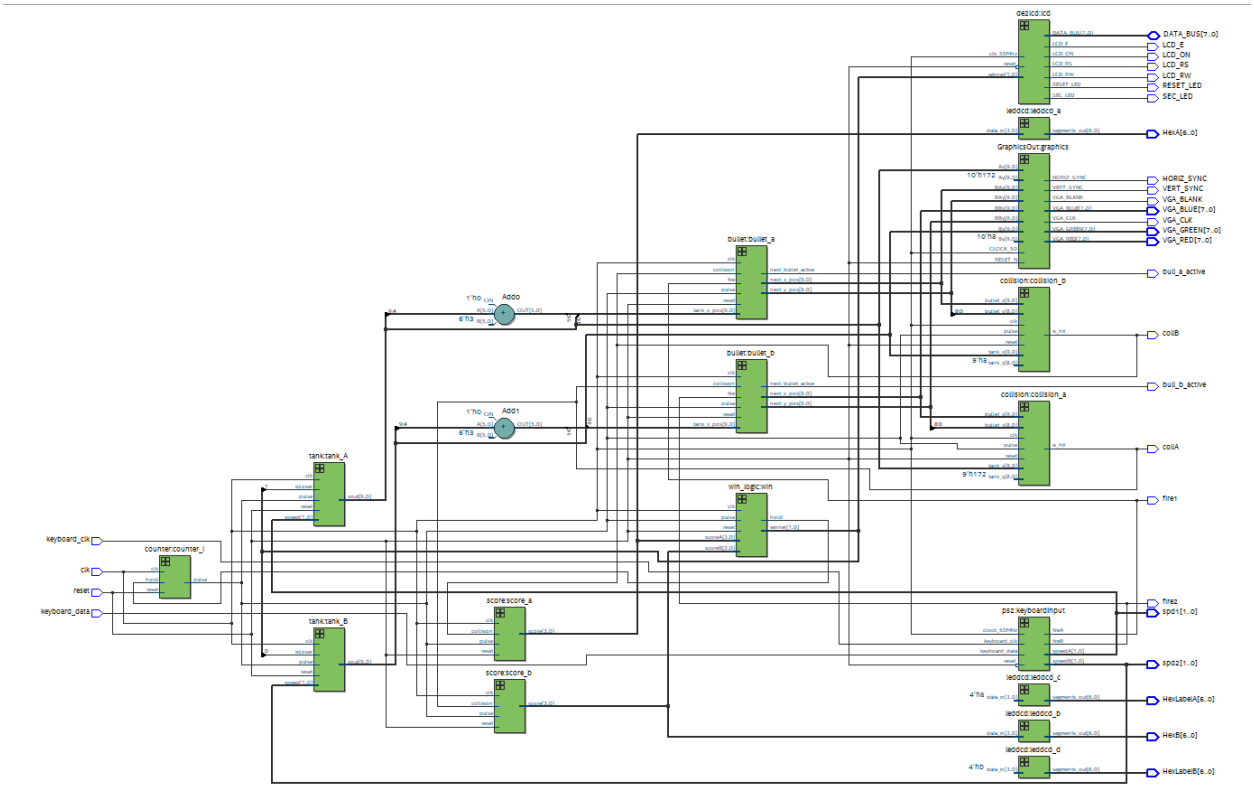
EECS 355 Final Project
TANK GAME

Graehme Blair, Kushal Gourikrishna, Drew Hasse, Will Wallace

Game Overview:

The game consists of two tank components (one for each player) and each tank entity is connected to its own respective bullet component. The basic behavior of the tank is that it moves back and forth on the screen horizontally. When the bullet component of a certain tank receives a signal from the correct keyboard key from the ps2 component the bullet is fired. The bullet fires from the current x position of its respective tank and travels vertically on the screen. Each tank and bullet also has its own collision component and score component. The collision component detects if the bullet has collided with the opposing tank. It outputs a one bit signal that is then fed into its respective score component. If the collide signal is high the score component adds 1 to the current score. Lastly, there is a win_logic component. This component simply detects whether the score of either tank has reached 3. If so the game is ended and the tank that lost disappears from the screen.

RTL View:



COMPONENTS:

tank_game:
Entity:
Desired Behavior:

Take in clock, reset and keyboard signals and output the necessary control and data signals to operate the LCD, the monitor via VGA and the HEX LEDs.

