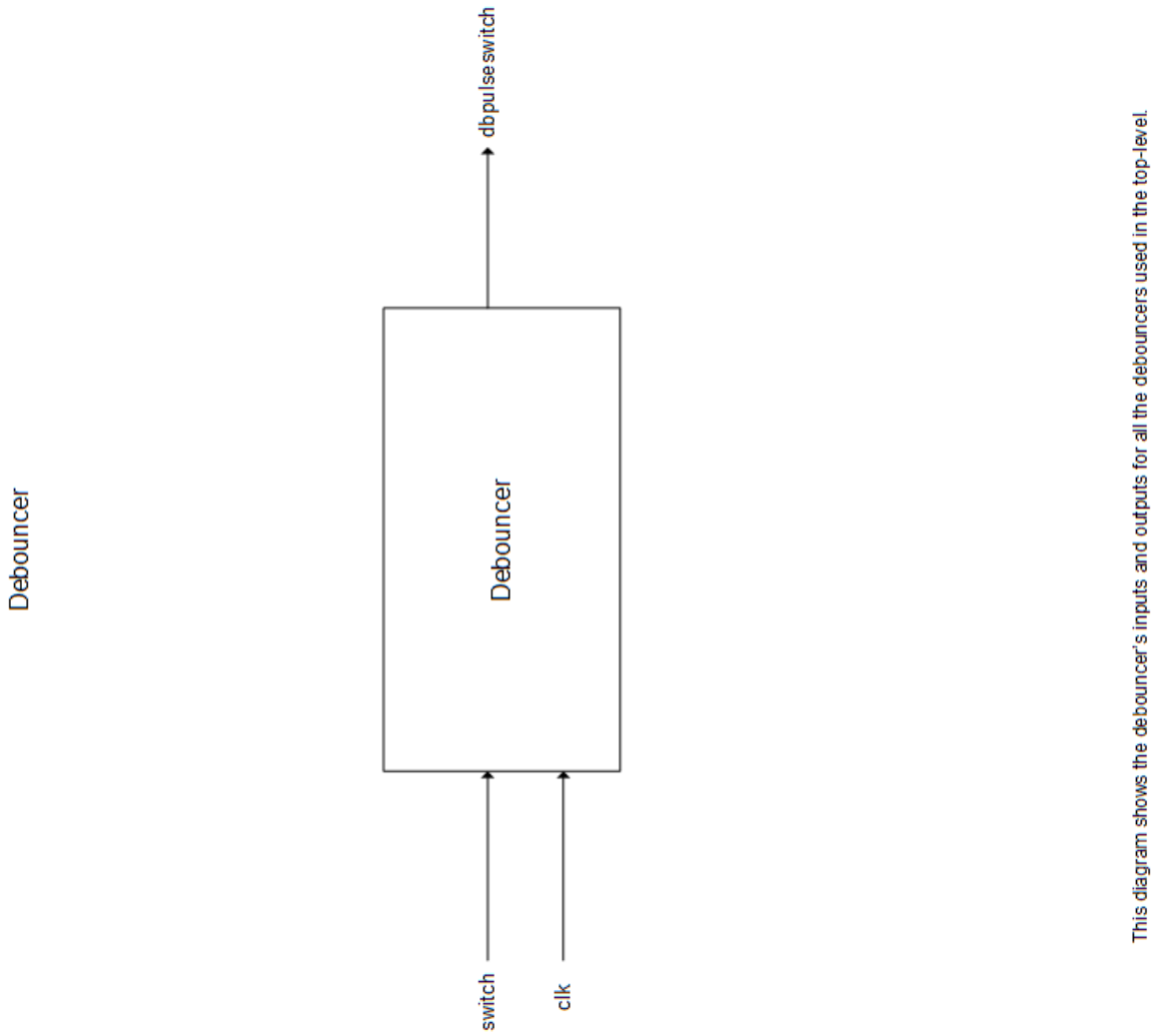


Fig. 11: Debouncer Monopulser block diagram

#### 7.1.3 Parts List

| Reference | Quantity | Part number | Description |
|---|---|---|---|
| Nexys2 | 1 | Nexys2 | Digilent Nexys2 board |
| BTN | 2 | Pmod-BTN | Digilent Pmod-BTN - Push buttons |