Implementation:

Subcomponents Used:

Counter, PS2, tank, bullet, collision, score, win_logic, de2lcd, GraphicsOut. See component descriptions of each of these for more information.

The tank_game component instantiates one counter, one PS2 component, two tank components, two bullet components, two collision components, two score components, one win_logic component, one de2lcd component and one GraphicsOut component. The components are connected internally in the following fashion:

Source of signal (signal name) => Destination of signal
Counter(pulse) => all components except keyboard and GraphicsOut.
PS2(speedA) => tankA
PS2(fireA) => bulletA
PS2(speedB) => tankB
PS2(fireB) => bulletB
tankA(xout) => bulletA, collisionA, GraphicsOut
tankB(xout) => bulletB, collisionB, GraphicsOut
bulletA(next_x_pos) => collisionB, GraphicsOut
bulletA(next_y_pos) => collisionB, GraphicsOut
bulletB(next_x_pos) => collisionA, GraphicsOut
bulletB(next_y_pos) => collisionA, GraphicsOut
collisionA(is_hit) => bulletB, scoreB
collisionB(is_hit) => bulletA, scoreA
scoreA(score) => win_logic
scoreB(score) => win_logic
Win_logic(hold) => counter
win_logic(winner) => tankA, tankB, de2lcd
De2lcd and GraphicsOut are purely periphal output

These connections provide all of the games fundamental components the signals they need to operate.

Simulation:

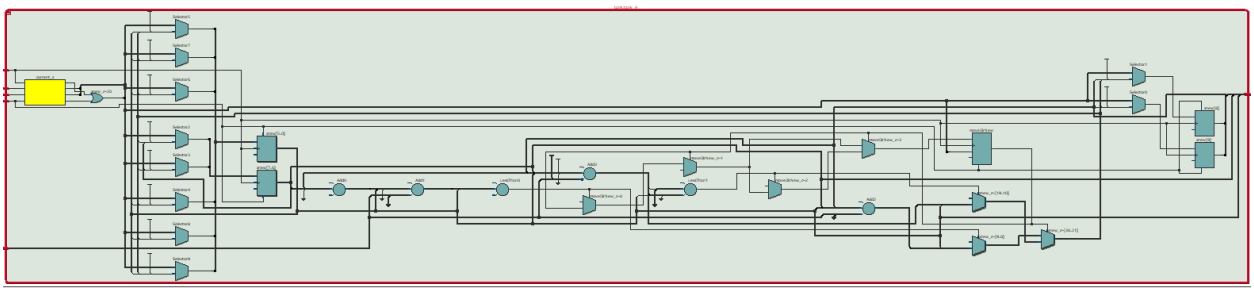
No simulation was made for this component as it functions purely in conjunction with the development board.

Tank:

As described in the project description, the tanks move horizontally along the screen at three different speeds controlled by the keyboard. When the edge of the tank reaches an edge of the screen, the tanks bounce back. The inputs for the tank entity are: clk, reset, pulse, speed, isLoser. The one output is the new x position, xout.

```
entity
tank
is
        generic (
            width : std_logic_vector(5 downto 0)
        );
        port (
            clk : in std_logic;
            reset : in std_logic;
            pulse : in std_logic;
            speed : in std_logic_vector(1 downto 0);
            isLoser : in std_logic;
            xout : out std_logic_vector(9 downto 0)
            --moveDirNew : out std_logic
        );
    end entity;
```

RTL View:



The tank movement was implemented with a finite state machine that consisted of four states: idle, update, lost, and waitOnPulseLow. The idle state, as its name implies, does nothing until a pulse is detected, in which case it moves to the update state. The tank waits for a pulse rather