Table 1: Parts List

## 7.2 Programmed Logic

### 7.2.1 State Diagrams

#### a) High Level State Machine

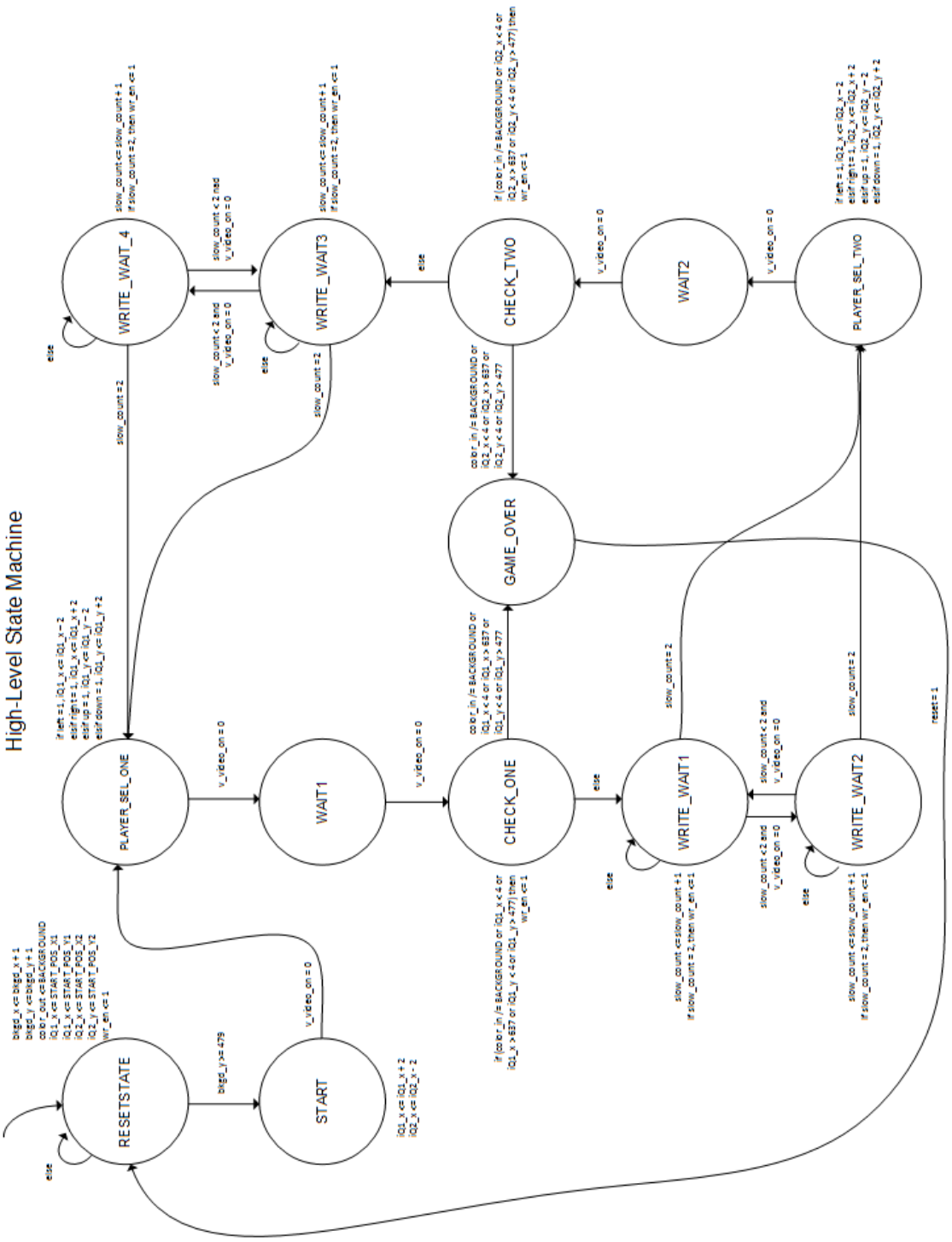


Fig. 12: High Level State Machine

**b) Game Controller FSM**

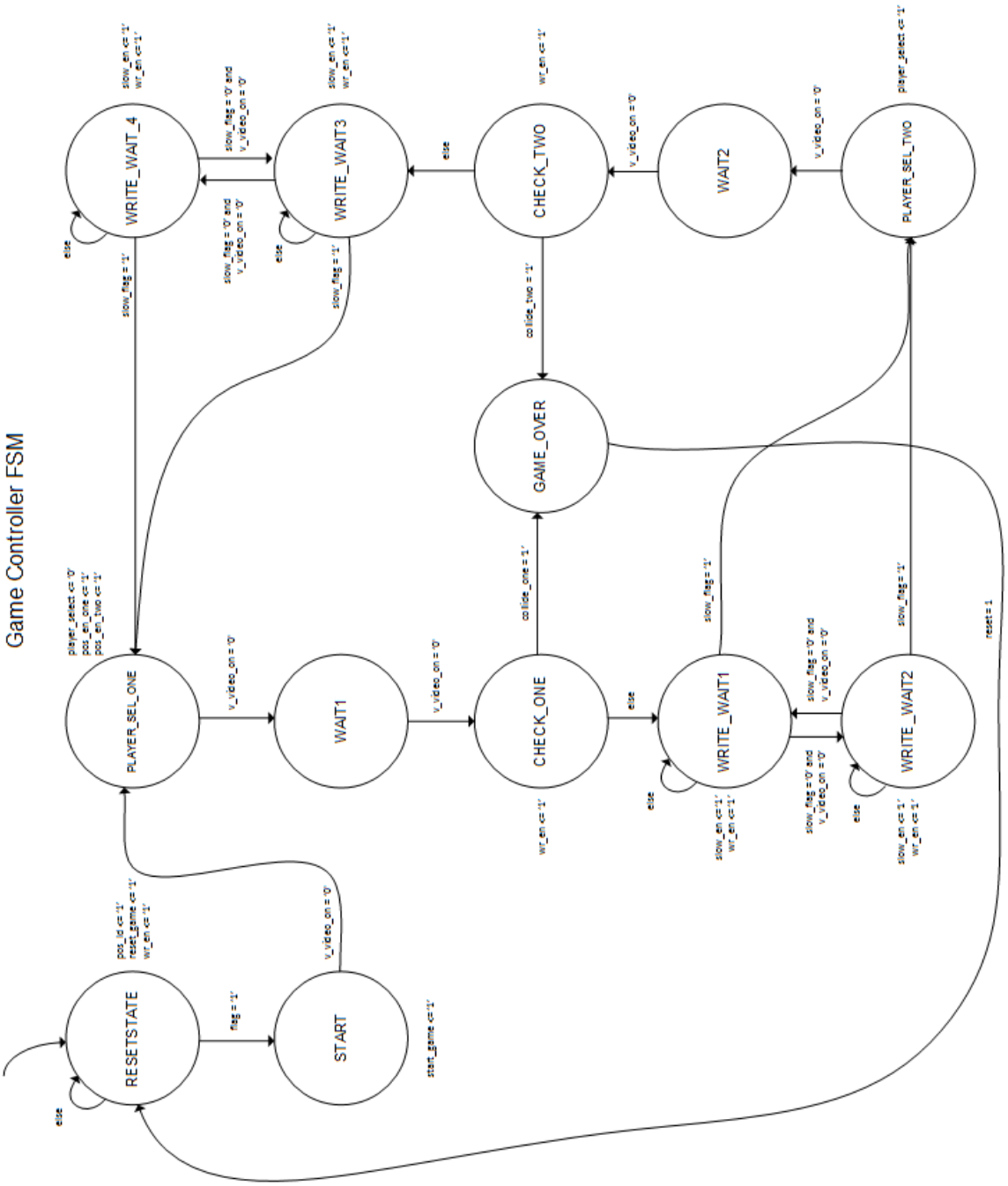


Fig. 13: Game Controller FSM

**c) Directional Controller FSM for Player 1**

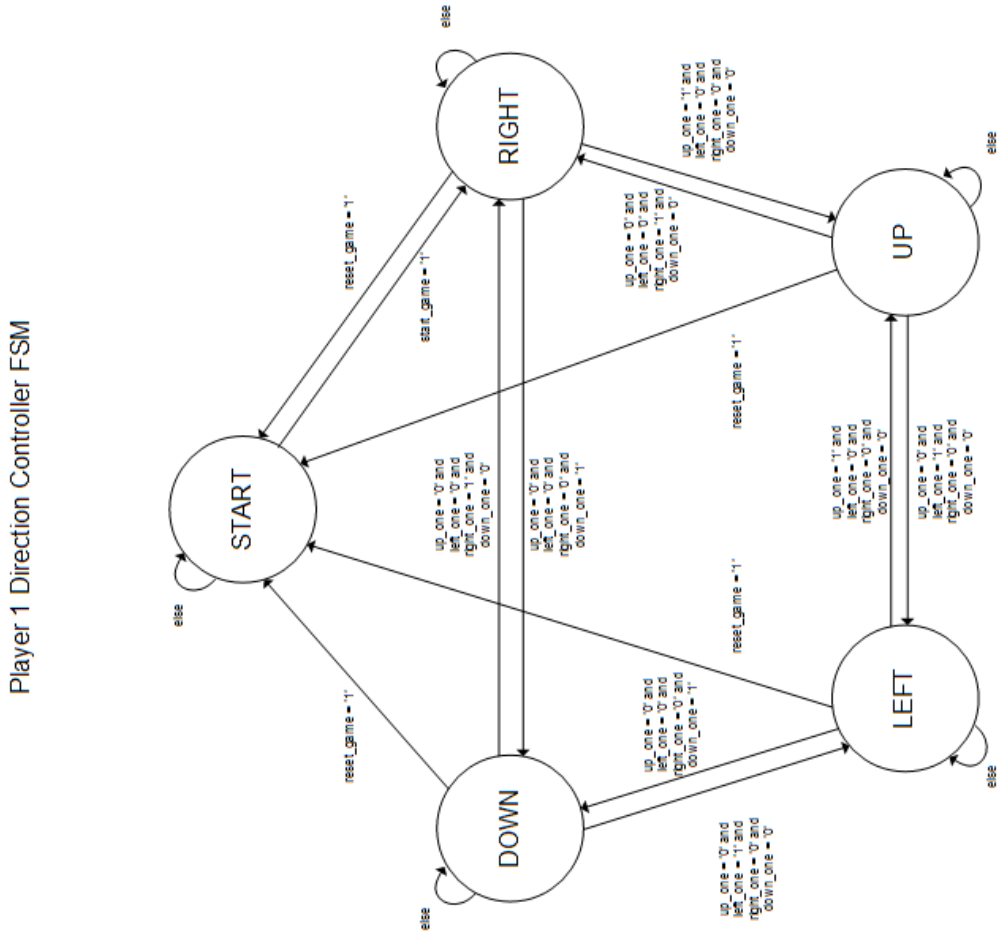


Fig. 14: Directional Controller FSM for Player 1

**d) Directional Controller FSM for Player 2**

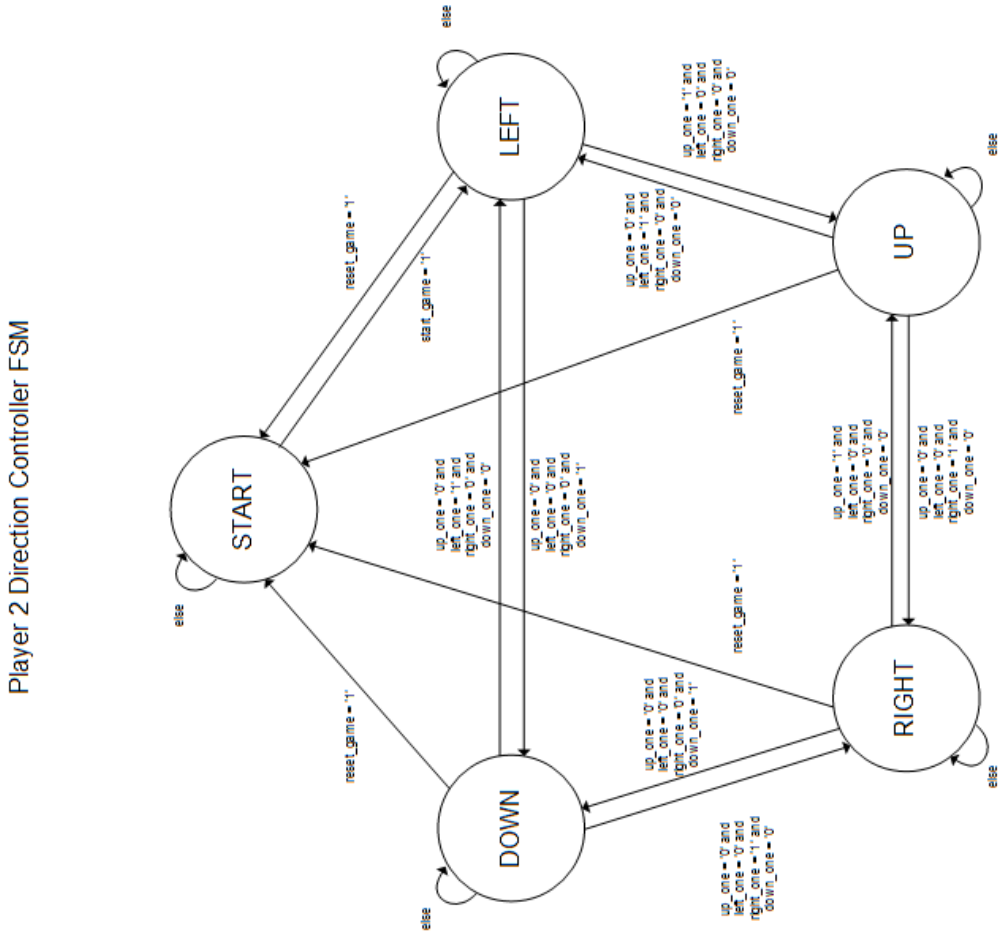


Fig. 15: Directional Controller FSM for Player 2

**e) Debouncer Monopulser FSM**

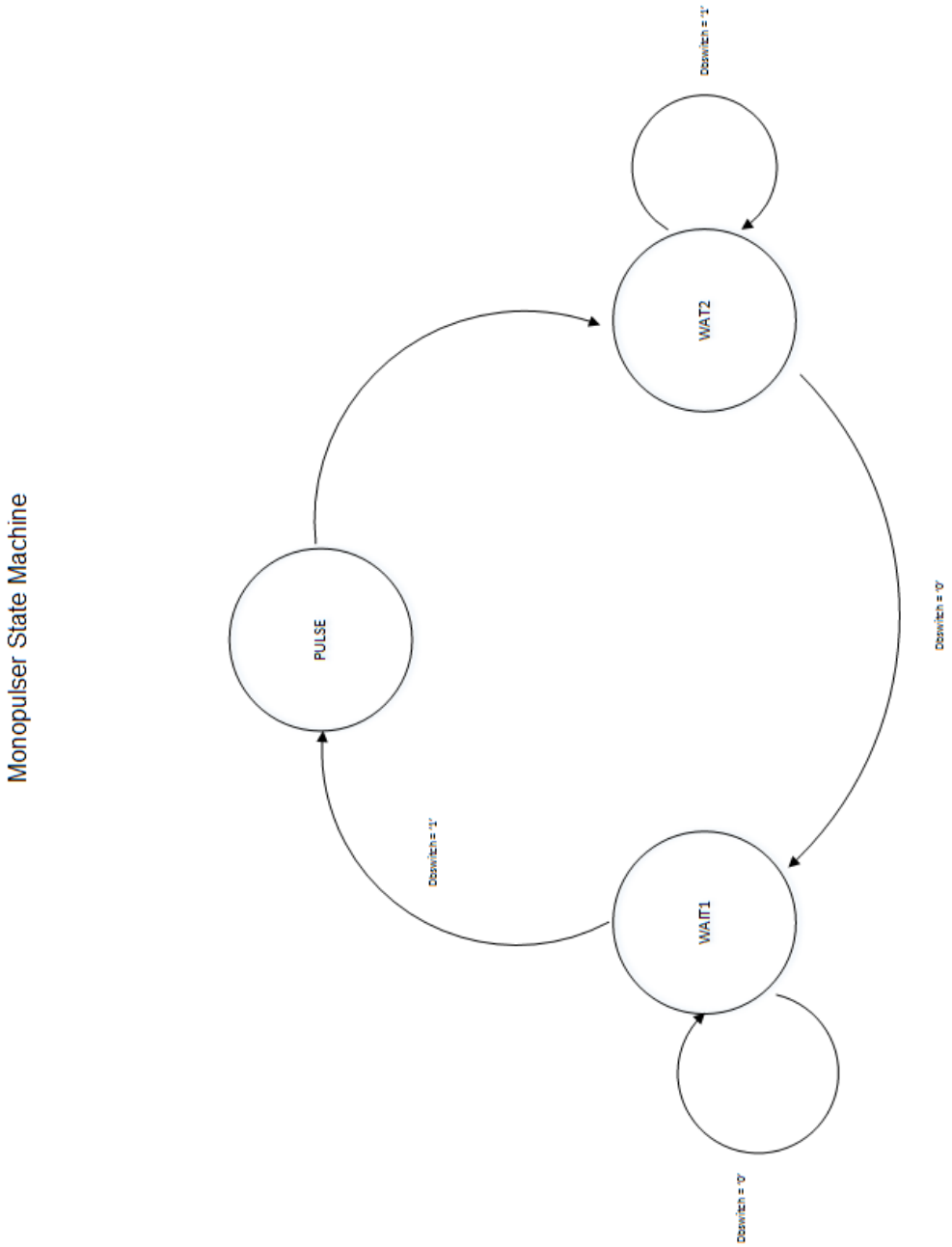


Fig. 16: Debouncer Monopulser FSM

### 7.2.2 VHDL code

#### a) Tron Top file

Listing 1: Top File

```
----------------------------------------------------------------------------------
-- Company:                     ENGS 31 15X
-- Engineer:                    Edrei Chua and Jonathan HUang
--
-- Create Date:                 10:45:01 08/19/2015
-- Design Name:                 Tron_top
-- Module Name:                 Tron_top - Behavioral
-- Project Name:                Tron
-- Target Devices:              Spartan 3E
-- Tool versions:               ISE 14.7
-- Description:                 Top level shell for Tron final project
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Revised (EC) 8.26.15 to include debouncers for all the buttons
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

library UNISIM;
use UNISIM.VComponents.all;

entity Tron_top is
    Port ( clk : in STD_LOGIC;
                        left_one     : in   STD_LOGIC;
                        right_one    : in   STD_LOGIC;
                        up_one       : in   STD_LOGIC;
                        down_one     : in   STD_LOGIC;
                        left_two     : in   STD_LOGIC;
                        right_two    : in   STD_LOGIC;
                        up_two       : in   STD_LOGIC;
                        down_two     : in   STD_LOGIC;
                        reset        : in STD_LOGIC;
           hsync : out STD_LOGIC;
           vsync : out STD_LOGIC;
           color : out STD_LOGIC_VECTOR (7 downto 0));
end Tron_top;

architecture Behavioral of Tron_top is

-- Component Declarations

-- VGA controller declaration
component VGAController is
    Port ( mclk       : in STD_LOGIC;
           pixel_x    : out STD_LOGIC_VECTOR (9 downto 0);
           pixel_y    : out STD_LOGIC_VECTOR (9 downto 0);
           h_video_on  : out STD_LOGIC;
```

```
          v_video_on  : out STD_LOGIC;
          hsync       : out STD_LOGIC;
          vsync       : out STD_LOGIC);
end component;

-- MemoryRAM declaration
component MemoryRAM is
    Port ( clk  : in STD_LOGIC;
          pixel_x_r   : in STD_LOGIC_VECTOR (9 downto 0);
          pixel_y_r   : in STD_LOGIC_VECTOR (9 downto 0);
          pixel_x_w   : in STD_LOGIC_VECTOR (9 downto 0);
          pixel_y_w   : in STD_LOGIC_VECTOR (9 downto 0);
          wr_en       : in STD_LOGIC;
          d_in        : in STD_LOGIC_VECTOR (1 downto 0);
          color       : out STD_LOGIC_VECTOR (7 downto 0);
                        collide : out STD_LOGIC);
end component;

-- Game Controller declaration
component GameController is
    Port ( clk        : in STD_LOGIC;
          left_one    : in STD_LOGIC;
          right_one   : in STD_LOGIC;
          up_one      : in STD_LOGIC;
          down_one    : in STD_LOGIC;
          left_two    : in STD_LOGIC;
          right_two   : in STD_LOGIC;
          up_two      : in STD_LOGIC;
          down_two    : in STD_LOGIC;
          color_in    : in STD_LOGIC;
          h_video_on  : in STD_LOGIC;
          v_video_on  : in STD_LOGIC;
          reset       : in STD_LOGIC;
          color_out   : out STD_LOGIC_VECTOR (1 downto 0);
          pixel_x_r   : out STD_LOGIC_VECTOR (9 downto 0);
          pixel_y_r   : out STD_LOGIC_VECTOR (9 downto 0);
          pixel_x_w   : out STD_LOGIC_VECTOR (9 downto 0);
          pixel_y_w   : out STD_LOGIC_VECTOR (9 downto 0);
          wr_en       : out STD_LOGIC);
end component;

-- Debouncer Monopulser declaration
component debouncer_monopulse is
       port(  clk            : in STD_LOGIC;
              switch         : in STD_LOGIC;
              dbpulseswitch  : out std_logic );
end component;

-- Signal declarations
signal pixel_x_r, pixel_y_r : std_logic_vector(9 downto 0); -- connection from game controller to
    memory
signal pixel_x_r_VGA, pixel_y_r_VGA : std_logic_vector(9 downto 0); -- VGA controller to read from
    memory
signal pixel_x_r_game, pixel_y_r_game : std_logic_vector(9 downto 0); -- Game controller to read
    from memory
signal pixel_x_w, pixel_y_w : std_logic_vector(9 downto 0); -- connection from game controller to
    memory
signal h_video_on, v_video_on : std_logic; -- connection from VGA controller to game controller
signal wr_en : std_logic;    -- connection from game controller to memoryRAM to choose to read or
```

```
    write to memory
signal color_in_ram : std_logic_vector(1 downto 0); -- color to be written to memory
signal reset_sig : std_logic; -- connection from debouncer output to game controller
signal collide_sig : std_logic; -- indicating collision
signal up_one_sig, down_one_sig, left_one_sig, right_one_sig :   STD_LOGIC;     -- connection from
    debouncer output to game controller
signal up_two_sig, down_two_sig, left_two_sig, right_two_sig :   STD_LOGIC;     -- connection from
    debouncer output to game controller

begin

-- VGA controller port map
VGAControl: VGAController PORT MAP(
       mclk => clk,
       pixel_x => pixel_x_r_VGA,
       pixel_y => pixel_y_r_VGA,
       h_video_on => h_video_on,
       v_video_on => v_video_on,
       hsync => hsync,
       vsync => vsync);

-- MemoryRAM port map
Memory: MemoryRAM PORT MAP(
       clk => clk,
       pixel_x_r => pixel_x_r,
       pixel_y_r => pixel_y_r,
       pixel_x_w => pixel_x_w,
       pixel_y_w => pixel_y_w,
       wr_en => wr_en,
       d_in => color_in_ram,
       color => color,
       collide => collide_sig);

-- Game Controller port map
Game: GameController PORT MAP(
       clk => clk,
       left_one => left_one_sig,
       right_one => right_one_sig,
       up_one => up_one_sig,
       down_one => down_one_sig,
       left_two => left_two_sig,
  right_two => right_two_sig,
  up_two => up_two_sig,
  down_two => down_two_sig,
       color_in => collide_sig,
       h_video_on => h_video_on,
       v_video_on => v_video_on,
       reset => reset_sig,
       color_out => color_in_ram,
       pixel_x_r => pixel_x_r_game,
       pixel_y_r => pixel_y_r_game,
       pixel_x_w => pixel_x_w,
       pixel_y_w => pixel_y_w,
       wr_en => wr_en);

-- Debouncer port map
Debounce_reset: debouncer_monopulse PORT MAP(
       clk => clk,
       switch => reset,
```

```
        dbpulseswitch => reset_sig);

Debounce_up_one: debouncer_monopulse PORT MAP(
        clk => clk,
        switch => up_one,
        dbpulseswitch => up_one_sig);

Debounce_down_one: debouncer_monopulse PORT MAP(
        clk => clk,
        switch => down_one,
        dbpulseswitch => down_one_sig);

Debounce_left_one: debouncer_monopulse PORT MAP(
        clk => clk,
        switch => left_one,
        dbpulseswitch => left_one_sig);
Debounce_right_one: debouncer_monopulse PORT MAP(
        clk => clk,
        switch => right_one,
        dbpulseswitch => right_one_sig);
Debounce_up_two: debouncer_monopulse PORT MAP(
        clk => clk,
        switch => up_two,
        dbpulseswitch => up_two_sig);
Debounce_down_two: debouncer_monopulse PORT MAP(
        clk => clk,
        switch => down_two,
        dbpulseswitch => down_two_sig);
Debounce_left_two: debouncer_monopulse PORT MAP(
        clk => clk,
        switch => left_two,
        dbpulseswitch => left_two_sig);
Debounce_right_two: debouncer_monopulse PORT MAP(
        clk => clk,
        switch => right_two,
        dbpulseswitch => right_two_sig);

-- Multiplexer to select pixel_reads
Mux: process(pixel_x_r_game, pixel_y_r_game, pixel_x_r_VGA, pixel_y_r_VGA, v_video_on)
begin
        if v_video_on = '0' then     -- game controller to read from memory when video is off
                pixel_x_r <= pixel_x_r_game;
                pixel_y_r <= pixel_y_r_game;
        else                         -- VGA controller to read from memory when video is on
                pixel_x_r <= pixel_x_r_VGA;
                pixel_y_r <= pixel_y_r_VGA;
        end if;
end process Mux;

end Behavioral;
```