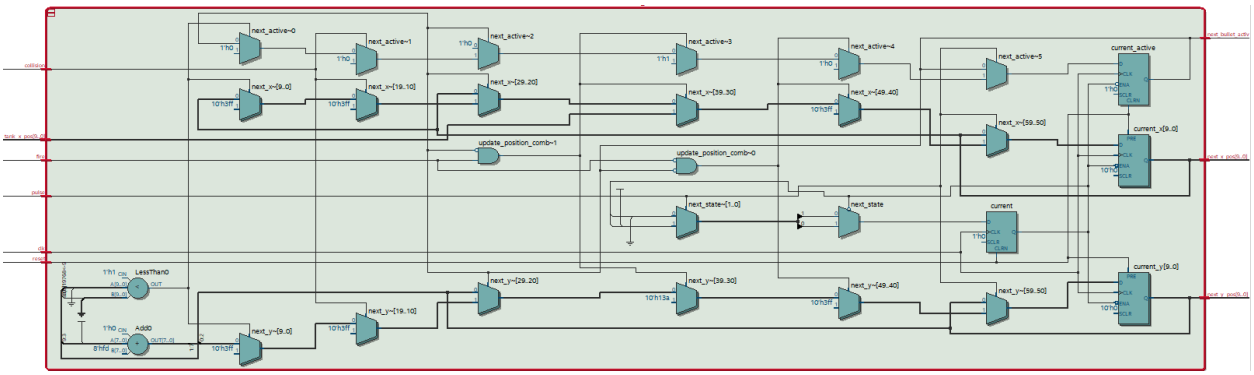
than a clock because the VGA monitor has a much slower frequency than the FPGA's clock, so we want the movement to be updated in the VGA monitor's time domain, so that the tank moves at the proper speed. In the update state, the tank's position is updated based on the speed. The tank has an internal move direction that is 0 if the tank is moving in the positive x direction, and 1 if the tank is moving in the negative x direction. Based on this move direction, the tank either adds or subtracts the speed times some speed factor set in tank_pack.vhd to its position and outputs this as xout. The width generic, is used to check if the tank is going to hit one of the edges. In the case that the tank does hit an edge, the move direction is inverted, and the tank then moves in the opposite direction to achieve the bouncing effect. From the update state the, tanks either goes to the waitOnPulseLow or the lost state. The tank will advance to the lost state if the isLoser input is 1. In this state the tank position is set so that only the winning tank is drawn on the VGA monitor. Otherwise, if isLoser is 0, the tank moves from the update state to the waitOnPulseLow state. This state is necessary because we want the tank to only update once per pulse, and since the pulse remains high for longer than one clock cycle, sending the tank to back to the idle state before pulse is low would cause the tank position to update multiple times. In the waitOnPulseLow state, the next state is always waitOnPulseLow, unless the pulse input is 0, in which case next state is idle.

Bullet:

The entity for the bullet component is described below:

```
entity
bullet
is
        generic (
                tank_y_pos : std_logic_vector (9 downto 0);
                bullet_direction : std_logic
        );
        port (
                clk : in std_logic;
                reset : in std_logic;
                pulse : in std_logic;
                fire : in std_logic;
                tank_x_pos : in std_logic_vector (9 downto 0);
                collision : in std_logic;
                next_bullet_active : out std_logic;
                next_y_pos : out std_logic_vector (9 downto 0);
                next_x_pos : out std_logic_vector (9 downto 0)
        );
        end entity bullet;
```

RTL View:



Like with every component, the bullet takes a clock, reset, and a pulse input. It also takes an input signal called “fire” which is a signal coming from the keyboard indicating when a bullet should be shot, the x-position of the tank, indicating where the bullet should originate when shot, and a collision input indicating if the bullet has collided with the opposing tank and should be

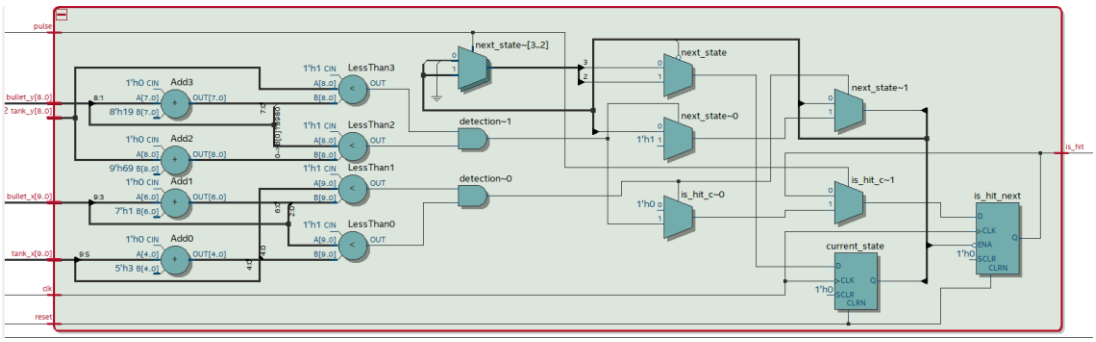
eliminated from the screen. The bullet component outputs whether it is active or not and its next x and y position after each pulse. The basic structure of the component is a simple state machine. In the clocked process, the current x, y, and active signals are updated with their corresponding next signals. The combinational process is made up of two states: the idle state and waitOnPulseLow state. The idle state handles all of the logic. In this state it is first checked whether a bullet has been fired, if so a bullet is fired from the tank's current x position. If a bullet is active the bullet position is updated. Finally, if the bullet has not been fired and it is not active it stays off screen. After these initial checks, the bullet direction is verified in order to update the direction properly and then the collision bit is checked to see if the bullet should remain active or disappear. The waitOnPulseLow state simply ensures that the idle state doesn't keep occurring on each clock cycle by sampling waiting until the overall pulse signal goes low.

Collision:

The collision entity detects when a bullet fired from one tank intersects with the other tank. Two collision entities are instantiated, one for each tank. The inputs are tank_x and tank_y, (the top-left x and y coordinates of a tank), bullet_x and bullet_y (the center of the bullet), clk, reset, and pulse. The one output is a one-bit flag is_hit. The bullet is much smaller than the tank and collision is detected just on a single point.

```
entity
collision
is
        port (
                tank_x : in std_logic_vector(9 downto 0);
                tank_y : in std_logic_vector (8 downto 0);
                bullet_x : in std_logic_vector(9 downto 0);
                bullet_y : in std_logic_vector(8 downto 0);
                clk : in std_logic;
                reset : in std_logic;
                pulse : in std_logic;
                is_hit : out std_logic
        );
        end entity;
```

RTL View



Calculating collision was implemented with a finite state machine that consisted of two states: idle and waitOnPulseLow. The idle state does nothing unless a pulse is detected at which point it takes in the current tank and bullet positions and checks for a collision. A collision is detected simply by comparing the x-positions of the bullet and tank to see if the bullet's x-position is within the tank's x-position. Both sides of the tank must be checked so the constant TANK_WIDTH from the tank package is used. If the bullet currently is within the left and right edge of the tank, the same comparison is done for the y-positions (using the constant TANK_HEIGHT). If there is overlap in the y-positions, then the is_hit flag is set (the signal is clocked, so only set on a new clock edge) and the state changes to waitOnPulseLow. If no collision is detected, the is_hit flag is set to 0 and the state does not change. The waitOnPulseLow is a state that holds the is_hit signal for the entire pulse width so that is available to other components (mainly score).

Score:

The entity for the score component is described below:

```
entity
score
is
        port (
        clk : in std_logic;
        reset : in std_logic;
        collision : in std_logic;
        pulse : in std_logic;
        score : out std_logic_vector(3 downto 0)
        );
      end entity;
```

The score component also takes a clock, reset, and pulse input. It also takes a collision bit input and outputs the current score. The basic structure for the score is a 2-state state machine. The clocked process simply updates the current score and current state. The two states in the combinational process are the idle state and waitforlow state. In the idle state, the score is updated by one if the collision bit is high while the pulse is high. If not it transitions to the waitforlow state and in this state the component just waits for the next pulse high sequence.

Win Condition:

The win condition module tracks the scores of each player and sets flags to stop the game and display which player won. The inputs for the win condition entity are: clk, reset, pulse, scoreA, and scoreB. The outputs are hold and winner.

```
entity
win_logic
is
                port (
                clk : in std_logic;
                reset : in std_logic;
                pulse : in std_logic;
                scoreA, scoreB : in std_logic_vector(3 downto 0);
                hold : out std_logic;
                winner: out std_logic_vector(1 downto 0)
                );
              end entity;
```

The win condition entity was implemented with a finate state machine that consisted of three states: idle, delay, and endgame. The idle state does nothing, waiting for a pulse to be detected by the entity. Once a pulse has been detected, scoreA and scoreB are compared to the constant SCORE_TO_WIN. If scoreA has reached the winning condition, the win signal is set to “01”, the encoding for player A winning the game. If player B has won the game, the win signal is set to “10”. The win signal is default set to “00” which indicates no player has achieved victory yet. This output is fed into the lcd entity which will display the name of player that has won. Once a win is detected, the entity enters the delay state. The delay state just waits for the pulse to drop to ‘0’ so that the rest of the game can finish computation. At this point the state is set to endgame. In the endgame state, the hold signal is raised to high. This signal is used to stop the counter, effectively stopping the game.