### b) Game Controller

Listing 2: Game Controller

```
----------------------------------------------------------------------------------
-- Company:            Engs 31 15X
-- Engineer:           Edrei Chua and Jonathan Huang
--
-- Create Date:        15:56:17 08/19/2015
-- Design Name:        Tron
-- Module Name:        GameController - Behavioral
-- Project Name:       Tron
-- Target Devices:     Spartan 3E
-- Tool versions:      ISE 14.7
-- Description:        Game controller for the Tron game. It controls the logic of
--                                      the game.
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
--      Revised: 8.20.15 (EC): changed timing of writing to slow down the game
--             Revised: 8.21.15 (JH): changed states of game controller to only control
--                     reading and writing (no counter control)
--             Revised: 8.22.15 (JH): added direction controllers to both players to
--                    facilitate movement
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

library UNISIM;
use UNISIM.VComponents.all;

entity GameController is
    Port ( clk : in STD_LOGIC;
           left_one : in STD_LOGIC;  -- debounced left signal for player 1
           right_one : in STD_LOGIC; -- debounced right signal for player 1
           up_one : in STD_LOGIC;    -- debounced up signal for player 1
           down_one : in STD_LOGIC;  -- debounced down signal for player 1
           left_two : in STD_LOGIC;  -- debounced left signal for player 2
           right_two : in STD_LOGIC; -- debounced right signal for player 2
           up_two : in STD_LOGIC;    -- debounced up signal for player 2
           down_two : in STD_LOGIC;  -- debounced down signal for player 2
           color_in : in STD_LOGIC;  -- color input from MemoryRAM
           h_video_on : in STD_LOGIC; -- 1 during horizontal update, 0 during horizontal refreshes
           v_video_on : in STD_LOGIC; -- 1 during vertical update, 0 during vertical refreshes
           reset : in STD_LOGIC;     -- debounced reset signal
           color_out : out STD_LOGIC_VECTOR (1 downto 0);  -- the color to write to memory
           pixel_x_r : out STD_LOGIC_VECTOR (9 downto 0);  -- the pixel x address to read from
               memory to detect collision
           pixel_y_r : out STD_LOGIC_VECTOR (9 downto 0);  -- the pixel y address to read from
               memory to detect collision
           pixel_x_w : out STD_LOGIC_VECTOR (9 downto 0);  -- the pixel x address to write to memory
           pixel_y_w : out STD_LOGIC_VECTOR (9 downto 0);  -- the pixel y address to write to memory
           wr_en : out STD_LOGIC);   -- write read enable for the memory. 1 to write and 0 to read
end GameController;
```

```
architecture Behavioral of GameController is
-- Constants
       constant PLAYER1_COLOR       : STD_LOGIC_VECTOR(1 downto 0) := "01";           -- Blue
       constant PLAYER2_COLOR       : STD_LOGIC_VECTOR(1 downto 0) := "10";           -- Green
       constant BACKGROUND          : STD_LOGIC_VECTOR(1 downto 0) := "00";           -- Black
       constant RED                 : STD_LOGIC_VECTOR(1 downto 0) := "11";           -- Red

       constant START_POS_X1        : UNSIGNED(9 downto 0) := "0010011111";           -- equals 159
       constant START_POS_Y1        : UNSIGNED(9 downto 0) := "0011101111";           -- equals 239
       constant START_POS_X2        : UNSIGNED(9 downto 0) := "0111011111";           -- equals 479
       constant START_POS_Y2        : UNSIGNED(9 downto 0) := "0011101111";           -- equals 239

-- Direction signals
       signal start_game                   : STD_LOGIC := '0';

       -- Enables to update the position counter for player 1
       signal up_en_one                    : STD_LOGIC := '0';
       signal right_en_one                 : STD_LOGIC := '0';
       signal down_en_one                  : STD_LOGIC := '0';
       signal left_en_one                  : STD_LOGIC := '0';

       -- Enables to update the position counter for player 2
       signal up_en_two                    : STD_LOGIC := '0';
       signal right_en_two                 : STD_LOGIC := '0';
       signal down_en_two                  : STD_LOGIC := '0';
       signal left_en_two                  : STD_LOGIC := '0';

-- Counter signals
       signal bkgd_x : unsigned(9 downto 0) := (others => '0');  -- pixel x count for background
           during reset
       signal bkgd_y : unsigned(9 downto 0) := (others => '0');  -- pixel y count for background
           during reset
       signal iQ1_x : unsigned(9 downto 0) := (others => '0'); -- pixel x count for player 1
       signal iQ1_y : unsigned(9 downto 0) := (others => '0'); -- pixel y count for player 1
       signal iQ2_x : unsigned(9 downto 0) := (others => '0'); -- pixel x count for player 2
       signal iQ2_y : unsigned(9 downto 0) := (others => '0'); -- pixel y count for player 2
       signal pos_ld : STD_LOGIC := '0';   -- load initial positions of both players
       signal pos_en_one: STD_LOGIC := '0'; -- enable the update of player 1's position counter
       signal pos_en_two: STD_LOGIC := '0'; -- enable the update of player 2's position counter

-- Flags
       signal reset_game : STD_LOGIC := '0'; -- reset game signal
       signal player_select: STD_LOGIC := '0'; -- select bit to choose which player position's to
           update; 0 for player 1 and 1 for player 2

       signal slow_count: unsigned(1 downto 0) := "00";  -- count the number of screen refreshes
       signal slow_flag: STD_LOGIC := '0'; -- indicate that the number of screen refreshes has
           reached the limit indicated
       signal slow_en: STD_LOGIC := '0';   -- enable the slow down counter
       signal flag: STD_LOGIC := '0'; -- indicates that all pixels have been updated to the
           background color during a reset

-- Collision signals
       signal collide_one    : STD_LOGIC := '0';   -- player 1 first collides with player 2
       signal collide_two    : STD_LOGIC := '0';   -- player 2 first collides with player 1
       signal border_collide_one    : STD_LOGIC := '0';   -- player 1 collides with the border
       signal border_collide_two    : STD_LOGIC := '0';   -- player 2 collides with the border
```

```
-- Game states
	type game_states is (RESETSTATE, START, PLAYER_SEL_ONE, WAIT1, CHECK_ONE, WRITE_WAIT1,
	    WRITE_WAIT2, PLAYER_SEL_TWO, WAIT2, CHECK_TWO, WRITE_WAIT3, WRITE_WAIT4, GAME_OVER);
	signal curr_game_state, next_game_state : game_states;

-- Player 1 Direction states
	type one_states is (START, RIGHT, LEFT, UP, DOWN);
	signal curr_one_state, next_one_state : one_states;

-- Player 2 Direction states
	type two_states is (START, RIGHT, LEFT, UP, DOWN);
	signal curr_two_state, next_two_state : two_states;

begin

-- Game Controller
GameControl: process(curr_game_state, collide_one, collide_two, v_video_on, flag, reset, slow_flag)
begin
-- defaults
	next_game_state <= curr_game_state;
	wr_en <= '0';
	reset_game <= '0';
	pos_ld <= '0';
	player_select <= '0';
	start_game <= '0';
	pos_en_one <= '0';
	pos_en_two <= '0';
	slow_en <= '0';

	case curr_game_state is
		when RESETSTATE =>    -- Reset state
			pos_ld <= '1'; -- Load both start positions
			if flag = '1' then    -- Start game only once background is reset
				next_game_state <= START;
			elsif v_video_on = '0' then  -- Reset
				reset_game <= '1';
				wr_en <= '1';
			end if;
		when START =>  -- Start state
			if v_video_on = '0' then
				start_game <= '1';    -- Makes both players go in their initial
				    positions
				next_game_state <= PLAYER_SEL_ONE;
			end if;


		when PLAYER_SEL_ONE =>
			if v_video_on = '0' then
				player_select <= '0'; -- Select player 1
				pos_en_one <= '1';    -- Update position
				pos_en_two <= '1';
				next_game_state <= WAIT1;
			end if;
		when WAIT1 =>
			if v_video_on = '0' then
				next_game_state <= CHECK_ONE; -- check if player 1 has collided
			end if;
		when CHECK_ONE =>
			if collide_one = '1' then    -- Collided so game over
```

```
                if v_video_on = '0' then
                        wr_en <= '1';
                        next_game_state <= GAME_OVER;
                end if;
        else
                next_game_state <= WRITE_WAIT1;
        end if;
when WRITE_WAIT1 =>   -- Slow down motorcycle 1
        if slow_flag = '1' then -- Write once counter reaches terminal count
                wr_en <= '1';
                next_game_state <= PLAYER_SEL_TWO;  -- go to player 2
        elsif v_video_on = '1' then  -- toggle between WRITE_WAIT1 and WRITE_WAIT2
                next_game_state <= WRITE_WAIT2;
                slow_en <= '1';
        end if;
when WRITE_WAIT2 =>   -- Slow down motorcycle 2
        if slow_flag = '1' then -- Write once counter reaches terminal count
                wr_en <= '1';
                next_game_state <= PLAYER_SEL_TWO;  -- go to player 2
        elsif v_video_on = '0' then
                next_game_state <= WRITE_WAIT1; -- toggle between WRITE_WAIT1 and
                    WRITE_WAIT2
                slow_en <= '1';
        end if;


when PLAYER_SEL_TWO =>
        if v_video_on = '0' then
                player_select <= '1'; -- Select player 2
                next_game_state <= WAIT2;
        end if;
when WAIT2 =>
        if v_video_on = '0' then
                player_select <= '1';
                next_game_state <= CHECK_TWO; -- check if player 2 has collided
        end if;
when CHECK_TWO =>
        if collide_two = '1' then    -- Collided so game over
                if v_video_on = '0' then
                        player_select <= '1';
                        wr_en <= '1';
                        next_game_state <= GAME_OVER;
                end if;
        else   -- Go to write state
                next_game_state <= WRITE_WAIT3;
        end if;
when WRITE_WAIT3 =>   -- Slow down motorcycle 3
        if slow_flag = '1' then -- Write once counter reaches terminal count
                player_select <= '1';
                wr_en <= '1';
                next_game_state <= PLAYER_SEL_ONE;
        elsif v_video_on = '1' then
                next_game_state <= WRITE_WAIT4; -- toggle between WRITE_WAIT3 and
                    WRITE_WAIT4
                slow_en <= '1';
        end if;
when WRITE_WAIT4 =>   -- Slow down motorcycle 4
        if slow_flag = '1' then -- Write once counter reaches terminal count
                player_select <= '1';
```

```
                        wr_en <= '1';
                        next_game_state <= PLAYER_SEL_ONE;
                    elsif v_video_on = '0' then
                        next_game_state <= WRITE_WAIT3; -- toggle between WRITE_WAIT3 and
                            WRITE_WAIT4
                        slow_en <= '1';
                    end if;
                when GAME_OVER =>
                    if reset = '1' then          -- wait for reset
                        next_game_state <= RESETSTATE;
                    end if;

        end case;
end process GameControl;

-- Player One Direction Controller
DirectionController_One: process(curr_one_state, up_one, down_one, right_one, left_one,
    start_game, reset_game)
begin
-- defaults
        next_one_state <= curr_one_state;
        up_en_one <= '0';
        right_en_one <= '0';
        down_en_one <= '0';
        left_en_one <= '0';

        case curr_one_state is
                when START =>  -- Player 1 initially goes right
                    if start_game = '1' then
                        next_one_state <= RIGHT;
                    end if;
                when RIGHT =>  -- Right
                    right_en_one <= '1';

                    if reset_game = '1' then
                        next_one_state <= START;
                    elsif up_one = '1' and left_one = '0' and right_one = '0' and down_one = '0'
                        then -- Go up
                        next_one_state <= UP;
                    elsif up_one = '0' and left_one = '0' and right_one = '0' and down_one = '1'
                        then -- Go down
                        next_one_state <= DOWN;
                    end if;
                when LEFT =>
                    left_en_one <= '1';

                    if reset_game = '1' then
                        next_one_state <= START;
                    elsif up_one = '1' and left_one = '0' and right_one = '0' and down_one = '0'
                        then -- Go up
                        next_one_state <= UP;
                    elsif up_one = '0' and left_one = '0' and right_one = '0' and down_one = '1'
                        then -- Go down
                        next_one_state <= DOWN;
                    end if;
                when UP =>
                    up_en_one <= '1';

                    if reset_game = '1' then
```

```
                        next_one_state <= START;
                    elsif up_one = '0' and left_one = '0' and right_one = '1' and down_one = '0'
                        then  -- Go right
                            next_one_state <= RIGHT;
                    elsif up_one = '0' and left_one = '1' and right_one = '0' and down_one = '0'
                        then -- Go left
                            next_one_state <= LEFT;
                    end if;
                when DOWN =>
                        down_en_one <= '1';

                        if reset_game = '1' then
                            next_one_state <= START;
                        elsif up_one = '0' and left_one = '0' and right_one = '1' and down_one = '0'
                            then  -- Go right
                            next_one_state <= RIGHT;
                        elsif up_one = '0' and left_one = '1' and right_one = '0' and down_one = '0'
                            then -- Go left
                            next_one_state <= LEFT;
                        end if;
        end case;
end process DirectionController_one;

-- Player 2 Direction Controller
DirectionController_Two: process(curr_two_state, up_two, down_two, right_two, left_two,
    start_game, reset_game)
begin
-- defaults
        next_two_state <= curr_two_state;
        up_en_two <= '0';
        right_en_two <= '0';
        down_en_two <= '0';
        left_en_two <= '0';

        case curr_two_state is
                when START =>  -- Player 2 initially goes left
                        if start_game = '1' then
                            next_two_state <= LEFT;
                        end if;
                when RIGHT =>
                        right_en_two <= '1';

                        if reset_game = '1' then
                            next_two_state <= START;
                        elsif up_two = '1' and left_two = '0' and right_two = '0' and down_two = '0'
                            then     -- go up
                            next_two_state <= UP;
                        elsif up_two = '0' and left_two = '0' and right_two = '0' and down_two = '1'
                            then     -- go down
                            next_two_state <= DOWN;
                        end if;
                when LEFT =>
                        left_en_two <= '1';

                        if reset_game = '1' then
                            next_two_state <= START;
                        elsif up_two = '1' and left_two = '0' and right_two = '0' and down_two = '0'
                            then -- go up
                            next_two_state <= UP;
```

```
                elsif up_two = '0' and left_two = '0' and right_two = '0' and down_two = '1'
                    then -- go down
                        next_two_state <= DOWN;
                end if;
            when UP =>
                up_en_two <= '1';

                if reset_game = '1' then
                        next_two_state <= START;
                elsif up_two = '0' and left_two = '0' and right_two = '1' and down_two = '0'
                    then  -- go right
                        next_two_state <= RIGHT;
                elsif up_two = '0' and left_two = '1' and right_two = '0' and down_two = '0'
                    then -- go left
                        next_two_state <= LEFT;
                end if;
            when DOWN =>
                down_en_two <= '1';

                if reset_game = '1' then
                        next_two_state <= START;
                elsif up_two = '0' and left_two = '0' and right_two = '1' and down_two = '0'
                    then  -- go right
                        next_two_state <= RIGHT;
                elsif up_two = '0' and left_two = '1' and right_two = '0' and down_two = '0'
                    then -- go left
                        next_two_state <= LEFT;
                end if;
        end case;
end process DirectionController_Two;

-- COUNTERS

-- Background Counter
BackgroundCounter: process(clk, reset_game, flag)
begin
        if rising_edge(clk) then
                flag <= '0';

                if(reset_game= '1') then
                -- Horizontal Counter
                if bkgd_x < 639 then  -- increment
                        bkgd_x <= bkgd_x + 1;
                else
                        bkgd_x <= (others => '0');   -- reset
                        -- Vertical Counter
                        if bkgd_y < 479 then  -- increment
                                bkgd_y <= bkgd_y + 1;
                        else    -- reset
                                bkgd_y <= (others => '0');
                                flag <= '1';
                        end if;
                end if;
                end if;
        end if;
end process BackgroundCounter;

-- Position Counter for player 1
```

```
Player_One_Counter: process(clk, up_en_one, down_en_one, right_en_one, left_en_one, pos_ld,
    pos_en_one, iQ1_x, iQ1_y)
begin
	if rising_edge(clk) then
		iQ1_x <= iQ1_x;
		iQ1_y <= iQ1_y;

		if pos_ld = '1' then  -- load position for player 1
			iQ1_x <= START_POS_X1;
			iQ1_y <= START_POS_Y1;
		elsif pos_en_one = '1' then
			if up_en_one = '1' then -- up
				if iQ1_y < 4 then
					border_collide_one <= '1';
				else
					iQ1_y <= iQ1_y -2;
					border_collide_one <= '0';
				end if;
			elsif down_en_one = '1' then -- down
				if iQ1_y > 477 then
					border_collide_one <= '1';
				else
					iQ1_y <= iQ1_y +2;
					border_collide_one <= '0';
				end if;
			elsif left_en_one = '1' then -- left
				if iQ1_x < 4 then
					border_collide_one <= '1';
				else
					iQ1_x <= iQ1_x -2;
					border_collide_one <= '0';
				end if;
			elsif right_en_one = '1' then       -- right
				if iQ1_x > 637 then
					border_collide_one <= '1';
				else
					iQ1_x <= iQ1_x +2;
					border_collide_one <= '0';
				end if;
			end if;
		end if;
	end if;
end process Player_One_Counter;

-- Position Counter for player 2
Player_Two_Counter: process(clk, up_en_two, down_en_two, right_en_two, left_en_two, pos_ld,
    pos_en_two, iQ2_x, iQ2_y)
begin
	if rising_edge(clk) then
		iQ2_x <= iQ2_x;
		iQ2_y <= iQ2_y;

		if pos_ld = '1' then  -- load position for player 2
			iQ2_x <= START_POS_X2;
			iQ2_y <= START_POS_Y2;
		elsif pos_en_two = '1' then
			if up_en_two = '1' then -- up
				if iQ2_y < 4 then
					border_collide_two <= '1';
```

```
                        else
                                iQ2_y <= iQ2_y -2;
                                border_collide_two <= '0';
                        end if;
                elsif down_en_two = '1' then  -- down
                        if iQ2_y > 477 then
                                border_collide_two <= '1';
                        else
                                iQ2_y <= iQ2_y +2;
                                border_collide_two <= '0';
                        end if;
                elsif left_en_two = '1' then -- left
                        if iQ2_x < 4 then
                                border_collide_two <= '1';
                        else
                                iQ2_x <= iQ2_x -2;
                                border_collide_two <= '0';
                        end if;
                elsif right_en_two = '1' then       -- right
                        if iQ2_x > 637 then
                                border_collide_two <= '1';
                        else
                                iQ2_x <= iQ2_x +2;
                                border_collide_two <= '0';
                        end if;
                end if;
        end if;
    end if;
end process Player_Two_Counter;

--Slow Down counter
SlowDown: process(clk, slow_en)
begin
        if rising_edge(clk) then
                slow_count <= slow_count;
                slow_flag <= '0';
                if slow_en = '1' then -- Only count if enabled
                        if slow_count = "10" then
                                slow_count <= "00";
                                slow_flag <= '1';
                        else
                                slow_count <= slow_count + 1;
                        end if;
                end if;

        end if;
end process SlowDown;




-- MULTIPLEXERS AND CHECKS

-- Collision Check
CheckCollision: process(iQ1_x, iQ1_y, iQ2_x, iQ2_y, player_select, border_collide_one,
    border_collide_two, color_in, v_video_on)
begin
        collide_one <= '0';
        collide_two <= '0';
```

```
        if player_select = '0' then  -- Get player 1's address for reading
                pixel_x_r <= std_logic_vector(iQ1_x);
                pixel_y_r <= std_logic_vector(iQ1_y);
        else
                                                                        -- Get player 2's
            address for reading
                pixel_x_r <= std_logic_vector(iQ2_x);
                pixel_y_r <= std_logic_vector(iQ2_y);
        end if;

        if v_video_on = '0' then
                if color_in = '1' or border_collide_one = '1' then -- Check collision for player 1
                        collide_one <= '1';
                else
                        collide_one <= '0';
                end if;

                if color_in = '1' or border_collide_two = '1' then -- Check collision for player 2
                        collide_two <= '1';
                else
                        collide_two <= '0';
                end if;
        end if;

end process CheckCollision;

-- Color Multiplexer
ColorMux: process(reset_game, player_select, collide_one, collide_two)
begin
        if reset_game = '1' then     -- BLACK when reset
                color_out <= BACKGROUND;
        elsif collide_one = '1' or collide_two = '1' then  -- RED when collided
                color_out <= RED;
        elsif player_select = '0' then      -- BLUE for player 1
                color_out <= PLAYER1_COLOR;
        elsif player_select = '1' then      -- GREEN for player 2
                color_out <= PLAYER2_COLOR;
        else
                color_out <= BACKGROUND;
        end if;
end process ColorMux;

-- Player Multiplexer
PlayerMux: process(reset_game, player_select, bkgd_x, bkgd_y, iQ1_x, iQ1_y, iQ2_x, iQ2_y)
begin
        if reset_game = '1' then     -- Write using background address
                pixel_x_w <= std_logic_vector(bkgd_x);
                pixel_y_w <= std_logic_vector(bkgd_y);
        elsif player_select = '0' then      -- Write player 1's address
                pixel_x_w <= std_logic_vector(iQ1_x);
                pixel_y_w <= std_logic_vector(iQ1_y);
        elsif player_select = '1' then  -- Write player 2's address
                pixel_x_w <= std_logic_vector(iQ2_x);
                pixel_y_w <= std_logic_vector(iQ2_y);
        else
                pixel_x_w <= std_logic_vector(iQ1_x);
                pixel_y_w <= std_logic_vector(iQ1_y);
        end if;
```

```
end process PlayerMux;

-- State Update
StateUpdate: process(clk)
begin
	if rising_edge(clk) then
		curr_game_state <= next_game_state;
		curr_one_state <= next_one_state;
		curr_two_state <= next_two_state;
	end if;
end process StateUpdate;

end Behavioral;
```