LCD:

The LCD display on the FPGA board was used to display the winning player. It follows the de2lcd mini project closely with a few modifications to meet our needs. The entity is declared as shown below:

```
ENTITY de2lcd
IS
        PORT(reset, clk_50Mhz          : IN    STD_LOGIC;
             winner
        : IN STD_LOGIC_VECTOR(1 downto 0);
             LCD_RS, LCD_E, LCD_ON, RESET_LED, SEC_LED
        : OUT  STD_LOGIC;
             LCD_RW                                    :
BUFFER STD_LOGIC;
             DATA_BUS                  : INOUT
        STD_LOGIC_VECTOR(7 DOWNTO 0));
END de2lcd;
```

The de2lcd consists of a state machine with clear, reset, and write character states. For our purposes, we wanted the LCD to display either “Player 1 Wins!” or “Player 2 Wins!” or nothing depending on the state of the game. Because of this, a two bit input called “winner” was added to the original de2lcd component which controls some of the state transitions to print the correct message. Player 1 wins, player 2 wins, and nobody has won yet correspond to the bit patterns 01, 10, and 00 respectively. The two messages that need to be displayed differ by only one character, so there are only 13 write character states needed to print the messages. At the state that writes the character that differs (the number of the player who wins) the appropriate character is selected based on the winner input. In order to print nothing on the screen, the MODE_SET state has an if statement that either advances to the first write character state or the RETURN_HOME state depending on the winner input. If the winner input is 00, the next state is RETURN_HOME which skips all the write characters.

Counter:

Desired Behavior:

Accept clock, reset and hold signals and output a pulse in a consistent manner.

Implementation:

A simple state machine with two states forms the skeleton of this component. The first state, increment, adds one to a clocked signal on every rising edge. Increment transitions to the hold state when the incoming hold signal is high. While in the hold state the component holds the pulse output low and waits for the hold signal to go low again, at which point it resumes counting in the increment state. The signal keeping track of count was set to be 18 bits wide so that it would eventually roll over back to zero (18 bits was chosen because it produced the most fluid motion onscreen without being too fast). Whenever the counter detects that count is zero while in the increment state it inverts the pulse output signal, thus producing a steady on-off signal.

Simulation:

No simulation was made for this component as it would be impractical to simulate the millions of cycles required to see proper output. It was tested through trial and error on the board.

PS2:

Desired Behavior:

Take in keyboard clock and data signals, a 50mhz clock and a reset signal. Output a 2 bit speed for each tank and a 1 bit fire signal for each bullet.

Implementation:

Brief Subcomponent Description:

PS2 uses a keyboard component and a oneshot component. Keyboard is passed the keyboard clock/data, 50mhz clock, reset and read signals and outputs scan code and scan ready signals. Oneshot takes in a clock and trigger and outputs a pulse. By passing the scan ready signal from the keyboard component to the oneshot component and the oneshot component's pulse signal to the keyboard component, the pulse from the oneshot component will only trigger a rising edge when a new key has been pressed.

PS2 consists of the above-mentioned components, keyboard and oneshot, and a process triggered by the rising edge of the pulse from oneshot. Within this process, the component looks at the scan code received from the keyboard. If it is a break code it sets a flag; if it is a key that is used for controlling something and the flag is low, sets that key's bit high; otherwise it sets the key's bit low.

This procedure allows the component to keep track of which keys are pressed continuously, so you can play the game by holding down keys. At the end of the component there is a combinational process that takes the bits corresponding to used keys and applies some logic to translate it into the desired outputs.

Simulation:

No simulation was made for this component as it functions purely in conjunction with the development board.

GraphicsOut:

Desired Behavior:

Take in (x,y) coordinates for both tanks and both bullets and generate the required signals to draw them onscreen using the VGA interface.

Implementation:

Brief Subcomponent Description:

Graphics out makes use of two components: vga_sync and pixelGenerator. Vga_sync simply takes in a 50mhz clock and generates the signals required to synchronize the output of the pixelGenerator to the monitor's scan. pixelGenerator takes in (x,y) positions for the tanks and bullets and the current pixel being scanned(generated by vga_sync) and outputs the RGB value for the current pixel.