### c) VGA Controller

Listing 3: VGA Controller

```
----------------------------------------------------------------------------------
-- Company:           Engs 31 15X
-- Engineer:          Edrei Chua and Jonathan Huang
--
-- Create Date:       22:07:26 08/11/2015
-- Design Name:       Tron
-- Module Name:       VGAController - Behavioral
-- Project Name:      Tron
-- Target Devices:    Spartan 3E
-- Tool versions:     ISE 14.7
-- Description:       Controller that controls the display of the VGA display
--
-- Dependencies:
--
-- Revision: 1.01 - 08/19/2015: Edited to change when v_video_on turns off
-- Revision: 1.00 - 08/13/2015: Put counters into separate clocked process.
--      Made vertical/horizontal logic asynchronous
--             Adjusted clock to make it 25 MHz
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

library UNISIM;
use UNISIM.VComponents.all;

entity VGAController is
    Port ( mclk : in STD_LOGIC;
           pixel_x : out STD_LOGIC_VECTOR (9 downto 0);
           pixel_y : out STD_LOGIC_VECTOR (9 downto 0);
           h_video_on : out STD_LOGIC;
           v_video_on : out STD_LOGIC;
           hsync : out STD_LOGIC;
           vsync : out STD_LOGIC);
end VGAController;

architecture Behavioral of VGAController is
-- internal signals
       signal iQ_x :  unsigned(9 downto 0) := (others => '0');
       signal iQ_y :   unsigned(9 downto 0) := (others => '0');
       signal v_en : std_logic := '0';


-- Signals for the clock divider, which divides the matser clock down to 25 MHz
-- Master clock frequency / CLOCK_DIVIDER_VALUE = 25 MHz
constant CDV2: integer := 50E6/50E6;        -- Nexys2 board has 50 MHz
constant CLOCK_DIVIDER_VALUE: integer := CDV2;     -- 10 for simulation, CDV2 for implementation
signal clkdiv : integer := 0; -- the clock divider count
signal clkdiv_tog: std_logic := '0';        -- terminal count
signal clk25 : std_logic := '0';    -- 25 MHz clock

begin
```

```
-- Clock buffer for 25 MHz clock
-- The BUFG component puts the slower clock onto the FPGA clocking network
Slow_clock_buffer: BUFG
       port map (I => clkdiv_tog,
                           O => clk25);

-- Divide the master clock down to 50 MHz, then toggling the clkdiv_tog at 50 MHz
-- gives a 25 MHz clock with 50% duty cycle.
Clock_divider: process(mclk)
begin
       if rising_edge(mclk) then
              if clkdiv = CLOCK_DIVIDER_VALUE-1 then
                     clkdiv_tog <= NOT(clkdiv_tog);
                     clkdiv <= 0;
              else
                     clkdiv <= clkdiv + 1;
              end if;
       end if;
end process Clock_divider;

-- Horizontal Logic
Horizontal: process(iQ_x)
begin
       hsync <= '1';
       h_video_on <= '0';
       v_en <= '0';                                    -- default v_en

       if iQ_x < 640 then                              -- display
              hsync <= '1';
       elsif iQ_x < 656 then                           -- right border
              h_video_on <= '0';
              hsync <= '1';
       elsif iQ_x < 752 then                           -- retrace
              h_video_on <= '0';
              hsync <= '0';
       elsif iQ_x < 800 then                           -- left border
              h_video_on <= '0';
              hsync <= '1';

              if iQ_x = 799 then
                     v_en <= '1';
              end if;
       end if;

end process Horizontal;

-- Vertical Logic
Vertical: process(iQ_y)
begin
       vsync <= '1';
       v_video_on <= '0';

       if iQ_y < 480 then                              -- display
              v_video_on <= '1';
              vsync <= '1';
       elsif iQ_y < 490 then                           -- bottom border
              v_video_on <= '1';
              vsync <= '1';
       elsif iQ_y < 492 then                           -- retrace
```

```
                v_video_on <= '0';
                vsync <= '0';
        elsif iQ_y < 521 then                           -- top border
                v_video_on <= '0';
                vsync <= '1';
        end if;

end process Vertical;

-- Counter
Counter: process(clk25)
begin
        if rising_edge(clk25) then
                -- Horizontal Counter
                if iQ_x < 799 then                      -- increment
                        iQ_x <= iQ_x + 1;
                else
                        iQ_x <= (others => '0');        -- reset
                end if;

                -- Vertical Counter
                if v_en = '1' then
                        if iQ_y < 520 then              -- increment
                                iQ_y <= iQ_y + 1;
                        else                            -- reset
                                iQ_y <= (others => '0');
                        end if;
                end if;
        end if;

end process Counter;

-- convert back to std_logic_vector

pixel_x <= std_logic_vector(iQ_x);
pixel_y <= std_logic_vector(iQ_y);

end Behavioral;
```

### d) Memory RAM

Listing 4: Memory RAM

```
 ----------------------------------------------------------------------------------
-- Company:            Engs 31 15X
-- Engineer:           Edrei Chua and Jonathan Huang
--
-- Create Date:        20:03:20 08/13/2015
-- Design Name:        Tron
-- Module Name:        MemoryRAM - Behavioral
-- Project Name:       Tron
-- Target Devices:     Spartan 3E
-- Tool versions:      ISE 14.7
-- Description:               Pixel generation circuit to store the memory (color) at each
--                                   pixel. Downsized to 320x240 to adjust for memory restrictions
--                                   on FPGA.
--
-- Dependencies:
--
-- Revision 1.00 - 08/15/2015: Added process to convert output (2-bit) of
--                                                                          memory into actual color.
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

library UNISIM;
use UNISIM.VComponents.all;

entity MemoryRAM is
    Port ( clk : in STD_LOGIC;
          pixel_x_r : in STD_LOGIC_VECTOR (9 downto 0);    -- pixel x address to read from memory
           pixel_y_r : in STD_LOGIC_VECTOR (9 downto 0);   -- pixel y address to read from memory
           pixel_x_w : in STD_LOGIC_VECTOR (9 downto 0);   -- pixel x address to write to memory
           pixel_y_w : in STD_LOGIC_VECTOR (9 downto 0);   -- pixel y address to write to memory
           wr_en : in STD_LOGIC;      -- 1 to write to memory and 0 to read from memory
           d_in : in STD_LOGIC_VECTOR (1 downto 0); -- color to write to memory
           color : out STD_LOGIC_VECTOR (7 downto 0); -- color read from memory
                       collide: out STD_LOGIC);   -- collision detection
end MemoryRAM;

architecture Behavioral of MemoryRAM is
-- BRAM signals
        signal rd_addr, wr_addr : std_logic_vector (16 downto 0) := (others => '0'); -- read/write
            address
        signal d_out : std_logic_vector (1 downto 0) := "00"; -- output of memory
        constant RAM_ADDR_BITS : integer := 17;
        constant RAM_WIDTH : integer := 2;
        type ram_type is array(2**RAM_ADDR_BITS-1 downto 0) of std_logic_vector (RAM_WIDTH-1 downto
            0);
        signal myRAM : ram_type;

-- Predefined 8-bit colors
        constant RED          : std_logic_vector(7 downto 0) := "11100000";
```

```
        constant GREEN        : std_logic_vector(7 downto 0) := "00011100";
        constant BLUE         : std_logic_vector(7 downto 0) := "00011011";
        constant YELLOW       : std_logic_vector(7 downto 0) := "11111100";
        constant BLACK        : std_logic_vector(7 downto 0) := "00000000";
        constant GRAY0        : std_logic_vector(7 downto 0) := "01001001";
        constant GRAY1        : std_logic_vector(7 downto 0) := "10010010";
begin

-- BRAM memory
wr_addr <= pixel_y_w(8 downto 1) & pixel_x_w(9 downto 1); -- concaternating the write address and
    group 4 pixels to one
rd_addr <= pixel_y_r(8 downto 1) & pixel_x_r(9 downto 1); -- concaternating the read address and
    group 4 pixels to one

BRAM: process(clk)
begin
        if rising_edge(clk) then
                if wr_en = '1' then   -- Write
                        myRAM(to_integer(unsigned(wr_addr))) <= d_in;
                else    -- Read
                        d_out <= myRAM(to_integer(unsigned(rd_addr)));
                end if;
        end if;
end process BRAM;

-- Convert d_out of memory to actual color
ColorMap: process(d_out)
begin

        if d_out = "00" then
                color <= BLACK;
                collide <= '0';
        elsif d_out = "01" then
                color <= BLUE;
                collide <= '1';
        elsif d_out = "10" then
                color <= GREEN;
                collide <= '1';
        else
                color <= RED;
                collide <= '1';
        end if;
end process ColorMap;


end Behavioral;
```

### e) Debouncer Monopulser

Listing 5: Debouncer Monopulser

```
----------------------------------------------------------------------------------
-- Company:                     Engs 31 15X
-- Engineer:                    Edrei Chua and Jonathan Huang
--
-- Create Date:                 08/06/2015
-- Design Name:
-- Module Name:                 debouncer_monopulser - Behavioral
-- Project Name:
-- Target Devices:              Spartan 3E
-- Tool versions:               ISE 14.7
-- Description:                 Debouncer code provided by Prof Hansen, integrated with a monopulser
--
-- Dependencies:
--
--
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

entity debouncer_monopulse is
       port(  clk, switch    : in STD_LOGIC;
              dbpulseswitch  : out std_logic );
end debouncer_monopulse;

architecture behavioral of debouncer_monopulse is
constant REG_LEN : integer := 10;                          -- typ 10ms, lengthen for more delay
signal dbreg   :      std_logic_vector(REG_LEN-1 downto 0) := (others => '0');
signal dbswitch : std_logic := '0';

-- signal for monopulser
type state_monopulser is (WAIT1, PULSE, WAIT2);
signal curr_monopulser, next_monopulser: state_monopulser; -- state for monopulser

begin

debounce:
process(clk, dbreg)
begin
       if rising_edge(clk) then
              if switch /= dbreg(0) then
                     dbreg <= switch & dbreg(REG_LEN-1 downto 1);
              else
                     dbreg <= (others => switch);
              end if;
       end if;

       dbswitch <= dbreg(0);
end process;

-- Combinational logic for monopulser
CombLogicMonopulser: process(curr_monopulser, dbswitch)
```

```
begin
-- default
dbpulseswitch <= '0';
next_monopulser <= curr_monopulser;

case curr_monopulser is
       when WAIT1 =>         -- next state is pulse if button is pressed
              if dbswitch ='1' then
                     next_monopulser <= PULSE
              end if;
       when PULSE => dbpulseswitch <= '1';        -- next state is WAIT2
              next_monopulser <= WAIT2;
       when WAIT2 =>         -- next state is WAIT1 after button is released
              if dbswitch='0' then
                     next_monopulser <= WAIT1;
              end if;
end case;
end process CombLogicMonopulser;

-- State update for monopulser
StateUpdate: process(clk)
begin
if rising_edge(clk) then
       curr_monopulser <= next_monopulser;
end if;
end process StateUpdate;

end behavioral;
```

### f) UCF

Listing 6: UCF

```
 # www.rose-hulman.edu/Class/ee/radu/DIGILENT/index_files/Nexys2General.ucf
# This file is a general .ucf for Nexys2 rev A board
# To use it in a project:
# - remove or comment the lines corresponding to unused pins
# - rename the used signals according to the project

# Signals Led<7> Led<4> are assigned to pins which change type from s3e500 to other dies using the
    same package
# Both versions are provided in this file.
# Keep only the appropriate one, and remove or comment the other one.


# clock pin for Nexys 2 Board
NET "clk" LOC = "B8"; # Bank = 0, Pin name = IP_L13P_0/GCLK8, Type = GCLK, Sch name = GCLK0
#NET "clk1" LOC = "U9"; # Bank = 2, Pin name = IO_L13P_2/D4/GCLK14, Type = DUAL/GCLK, Sch name =
    GCLK1

# onBoard USB controller
#NET "EppAstb" LOC = "V14"; # Bank = 2, Pin name = IP_L23P_2, Type = INPUT, Sch name = U-FLAGA
#NET "EppDstb" LOC = "U14"; # Bank = 2, Pin name = IP_L23N_2, Type = INPUT, Sch name = U-FLAGB
#NET "UsbFlag" LOC = "V16"; # Bank = 2, Pin name = IP, Type = INPUT, Sch name = U-FLAGC
#NET "EppWait" LOC = "N9"; # Bank = 2, Pin name = IO_L12P_2/D7/GCLK12, Type = DUAL/GCLK, Sch name
    = U-SLRD
#NET "EppDB<0>" LOC = "R14"; # Bank = 2, Pin name = IO_L24N_2/A20, Type = DUAL, Sch name = U-FD0
#NET "EppDB<1>" LOC = "R13"; # Bank = 2, Pin name = IO_L22N_2/A22, Type = DUAL, Sch name = U-FD1
#NET "EppDB<2>" LOC = "P13"; # Bank = 2, Pin name = IO_L22P_2/A23, Type = DUAL, Sch name = U-FD2
#NET "EppDB<3>" LOC = "T12"; # Bank = 2, Pin name = IO_L20P_2, Type = I/O, Sch name = U-FD3
#NET "EppDB<4>" LOC = "N11"; # Bank = 2, Pin name = IO_L18N_2, Type = I/O, Sch name = U-FD4
#NET "EppDB<5>" LOC = "R11"; # Bank = 2, Pin name = IO, Type = I/O, Sch name = U-FD5
#NET "EppDB<6>" LOC = "P10"; # Bank = 2, Pin name = IO_L15N_2/D1/GCLK3, Type = DUAL/GCLK, Sch name
    = U-FD6
#NET "EppDB<7>" LOC = "R10"; # Bank = 2, Pin name = IO_L15P_2/D2/GCLK2, Type = DUAL/GCLK, Sch name
    = U-FD7
#
#NET "UsbClk" LOC = "T15"; # Bank = 2, Pin name = IO/VREF_2, Type = VREF, Sch name = U-IFCLK
#
#NET "UsbOE" LOC = "V15"; # Bank = 2, Pin name = IO_L25P_2/VS2/A19, Type = DUAL, Sch name = U-SLOE
#NET "UsbWR" LOC = "V9"; # Bank = 2, Pin name = IO_L13N_2/D3/GCLK15, Type = DUAL/GCLK, Sch name =
    U-SWLR
#NET "UsbPktEnd" LOC = "V12"; # Bank = 2, Pin name = IO_L19P_2, Type = I/O, Sch name = U-PKTEND
#
#NET "UsbDir" LOC = "T16"; # Bank = 2, Pin name = IO_L26P_2/VS0/A17, Type = DUAL, Sch name = U-SLCS
#NET "UsbMode" LOC = "U15"; # Bank = 2, Pin name = IO_L25N_2/VS1/A18, Type = DUAL, Sch name =
    U-INT0#
#
#NET "UsbAdr<0>" LOC = "T14"; # Bank = 2, Pin name = IO_L24P_2/A21, Type = DUAL, Sch name =
    U-FIFOAD0
#NET "UsbAdr<1>" LOC = "V13"; # Bank = 2, Pin name = IO_L19N_2/VREF_2, Type = VREF, Sch name =
    U-FIFOAD1
#
##NET "UsbRdy" LOC = "U13"; # Bank = 2, Pin name = IP, Type = INPUT, Sch name = U-RDY
#
## onBoard Cellular RAM and StrataFlash
#NET "MemOE" LOC = "T2"; # Bank = 3, Pin name = IO_L24P_3, Type = I/O, Sch name = OE
```

```
#NET "MemWR" LOC = "N7"; # Bank = 2, Pin name = IO_L07P_2, Type = I/O, Sch name = WE
#
#NET "RamAdv" LOC = "J4"; # Bank = 3, Pin name = IO_L11N_3/LHCLK1, Type = LHCLK, Sch name = MT-ADV
#NET "RamCS" LOC = "R6"; # Bank = 2, Pin name = IO_L05P_2, Type = I/O, Sch name = MT-CE
#NET "RamClk" LOC = "H5"; # Bank = 3, Pin name = IO_L08N_3, Type = I/O, Sch name = MT-CLK
#NET "RamCRE" LOC = "P7"; # Bank = 2, Pin name = IO_L07N_2, Type = I/O, Sch name = MT-CRE
#NET "RamLB" LOC = "K5"; # Bank = 3, Pin name = IO_L14N_3/LHCLK7, Type = LHCLK, Sch name = MT-LB
#NET "RamUB" LOC = "K4"; # Bank = 3, Pin name = IO_L13N_3/LHCLK5, Type = LHCLK, Sch name = MT-UB
#NET "RamWait" LOC = "F5"; # Bank = 3, Pin name = IP, Type = INPUT, Sch name = MT-WAIT
#
#NET "FlashRp" LOC = "T5"; # Bank = 2, Pin name = IO_L04N_2, Type = I/O, Sch name = RP#
#NET "FlashCS" LOC = "R5"; # Bank = 2, Pin name = IO_L04P_2, Type = I/O, Sch name = ST-CE
#NET "FlashStSts" LOC = "D3"; # Bank = 3, Pin name = IP, Type = INPUT, Sch name = ST-STS
#
#NET "MemAdr<1>" LOC = "J1"; # Bank = 3, Pin name = IO_L12P_3/LHCLK2, Type = LHCLK, Sch name = ADR1
#NET "MemAdr<2>" LOC = "J2"; # Bank = 3, Pin name = IO_L12N_3/LHCLK3/IRDY2, Type = LHCLK, Sch name
    = ADR2
#NET "MemAdr<3>" LOC = "H4"; # Bank = 3, Pin name = IO_L09P_3, Type = I/O, Sch name = ADR3
#NET "MemAdr<4>" LOC = "H1"; # Bank = 3, Pin name = IO_L10N_3, Type = I/O, Sch name = ADR4
#NET "MemAdr<5>" LOC = "H2"; # Bank = 3, Pin name = IO_L10P_3, Type = I/O, Sch name = ADR5
#NET "MemAdr<6>" LOC = "J5"; # Bank = 3, Pin name = IO_L11P_3/LHCLK0, Type = LHCLK, Sch name = ADR6
#NET "MemAdr<7>" LOC = "H3"; # Bank = 3, Pin name = IO_L09N_3, Type = I/O, Sch name = ADR7
#NET "MemAdr<8>" LOC = "H6"; # Bank = 3, Pin name = IO_L08P_3, Type = I/O, Sch name = ADR8
#NET "MemAdr<9>" LOC = "F1"; # Bank = 3, Pin name = IO_L05P_3, Type = I/O, Sch name = ADR9
#NET "MemAdr<10>" LOC = "G3"; # Bank = 3, Pin name = IO_L06P_3, Type = I/O, Sch name = ADR10
#NET "MemAdr<11>" LOC = "G6"; # Bank = 3, Pin name = IO_L07P_3, Type = I/O, Sch name = ADR11
#NET "MemAdr<12>" LOC = "G5"; # Bank = 3, Pin name = IO_L07N_3, Type = I/O, Sch name = ADR12
#NET "MemAdr<13>" LOC = "G4"; # Bank = 3, Pin name = IO_L06N_3/VREF_3, Type = VREF, Sch name =
    ADR13
#NET "MemAdr<14>" LOC = "F2"; # Bank = 3, Pin name = IO_L05N_3, Type = I/O, Sch name = ADR14
#NET "MemAdr<15>" LOC = "E1"; # Bank = 3, Pin name = IO_L03N_3, Type = I/O, Sch name = ADR15
#NET "MemAdr<16>" LOC = "M5"; # Bank = 3, Pin name = IO_L19P_3, Type = I/O, Sch name = ADR16
#NET "MemAdr<17>" LOC = "E2"; # Bank = 3, Pin name = IO_L03P_3, Type = I/O, Sch name = ADR17
#NET "MemAdr<18>" LOC = "C2"; # Bank = 3, Pin name = IO_L01N_3, Type = I/O, Sch name = ADR18
#NET "MemAdr<19>" LOC = "C1"; # Bank = 3, Pin name = IO_L01P_3, Type = I/O, Sch name = ADR19
#NET "MemAdr<20>" LOC = "D2"; # Bank = 3, Pin name = IO_L02N_3/VREF_3, Type = VREF, Sch name =
    ADR20
#NET "MemAdr<21>" LOC = "K3"; # Bank = 3, Pin name = IO_L13P_3/LHCLK4/TRDY2, Type = LHCLK, Sch
    name = ADR21
#NET "MemAdr<22>" LOC = "D1"; # Bank = 3, Pin name = IO_L02P_3, Type = I/O, Sch name = ADR22
#NET "MemAdr<23>" LOC = "K6"; # Bank = 3, Pin name = IO_L14P_3/LHCLK6, Type = LHCLK, Sch name =
    ADR23
#
#NET "MemDB<0>" LOC = "L1"; # Bank = 3, Pin name = IO_L15P_3, Type = I/O, Sch name = DB0
#NET "MemDB<1>" LOC = "L4"; # Bank = 3, Pin name = IO_L16N_3, Type = I/O, Sch name = DB1
#NET "MemDB<2>" LOC = "L6"; # Bank = 3, Pin name = IO_L17P_3, Type = I/O, Sch name = DB2
#NET "MemDB<3>" LOC = "M4"; # Bank = 3, Pin name = IO_L18P_3, Type = I/O, Sch name = DB3
#NET "MemDB<4>" LOC = "N5"; # Bank = 3, Pin name = IO_L20N_3, Type = I/O, Sch name = DB4
#NET "MemDB<5>" LOC = "P1"; # Bank = 3, Pin name = IO_L21N_3, Type = I/O, Sch name = DB5
#NET "MemDB<6>" LOC = "P2"; # Bank = 3, Pin name = IO_L21P_3, Type = I/O, Sch name = DB6
#NET "MemDB<7>" LOC = "R2"; # Bank = 3, Pin name = IO_L23N_3, Type = I/O, Sch name = DB7
#NET "MemDB<8>" LOC = "L3"; # Bank = 3, Pin name = IO_L16P_3, Type = I/O, Sch name = DB8
#NET "MemDB<9>" LOC = "L5"; # Bank = 3, Pin name = IO_L17N_3/VREF_3, Type = VREF, Sch name = DB9
#NET "MemDB<10>" LOC = "M3"; # Bank = 3, Pin name = IO_L18N_3, Type = I/O, Sch name = DB10
#NET "MemDB<11>" LOC = "M6"; # Bank = 3, Pin name = IO_L19N_3, Type = I/O, Sch name = DB11
#NET "MemDB<12>" LOC = "L2"; # Bank = 3, Pin name = IO_L15N_3, Type = I/O, Sch name = DB12
#NET "MemDB<13>" LOC = "N4"; # Bank = 3, Pin name = IO_L20P_3, Type = I/O, Sch name = DB13
#NET "MemDB<14>" LOC = "R3"; # Bank = 3, Pin name = IO_L23P_3, Type = I/O, Sch name = DB14
#NET "MemDB<15>" LOC = "T1"; # Bank = 3, Pin name = IO_L24N_3, Type = I/O, Sch name = DB15
```

```
#
## 7 segment display
#NET "segments(0)" LOC = "L18"; # Bank = 1, Pin name = IO_L10P_1, Type = I/O, Sch name = CA
#NET "segments(1)" LOC = "F18"; # Bank = 1, Pin name = IO_L19P_1, Type = I/O, Sch name = CB
#NET "segments(2)" LOC = "D17"; # Bank = 1, Pin name = IO_L23P_1/HDC, Type = DUAL, Sch name = CC
#NET "segments(3)" LOC = "D16"; # Bank = 1, Pin name = IO_L23N_1/LDC0, Type = DUAL, Sch name = CD
#NET "segments(4)" LOC = "G14"; # Bank = 1, Pin name = IO_L20P_1, Type = I/O, Sch name = CE
#NET "segments(5)" LOC = "J17"; # Bank = 1, Pin name = IO_L13P_1/A6/RHCLK4/IRDY1, Type =
    RHCLK/DUAL, Sch name = CF
#NET "segments(6)" LOC = "H14"; # Bank = 1, Pin name = IO_L17P_1, Type = I/O, Sch name = CG
#NET "dp" LOC = "C17"; # Bank = 1, Pin name = IO_L24N_1/LDC2, Type = DUAL, Sch name = DP

#NET "anodes(0)" LOC = "F17"; # Bank = 1, Pin name = IO_L19N_1, Type = I/O, Sch name = AN0
#NET "anodes(1)" LOC = "H17"; # Bank = 1, Pin name = IO_L16N_1/A0, Type = DUAL, Sch name = AN1
#NET "anodes(2)" LOC = "C18"; # Bank = 1, Pin name = IO_L24P_1/LDC1, Type = DUAL, Sch name = AN2
#NET "anodes(3)" LOC = "F15"; # Bank = 1, Pin name = IO_L21P_1, Type = I/O, Sch name = AN3

## Leds
#NET "Led<0>" LOC = "J14"; # Bank = 1, Pin name = IO_L14N_1/A3/RHCLK7, Type = RHCLK/DUAL, Sch name
    = JD10/LD0
#NET "Led<1>" LOC = "J15"; # Bank = 1, Pin name = IO_L14P_1/A4/RHCLK6, Type = RHCLK/DUAL, Sch name
    = JD9/LD1
#NET "Led<2>" LOC = "K15"; # Bank = 1, Pin name = IO_L12P_1/A8/RHCLK2, Type = RHCLK/DUAL, Sch name
    = JD8/LD2
#NET "Led<3>" LOC = "K14"; # Bank = 1, Pin name = IO_L12N_1/A7/RHCLK3/TRDY1, Type = RHCLK/DUAL,
    Sch name = JD7/LD3