As GraphicsOut just provides an interface for accessing vga_sync and pixelGenerator, pixelGenerator will be discussed here, as it is the largest component. The pixel generator instantiates 5 roms that contain pixel information for tanks A and B, bullets A and B, and the background. It then takes the current pixel being scanned and checks to see if it is within the bounds of bullet A, bullet B, tank A, tank B in that order. If it is in within an object, it then reads the color from that objects rom into a signal. If the color read is the chroma key, it reads and uses the color from the background instead, otherwise it writes the color read to the RGB output for that pixel. If the current pixel is not within any components it writes the color from the background rom to the RGB output.

Simulation:

No simulation was made for this component as it functions purely in conjunction with the development board.

APPENDIX:

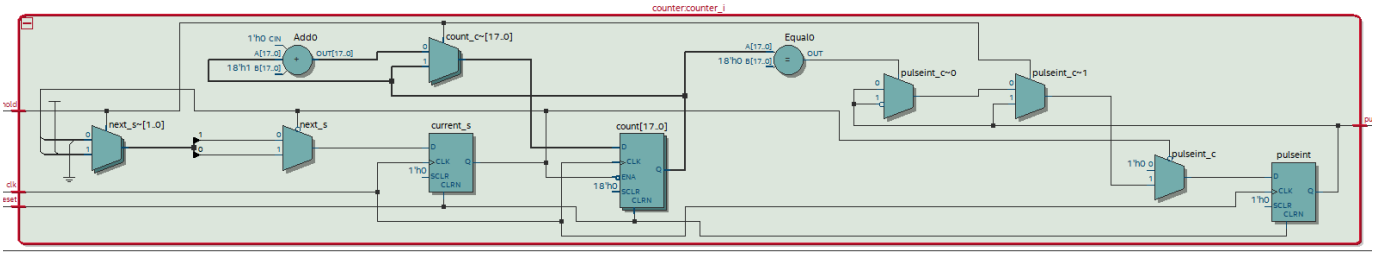


Figure 1: Counter RTL schematic

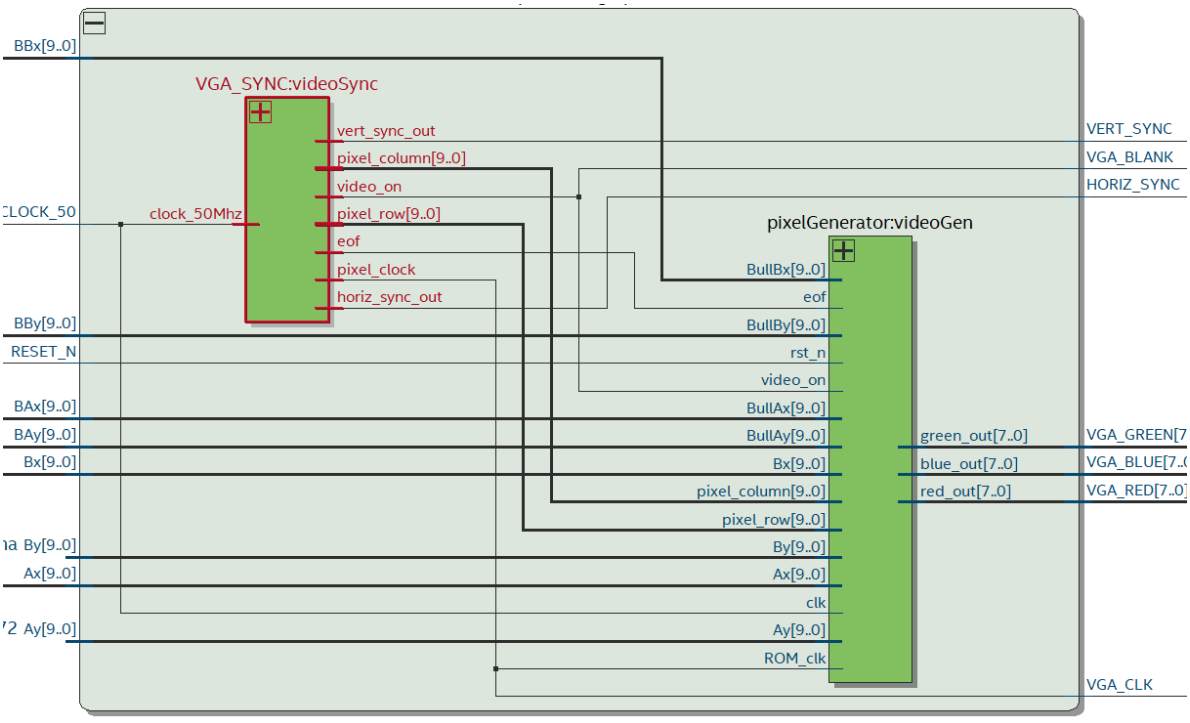


Figure 2: Graphics output RTL schematic

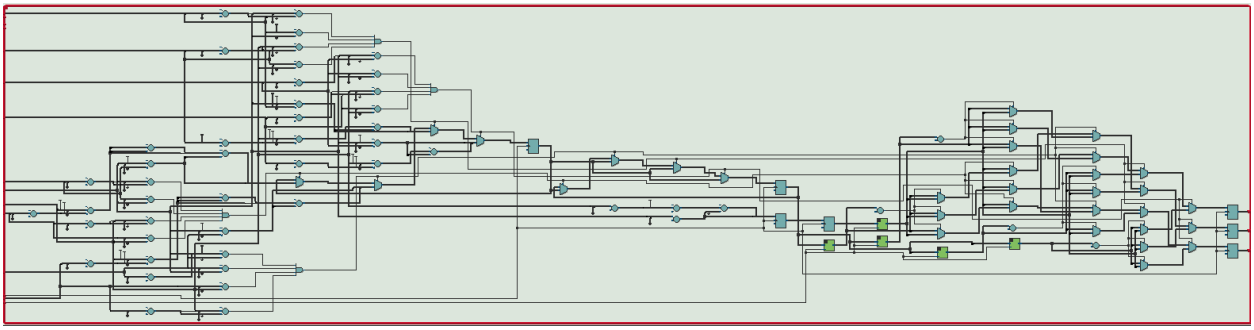


Figure 3: Pixel generator RTL schematic

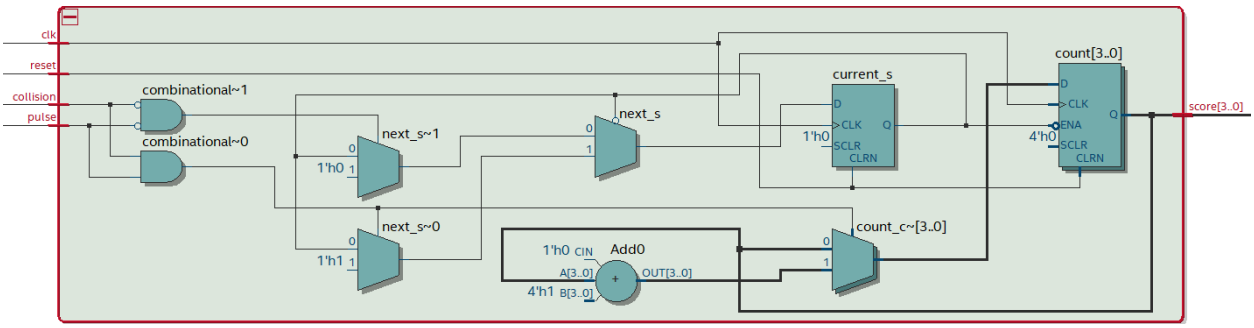


Figure 4: Score component RTL schematic

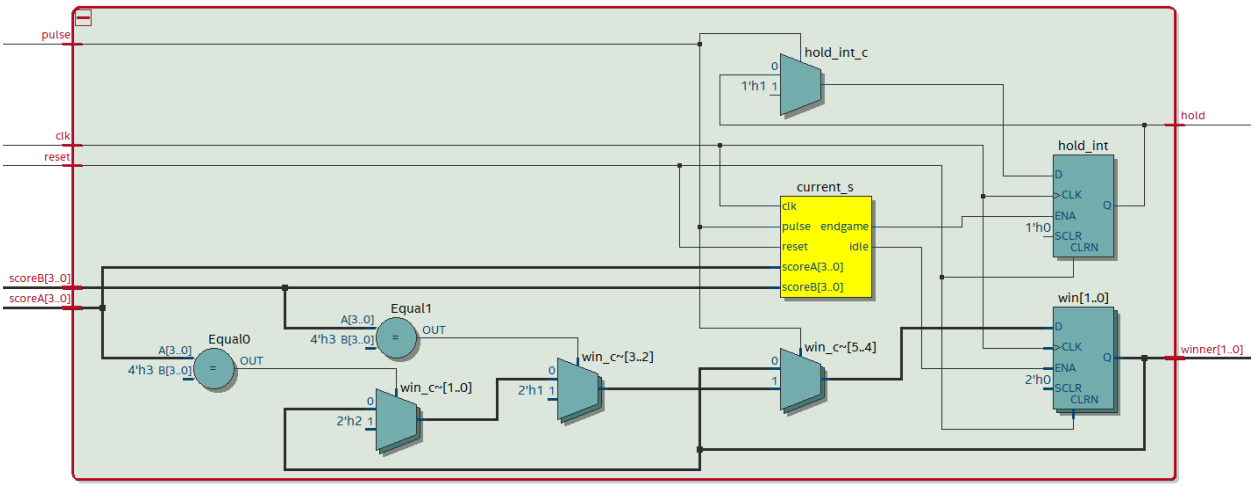


Figure 5: Win logic RTL schematic