#NET "Led<4>" LOC = "E17"; # Bank = 1, Pin name = IO, Type = I/O, Sch name = LD4?
#NET "Led<5>" LOC = "P15"; # Bank = 1, Pin name = IO, Type = I/O, Sch name = LD5?
#NET "Led<6>" LOC = "F4"; # Bank = 3, Pin name = IO, Type = I/O, Sch name = LD6?
#NET "Led<7>" LOC = "R4"; # Bank = 3, Pin name = IO/VREF_3, Type = VREF, Sch name = LD7?
##NET "Led<4>" LOC = "E16"; # Bank = 1, Pin name = N.C., Type = N.C., Sch name = LD4?
##NET "Led<5>" LOC = "P16"; # Bank = 1, Pin name = N.C., Type = N.C., Sch name = LD5?
##NET "Led<6>" LOC = "E4"; # Bank = 3, Pin name = N.C., Type = N.C., Sch name = LD6?
##NET "Led<7>" LOC = "P4"; # Bank = 3, Pin name = N.C., Type = N.C., Sch name = LD7?
#
## Switches
#NET "sw<0>" LOC = "G18"; # Bank = 1, Pin name = IP, Type = INPUT, Sch name = SW0
#NET "sw<1>" LOC = "H18"; # Bank = 1, Pin name = IP/VREF_1, Type = VREF, Sch name = SW1
#NET "sw<2>" LOC = "K18"; # Bank = 1, Pin name = IP, Type = INPUT, Sch name = SW2
#NET "sw<3>" LOC = "K17"; # Bank = 1, Pin name = IP, Type = INPUT, Sch name = SW3
#NET "sw<4>" LOC = "L14"; # Bank = 1, Pin name = IP, Type = INPUT, Sch name = SW4
#NET "sw<5>" LOC = "L13"; # Bank = 1, Pin name = IP, Type = INPUT, Sch name = SW5
#NET "sw<6>" LOC = "N17"; # Bank = 1, Pin name = IP, Type = INPUT, Sch name = SW6
#NET "sw<7>" LOC = "R17"; # Bank = 1, Pin name = IP, Type = INPUT, Sch name = SW7
#
# Buttons
#
NET "reset" LOC = "B18"; # Bank = 1, Pin name = IP, Type = INPUT, Sch name = BTN0
#NET "reset" LOC = "D18"; # Bank = 1, Pin name = IP/VREF_1, Type = VREF, Sch name = BTN1
#NET "up" LOC = "E18"; # Bank = 1, Pin name = IP, Type = INPUT, Sch name = BTN2
#NET "btn<3>" LOC = "H13"; # Bank = 1, Pin name = IP, Type = INPUT, Sch name = BTN3
#
# VGA Connector
NET "color(7)" LOC = "R9"; # Bank = 2, Pin name = IO/D5, Type = DUAL, Sch name = RED0
NET "color(6)" LOC = "T8"; # Bank = 2, Pin name = IO_L10N_2, Type = I/O, Sch name = RED1
NET "color(5)" LOC = "R8"; # Bank = 2, Pin name = IO_L10P_2, Type = I/O, Sch name = RED2
NET "color(4)" LOC = "N8"; # Bank = 2, Pin name = IO_L09N_2, Type = I/O, Sch name = GRN0
NET "color(3)" LOC = "P8"; # Bank = 2, Pin name = IO_L09P_2, Type = I/O, Sch name = GRN1
NET "color(2)" LOC = "P6"; # Bank = 2, Pin name = IO_L05N_2, Type = I/O, Sch name = GRN2
```

```
NET "color(1)" LOC = "U5"; # Bank = 2, Pin name = IO/VREF_2, Type = VREF, Sch name = BLU1
NET "color(0)" LOC = "U4"; # Bank = 2, Pin name = IO_L03P_2/DOUT/BUSY, Type = DUAL, Sch name = BLU2

NET "hsync" LOC = "T4"; # Bank = 2, Pin name = IO_L03N_2/MOSI/CSI_B, Type = DUAL, Sch name = HSYNC
NET "vsync" LOC = "U3"; # Bank = 2, Pin name = IO_L01P_2/CSO_B, Type = DUAL, Sch name = VSYNC

# PS/2 connector
#NET "PS2C" LOC = "R12"; # Bank = 2, Pin name = IO_L20N_2, Type = I/O, Sch name = PS2C
#NET "PS2D" LOC = "P11"; # Bank = 2, Pin name = IO_L18P_2, Type = I/O, Sch name = PS2D

# FX2 connector
#NET "PIO<0>" LOC = "B4"; # Bank = 0, Pin name = IO_L24N_0, Type = I/O, Sch name = R-IO1
#NET "PIO<1>" LOC = "A4"; # Bank = 0, Pin name = IO_L24P_0, Type = I/O, Sch name = R-IO2
#NET "PIO<2>" LOC = "C3"; # Bank = 0, Pin name = IO_L25P_0, Type = I/O, Sch name = R-IO3
#NET "PIO<3>" LOC = "C4"; # Bank = 0, Pin name = IO, Type = I/O, Sch name = R-IO4
#NET "PIO<4>" LOC = "B6"; # Bank = 0, Pin name = IO_L20P_0, Type = I/O, Sch name = R-IO5
#NET "PIO<5>" LOC = "D5"; # Bank = 0, Pin name = IO_L23N_0/VREF_0, Type = VREF, Sch name = R-IO6
#NET "PIO<6>" LOC = "C5"; # Bank = 0, Pin name = IO_L23P_0, Type = I/O, Sch name = R-IO7
#NET "PIO<7>" LOC = "F7"; # Bank = 0, Pin name = IO_L19P_0, Type = I/O, Sch name = R-IO8
#NET "PIO<8>" LOC = "E7"; # Bank = 0, Pin name = IO_L19N_0/VREF_0, Type = VREF, Sch name = R-IO9
#NET "PIO<9>" LOC = "A6"; # Bank = 0, Pin name = IO_L20N_0, Type = I/O, Sch name = R-IO10
#NET "PIO<10>" LOC = "C7"; # Bank = 0, Pin name = IO_L18P_0, Type = I/O, Sch name = R-IO11
#NET "PIO<11>" LOC = "F8"; # Bank = 0, Pin name = IO_L17N_0, Type = I/O, Sch name = R-IO12
#NET "PIO<12>" LOC = "D7"; # Bank = 0, Pin name = IO_L18N_0/VREF_0, Type = VREF, Sch name = R-IO13
#NET "PIO<13>" LOC = "E8"; # Bank = 0, Pin name = IO_L17P_0, Type = I/O, Sch name = R-IO14
#NET "PIO<14>" LOC = "E9"; # Bank = 0, Pin name = IO_L15P_0, Type = I/O, Sch name = R-IO15
#NET "PIO<15>" LOC = "C9"; # Bank = 0, Pin name = IO_L14P_0/GCLK10, Type = GCLK, Sch name = R-IO16
#NET "PIO<16>" LOC = "A8"; # Bank = 0, Pin name = IO, Type = I/O, Sch name = R-IO17
#NET "PIO<17>" LOC = "G9"; # Bank = 0, Pin name = IO, Type = I/O, Sch name = R-IO18
#NET "PIO<18>" LOC = "F9"; # Bank = 0, Pin name = IO_L15N_0, Type = I/O, Sch name = R-IO19
#NET "PIO<19>" LOC = "D10"; # Bank = 0, Pin name = IO_L11P_0/GCLK4, Type = GCLK, Sch name = R-IO20
#NET "PIO<20>" LOC = "A10"; # Bank = 0, Pin name = IO_L12N_0/GCLK7, Type = GCLK, Sch name = R-IO21
#NET "PIO<21>" LOC = "B10"; # Bank = 0, Pin name = IO_L12P_0/GCLK6, Type = GCLK, Sch name = R-IO22
#NET "PIO<22>" LOC = "A11"; # Bank = 0, Pin name = IO, Type = I/O, Sch name = R-IO23
#NET "PIO<23>" LOC = "D11"; # Bank = 0, Pin name = IO_L09N_0, Type = I/O, Sch name = R-IO24
#NET "PIO<24>" LOC = "E10"; # Bank = 0, Pin name = IO_L11N_0/GCLK5, Type = GCLK, Sch name = R-IO25
#NET "PIO<25>" LOC = "B11"; # Bank = 0, Pin name = IO/VREF_0, Type = VREF, Sch name = R-IO26
#NET "PIO<26>" LOC = "C11"; # Bank = 0, Pin name = IO_L09P_0, Type = I/O, Sch name = R-IO27
#NET "PIO<27>" LOC = "E11"; # Bank = 0, Pin name = IO_L08P_0, Type = I/O, Sch name = R-IO28
#NET "PIO<28>" LOC = "F11"; # Bank = 0, Pin name = IO_L08N_0, Type = I/O, Sch name = R-IO29
#NET "PIO<29>" LOC = "E12"; # Bank = 0, Pin name = IO_L06N_0, Type = I/O, Sch name = R-IO30
#NET "PIO<30>" LOC = "F12"; # Bank = 0, Pin name = IO_L06P_0, Type = I/O, Sch name = R-IO31
#NET "PIO<31>" LOC = "A13"; # Bank = 0, Pin name = IO_L05P_0, Type = I/O, Sch name = R-IO32
#NET "PIO<32>" LOC = "B13"; # Bank = 0, Pin name = IO_L05N_0/VREF_0, Type = VREF, Sch name = R-IO33
#NET "PIO<33>" LOC = "E13"; # Bank = 0, Pin name = IO, Type = I/O, Sch name = R-IO34
#NET "PIO<34>" LOC = "A14"; # Bank = 0, Pin name = IO_L04N_0, Type = I/O, Sch name = R-IO35
#NET "PIO<35>" LOC = "C14"; # Bank = 0, Pin name = IO_L03N_0/VREF_0, Type = VREF, Sch name = R-IO36
#NET "PIO<36>" LOC = "D14"; # Bank = 0, Pin name = IO_L03P_0, Type = I/O, Sch name = R-IO37
#NET "PIO<37>" LOC = "B14"; # Bank = 0, Pin name = IO_L04P_0, Type = I/O, Sch name = R-IO38
#NET "PIO<38>" LOC = "A16"; # Bank = 0, Pin name = IO_L01N_0, Type = I/O, Sch name = R-IO39
#NET "PIO<39>" LOC = "B16"; # Bank = 0, Pin name = IO_L01P_0, Type = I/O, Sch name = R-IO40
#
# 12 pin connectors
#NET "up" LOC = "L15"; # Bank = 1, Pin name = IO_L09N_1/A11, Type = DUAL, Sch name = JA1
#NET "right" LOC = "K12"; # Bank = 1, Pin name = IO_L11N_1/A9/RHCLK1, Type = RHCLK/DUAL, Sch name
    = JA2
#NET "left" LOC = "L17"; # Bank = 1, Pin name = IO_L10N_1/VREF_1, Type = VREF, Sch name = JA3
#NET "down" LOC = "M15"; # Bank = 1, Pin name = IO_L07P_1, Type = I/O, Sch name = JA4
```

```
#NET "JA<4>" LOC = "K13"; # Bank = 1, Pin name = IO_L11P_1/A10/RHCLK0, Type = RHCLK/DUAL, Sch name
    = JA7
#NET "JA<5>" LOC = "L16"; # Bank = 1, Pin name = IO_L09P_1/A12, Type = DUAL, Sch name = JA8
#NET "JA<6>" LOC = "M14"; # Bank = 1, Pin name = IO_L05P_1, Type = I/O, Sch name = JA9
#NET "JA<7>" LOC = "M16"; # Bank = 1, Pin name = IO_L07N_1, Type = I/O, Sch name = JA10

NET "up_two" LOC = "M13"; # Bank = 1, Pin name = IO_L05N_1/VREF_1, Type = VREF, Sch name = JB1
NET "right_two" LOC = "R18"; # Bank = 1, Pin name = IO_L02P_1/A14, Type = DUAL, Sch name = JB2
NET "left_two" LOC = "R15"; # Bank = 1, Pin name = IO_L03P_1, Type = I/O, Sch name = JB3
NET "down_two" LOC = "T17"; # Bank = 1, Pin name = IO_L01N_1/A15, Type = DUAL, Sch name = JB4
#NET "JB<4>" LOC = "P17"; # Bank = 1, Pin name = IO_L06P_1, Type = I/O, Sch name = JB7
#NET "JB<5>" LOC = "R16"; # Bank = 1, Pin name = IO_L03N_1/VREF_1, Type = VREF, Sch name = JB8
#NET "JB<6>" LOC = "T18"; # Bank = 1, Pin name = IO_L02N_1/A13, Type = DUAL, Sch name = JB9
#NET "JB<7>" LOC = "U18"; # Bank = 1, Pin name = IO_L01P_1/A16, Type = DUAL, Sch name = JB10

NET "up_one" LOC = "G15"; # Bank = 1, Pin name = IO_L18P_1, Type = I/O, Sch name = JC1
NET "right_one" LOC = "J16"; # Bank = 1, Pin name = IO_L13N_1/A5/RHCLK5, Type = RHCLK/DUAL, Sch
    name = JC2
NET "left_one" LOC = "G13"; # Bank = 1, Pin name = IO_L20N_1, Type = I/O, Sch name = JC3
NET "down_one" LOC = "H16"; # Bank = 1, Pin name = IO_L16P_1, Type = I/O, Sch name = JC4
#NET "JC<4>" LOC = "H15"; # Bank = 1, Pin name = IO_L17N_1, Type = I/O, Sch name = JC7
#NET "JC<5>" LOC = "F14"; # Bank = 1, Pin name = IO_L21N_1, Type = I/O, Sch name = JC8
#NET "JC<6>" LOC = "G16"; # Bank = 1, Pin name = IO_L18N_1, Type = I/O, Sch name = JC9
#NET "JC<7>" LOC = "J12"; # Bank = 1, Pin name = IO_L15P_1/A2, Type = DUAL, Sch name = JC10

#NET "JD<0>" LOC = "J13"; # Bank = 1, Pin name = IO_L15N_1/A1, Type = DUAL, Sch name = JD1
#NET "JD<1>" LOC = "M18"; # Bank = 1, Pin name = IO_L08N_1, Type = I/O, Sch name = JD2
#NET "JD<2>" LOC = "N18"; # Bank = 1, Pin name = IO_L08P_1, Type = I/O, Sch name = JD3
#NET "JD<3>" LOC = "P18"; # Bank = 1, Pin name = IO_L06N_1, Type = I/O, Sch name = JD4

# RS232 connector
#NET "RsRx" LOC = "U6"; # Bank = 2, Pin name = IP, Type = INPUT, Sch name = RS-RX
#NET "RsTx" LOC = "P9"; # Bank = 2, Pin name = IO, Type = I/O, Sch name = RS-TX
```

### 7.2.3 Resource Utilization

Table 2 presents the utilization statistics of our design while Table 3 shows the quantity of macro units in our design.

| | | | | |
|---|---|---|---|---|
| Number of Slices: | 226 | out of | 4656 | 4% |
| Number of Slice Flip Flops: | 249 | out of | 9312 | 2% |
| Number of 4 input LUTs: | 428 | out of | 9312 | 4% |
| Number of IOs: | 20 | | | |
| Number of bonded IOBs: | 20 | out of | 232 | 8% |
| Number of BRAMs: | 16 | out of | 20 | 8-% |
| Number of GCLKs: | 2 | out of | 24 | 8% |

Table 2: Device Utilization Table

| **Macros** | **Quantity** |
|---|---|
| **FSMs** | 4 |
| **RAMs (131072x2-bit dual-port block RAM)** | 1 |
| **ROMs (4x8-bit ROM)** | 1 |
| **Counters** | 6 |
| 10-bit up counter | 4 |
| 2-bit up counter | 1 |
| 32-bit up counter | 1 |
| **Accumulators (10-bit updown accumulator)** | 4 |
| **Registers (Flip-Flops)** | 95 |
| **Comparators** | 19 |
| 10-bit comparator greater | 4 |
| 10-bit comparator less | 15 |
| **Xors** | 9 |

Table 3: Advanced HDL Synthesis Report - Macro Statistic Table

### 7.2.4 Critical Timing Path

The time delay of the critical path (Memory/Mram_myRAM1 to Memory/Mram_myRAM2)is 11.886 ns. It goes through 7 levels of logic and happens in the path starting from reading the MemoryRAM to determine collision for player 2 and ending with updating the MemoryRAM with the color of either player 2 or the collision color. The logic delay accounts for 70.8% of the critical time delay and the wire delay accounts for the remaining 29.2% .