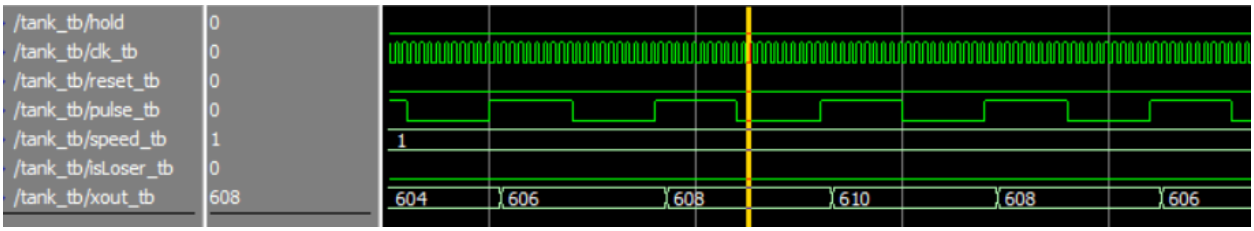


Figure 6: Tank ModelSim wave

Figure 7: Bullet ModelSim wave

Flow Summary

<<Filter>>

Flow Status	Successful - Tue Nov 28 15:56:35 2017
Quartus Prime Version	17.0.0 Build 595 04/25/2017 SJ Lite Edition
Revision Name	tank_game
Top-level Entity Name	tank_game
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	1,485 / 114,480 (1 %)
Total registers	364
Total pins	84 / 529 (16 %)
Total virtual pins	0
Total memory bits	2,075,784 / 3,981,312 (52 %)
Embedded Multiplier 9-bit elements	0 / 532 (0 %)
Total PLLs	0 / 4 (0 %)

Figure 8: Flow summary showing resource usage

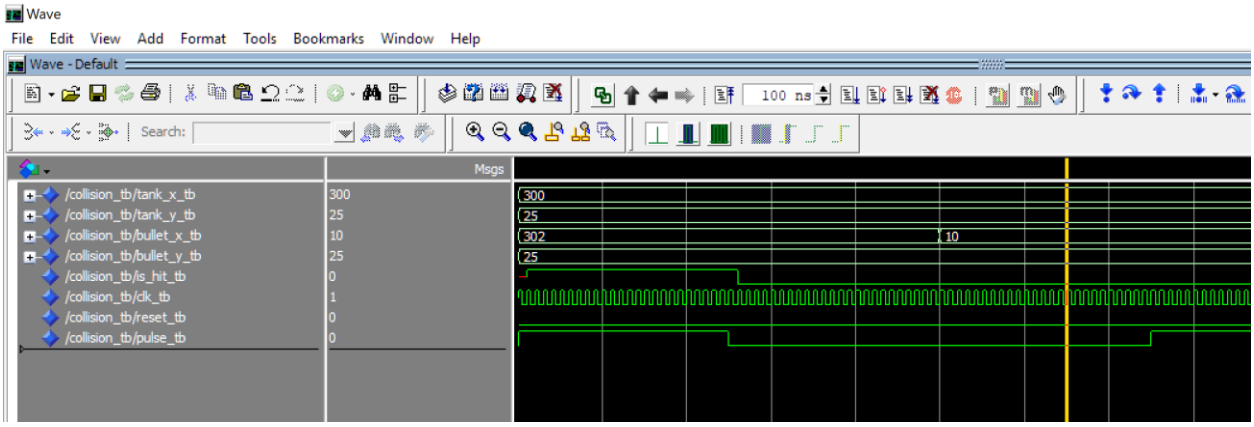


Figure 9: Collision Model Sim Wave