The table below shows the detailed critical path information based on the synthesis report:

| **Cell:in-out** | **Fanout** | **Gate Delay** | **Net Delay** | **Logical Name** |
|---|---|---|---|---|
| RAMB16_S1_S1:CLKB-DOB0 | 1 | 2.800 | 0.499 | Memory/Mram_myRAM1 (N11) |
| LUT3:I1-O | 1 | 0.704 | 0.000 | inst_LPM_MUX_6 (inst_LPM_MUX_6) |
| MUXF5:I0-O | 1 | 0.321 | 0.000 | inst_LPM_MUX_4_f5 (inst_LPM_MUX_4_f5) |
| MUXF6:I0-O | 7 | 0.521 | 0.743 | inst_LPM_MUX_2_f6 (Memory/d_out0) |
| LUT4_D:I2-O | 3 | 0.704 | 0.535 | Game/collide_two1 (Game/collide_two) |
| LUT4:I3-O | 3 | 0.704 | 0.535 | Game/curr_game_state_FSM_FFd8-In11 |
| LUT4_D:I3-O | 7 | 0.704 | 0.712 | Game/wr_en24 (wr_en) |
| LUT4:I3-O | 2 | 0.704 | 0.447 | write_ctrl1 (write_ctrl1) |
| RAMB16_S1_S1:WEA | | 1.253 | | Memory/Mram_myRAM2 |
| **Total** | | **11.886ns** | | |

Table 4: Detailed timing break down for critical path

### 7.2.5 Analysis of Residual Warnings

Listing 7: Residual Warnings left over from synthesis

```
Xst:647 - Input <pixel_x_r<0>> is never used. This port will be preserved and left unconnected if
    it belongs to a top-level block or it belongs to a sub-block and the hierarchy of this
    sub-block is preserved.
Xst:647 - Input <pixel_x_w<0>> is never used. This port will be preserved and left unconnected if
    it belongs to a top-level block or it belongs to a sub-block and the hierarchy of this
    sub-block is preserved.
Xst:647 - Input <pixel_y_r<9>> is never used. This port will be preserved and left unconnected if
    it belongs to a top-level block or it belongs to a sub-block and the hierarchy of this
    sub-block is preserved.
Xst:647 - Input <pixel_y_r<0>> is never used. This port will be preserved and left unconnected if
    it belongs to a top-level block or it belongs to a sub-block and the hierarchy of this
    sub-block is preserved.
Xst:647 - Input <pixel_y_w<9>> is never used. This port will be preserved and left unconnected if
    it belongs to a top-level block or it belongs to a sub-block and the hierarchy of this
    sub-block is preserved.
Xst:647 - Input <pixel_y_w<0>> is never used. This port will be preserved and left unconnected if
    it belongs to a top-level block or it belongs to a sub-block and the hierarchy of this
    sub-block is preserved.
Xst:647 - Input <h_video_on> is never used. This port will be preserved and left unconnected if it
    belongs to a top-level block or it belongs to a sub-block and the hierarchy of this sub-block
    is preserved.
```

All the warnings above can be ignored. The least significant bit of the vector bus for pixel_x_r and pixel_x_w is not used because of the rescaling of the pixel size (grouping 4 pixels as one pixel) so that we can go from 640 x 480 to 320 x 240 pixels. The least significant bit and the most significant bit of the vector bus for pixel_y_r and pixel_y_w are not used for the same reason.

We did not use h_video_on in our game controller even though it is created in the VGA controller. This is because we only write to memory when v_video_on = 0 and we did not write to memory in the short time span in which h_video_on = 0. We noted that it is a convention for the VGA controller to have a h_video_on output and it allows for portability of the VGA controller code to future projects that might require updating during the time in which h_video_on = 0.

## 7.3 Waveform graphs

Our group did not make use of the ISIM simulator to debug our code since it takes too long to render (over a couple hundred cycles of the screen refreshes) and does not present us with real time data. Instead, we got the VGA controller to work first and depended a lot on the VGA to debug our code. The details of how we debugged using the VGA monitor is presented in Section 2.4 of the main report.

We also attached the code that we used to test the VGA controller (VGA test pattern) and the MemoryRAM (Memory Test pattern). Screenshots of the test pattern are also included in this section.

**a) Screenshot of the VGA Controller Test Pattern**

Fig. 17: Screenshot of the VGA Controller Test Pattern

### b) Screenshot of Memory Test Pattern

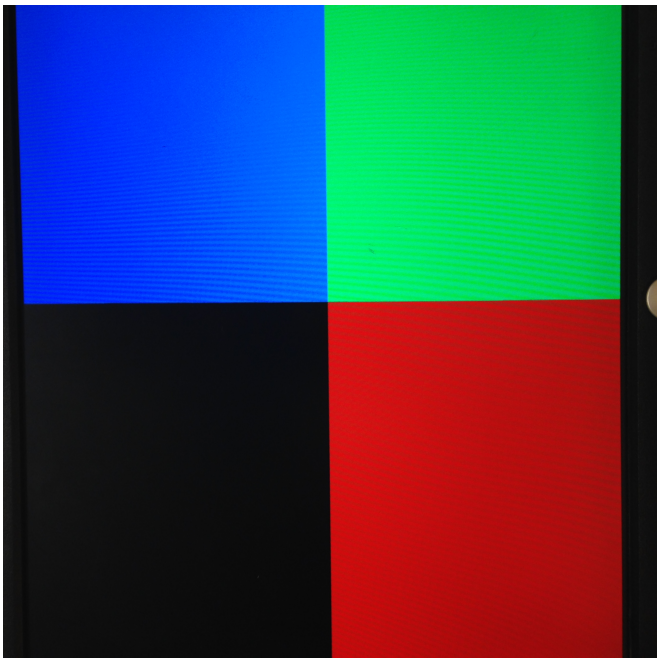

Fig. 18: Screenshot of Memory Test Pattern

### c) VGA Controller Test Pattern

Listing 8: VGA Controller Test Pattern

```
----------------------------------------------------------------------------------
-- Engineer:          J. Graham Keggi
--
-- Create Date:       15:10:36 07/12/2010
-- Module Name:       vga_test_pattern - Behavioral
-- Target Device:     Spartan3E-500 Nexys2
--
-- Description: Reads in current pixel X and Y as 2 10-bit vectors and supplys
--                    an 8-bit color code consistent with the VGA test pattern
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity vga_test_pattern is
       port(row,column              : in std_logic_vector(9 downto 0);
             color                  : out std_logic_vector(7 downto 0));
end vga_test_pattern;

architecture Behavioral of vga_test_pattern is

       -- Predefined 8-bit colors that nearly match real test pattern colors
       constant RED          : std_logic_vector(7 downto 0) := "11100000";
```

```
        constant GREEN       : std_logic_vector(7 downto 0) := "00011100";
        constant BLUE        : std_logic_vector(7 downto 0) := "00000011";
        constant CYAN        : std_logic_vector(7 downto 0) := "00011111";
        constant YELLOW      : std_logic_vector(7 downto 0) := "11111100";
        constant PURPLE      : std_logic_vector(7 downto 0) := "11100011";
        constant WHITE       : std_logic_vector(7 downto 0) := "11011011";
        constant BLACK       : std_logic_vector(7 downto 0) := "00000000";
        constant GRAY0       : std_logic_vector(7 downto 0) := "01001001";
        constant GRAY1       : std_logic_vector(7 downto 0) := "10010010";
        constant DARK_BLU    : std_logic_vector(7 downto 0) := "00001010";
        constant DARK_PUR    : std_logic_vector(7 downto 0) := "01000010";

begin

        -- set the colors based on the current pixel
        process(row,column)
        begin
                -- large vertical color bands, evenly spaced horizontally, 320px vertically
                -- Gray, yellow, cyan, green, purple, red, blue
                if (column >= 0) and (column < 92) and (row >= 0) and (row < 320) then
                        color <= GRAY1;
                elsif (column >= 92) and (column < 184) and (row >= 0) and (row < 320) then
                        color <= YELLOW;
                elsif (column >= 184) and (column < 276) and (row >= 0) and (row < 320) then
                        color <= CYAN;
                elsif (column >= 276) and (column < 368) and (row >= 0) and (row < 320) then
                        color <= GREEN;
                elsif (column >= 368) and (column < 460) and (row >= 0) and (row < 320) then
                        color <= PURPLE;
                elsif (column >= 460) and (column < 552) and (row >= 0) and (row < 320) then
                        color <= RED;
                elsif (column >= 552) and (column <= 640) and (row >= 0) and (row < 320) then
                        color <= BLUE;

                -- small colored rectangles from 320->360 pixels, evenly spaced horizontally
                -- blue, black, purple, black, cyan, black, gray
                elsif (column >= 0) and (column < 92) and (row >= 320) and (row < 360) then
                        color <= BLUE;
                elsif (column >= 92) and (column < 184) and (row >= 320) and (row < 360) then
                        color <= BLACK;
                elsif (column >= 184) and (column < 276) and (row >= 320) and (row < 360) then
                        color <= PURPLE;
                elsif (column >= 276) and (column < 368) and (row >= 320) and (row < 360) then
                        color <= BLACK;
                elsif (column >= 368) and (column < 460) and (row >= 320) and (row < 360) then
                        color <= CYAN;
                elsif (column >= 460) and (column < 552) and (row >= 320) and (row < 360) then
                        color <= BLACK;
                elsif (column >= 552) and (column <= 640) and (row >= 320) and (row < 360) then
                        color <= GRAY1;

                -- bottom large blocks
                elsif (column >= 0) and (column < 114) and (row >= 360) and (row <= 480) then
                        color <= DARK_BLU;
                elsif (column >=114) and (column < 228) and (row >= 360) and (row <= 480) then
                        color <= WHITE;
                elsif (column >= 228) and (column < 342) and (row >= 360) and (row <= 480) then
                        color <= DARK_PUR;
                elsif (column >= 342) and (column < 460) and (row >= 360) and (row <= 480) then
```

```
                color <= GRAY0;
            elsif (column >= 460) and (column < 491) and (row >= 360) and (row <= 480) then
                color <= BLACK;
            elsif (column >= 491) and (column < 521) and (row >= 360) and (row <= 480) then
                color <= GRAY0;
            elsif (column >= 521) and (column < 552) and (row >= 360) and (row <= 480) then
                color <= GRAY1;
            elsif (column >= 552) and (column <= 640) and (row >= 360) and (row <= 480) then
                color <= GRAY0;

            -- black for any gaps, shouldn't be any
            else
                color <= BLACK;
            end if;
        end process;

end Behavioral;
```

### d) Memory RAM Test Pattern

Listing 9: Memory RAM Test Pattern

```
----------------------------------------------------------------------------------
-- Company:            Engs 31 15X
-- Engineer:           Edrei Chua and Jonathan Huang
--
-- Create Date:        10:32:22 08/14/2015
-- Design Name:        Tron
-- Module Name:        memory_test_pattern - Behavioral
-- Project Name:       Tron
-- Target Devices:     Spartan 3E
-- Tool versions:      ISE 14.7
-- Description:        Test pattern to test the MemoryRAM and VGAController
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
library UNISIM;
use UNISIM.VComponents.all;

entity memory_test_pattern is
    Port ( clk : in STD_LOGIC;
                        h_video_on : in STD_LOGIC;
```

```
                        v_video_on : in STD_LOGIC;
                        pixel_x_w : out STD_LOGIC_VECTOR (9 downto 0);
          pixel_y_w : out STD_LOGIC_VECTOR (9 downto 0);
                        wr_en : out STD_LOGIC;
          color : out STD_LOGIC_VECTOR (1 downto 0));
end memory_test_pattern;

architecture Behavioral of memory_test_pattern is
-- internal signals
       signal iQ_x : unsigned(9 downto 0) := (others => '0');
       signal iQ_y : unsigned(9 downto 0) := (others => '0');

begin
-- Test Pattern
Pattern: process(iQ_x, iQ_y)
begin
       if (iQ_x >= 0) and (iQ_x < 320) and (iQ_y >= 0) and (iQ_y < 240) then -- top left is black
              color <= "00";
       elsif (iQ_x >= 320) and (iQ_x < 640) and (iQ_y >= 0) and (iQ_y < 240) then -- top right is
           blue
              color <= "01";
       elsif (iQ_x >= 320) and (iQ_x < 640) and (iQ_y >= 240) and (iQ_y < 480) then -- bottom
           right is green
              color <= "10";
       elsif (iQ_x >= 0) and (iQ_x <320) and (iQ_y >= 240) and (iQ_y < 480) then -- bottom left is
           red
              color <= "11";
       else
              color <= "00";
       end if;

end process Pattern;


-- Counter
Counter: process(clk)
begin
       if rising_edge(clk) then
              -- only write when video is not on
              if h_video_on = '0' or v_video_on = '0' then
                     -- Horizontal Counter
                     if iQ_x < 639 then                                    -- increment
                            iQ_x <= iQ_x + 1;
                     else
                            iQ_x <= (others => '0');                       -- reset
                            -- Vertical Counter
                            if iQ_y < 479 then                             -- increment
                                   iQ_y <= iQ_y + 1;
                            else                                           -- reset
                                   iQ_y <= (others => '0');
                            end if;
                     end if;
              end if;
       end if;
end process Counter;

-- convert back to std_logic_vector
pixel_x_w <= std_logic_vector(iQ_x);
pixel_y_w <= std_logic_vector(iQ_y);
```

```
-- Write when v_video_on is off
wr_en <= NOT(v_video_on);

end Behavioral;
```