ECE 385

Fall 2021

Final Project Report

Tank Trouble

Blerim Abdullai

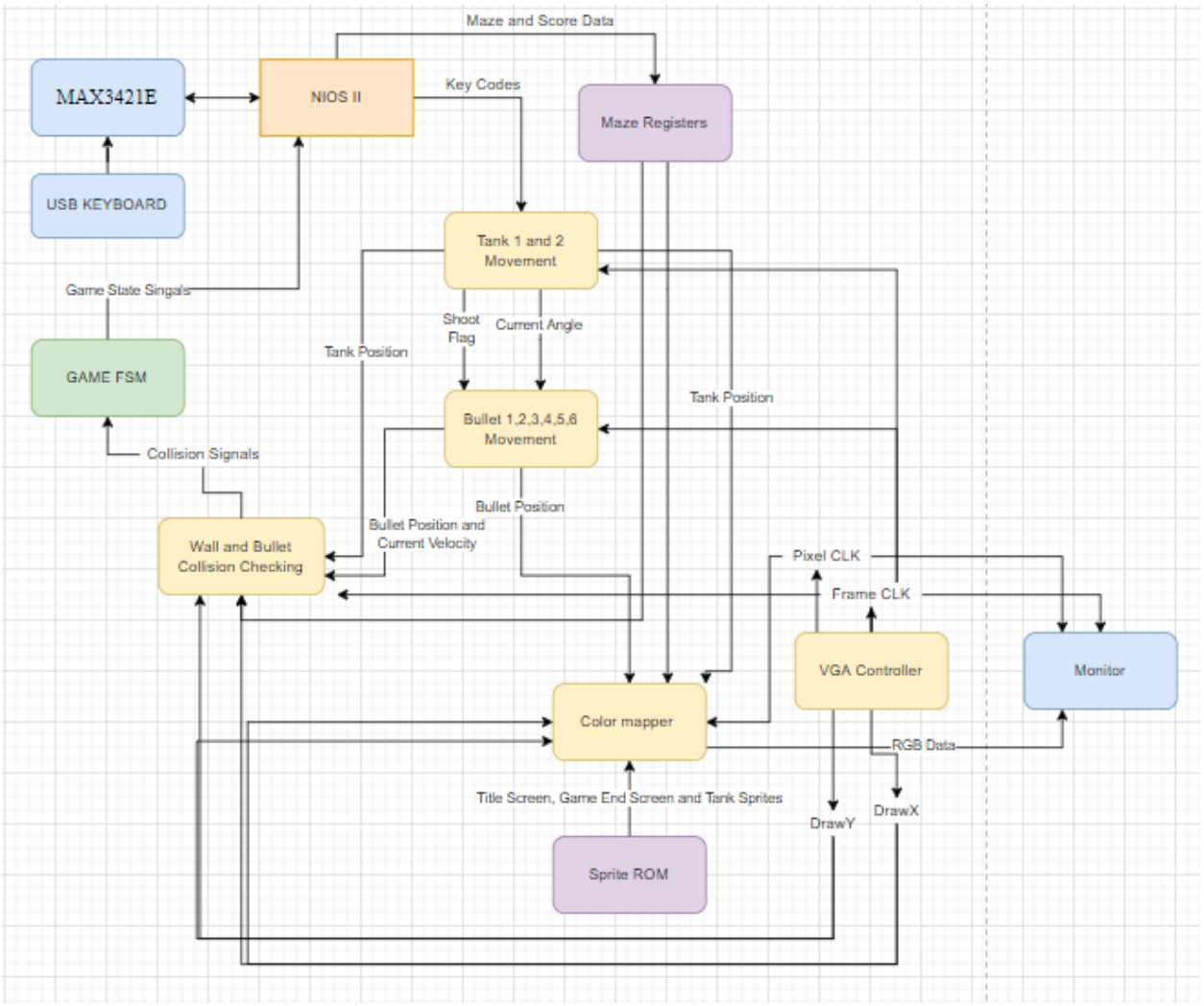
Sara Spahi

Section AB1

Introduction:

This report details the design and implementation of the flash game Tank Trouble within System Verilog and the NIOS II. Tank Trouble is a competitive two player game in which two tanks spawn on opposite corners of the map and can move and rotate similar to an actual tank. Each tank can shoot three bullets at a time, and the objective is to try and hit the other tank with a bullet. The map is a randomly generated maze and the bullets can bounce off of the walls. When a tank gets hit by a bullet, whether its own or from the other player, a round is over, the other player is awarded a point, a new map is generated, and the players are placed in the opposite corners to start a new round. This implementation contains nearly every feature of the original Tank Trouble flash game with the exception of sound and power ups. As an overview, about half of the game features are within software and the NIOS II and half of them are within hardware. Our maze generation, score keeping and round resetting is done within software, and the movement and collisions are handled within the hardware. We also do not have anything "hardcoded" in the sense that we only have a single sprite for each tank and the mazes are generated on the fly by a random algorithm.

High Level Block Diagram:



Player/Sprite Movement and Rotation:

One of the core game mechanics in this game is the ability to rotate in place with the right and left keys and then moving forward and back with the corresponding keys. A bullet should also come out of the tank at the same angle that it is currently facing. We did not want to store a bunch of sprites and velocity data in memory and have to enumerate every single angle with this data in order to achieve the desired behavior. This approach would be time consuming to implement, not be scalable, and would create a choppy feel to the game.

Instead we opted to use sin and cos values in order to calculate X and Y velocities for the tanks and bullets, and to rotate the sprites for the tanks. In order to implement sin and cos within system verilog we created a look up table module with 8-bit values for sin and cos at every 4

degrees. We then pass these values through a MUX which uses the angle index to decide the sign of the value. Since sin and cos are in decimal form we needed to use a standardized decimal convention so that all of our modules are able to use and complete operations with these sin and cos values. The decimal standard we decided to use was fixed point notation because of its simplicity and the fact that all of our operations are on the same order of magnitude.

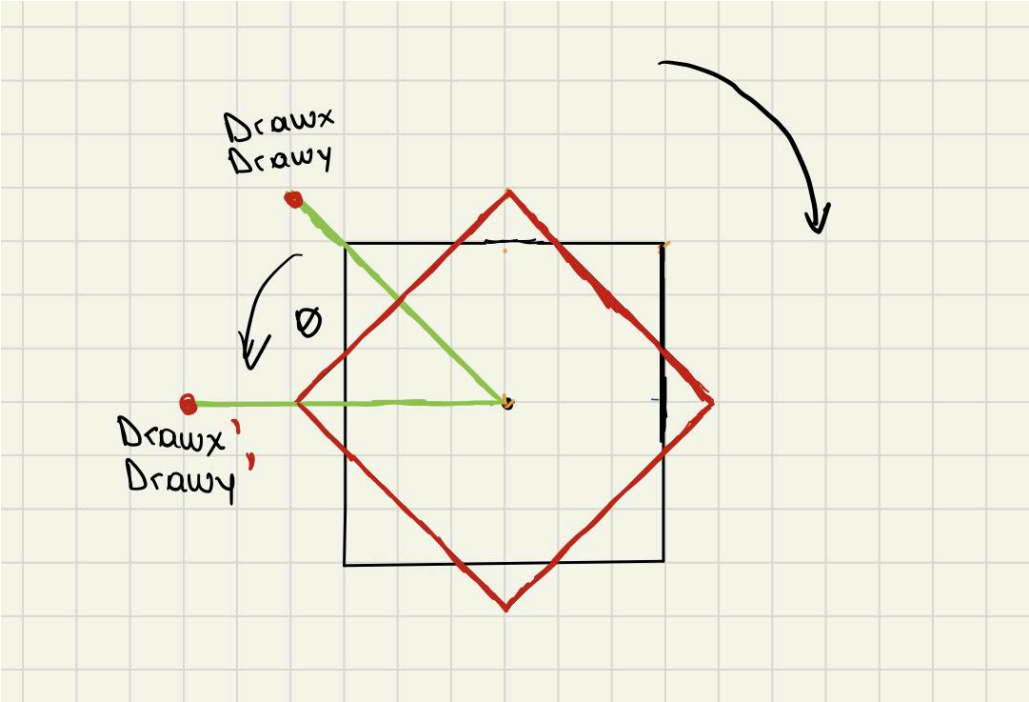
Fixed point notation essentially revolves around picking an arbitrary number of bits to represent the integer portion of a number and a portion of bits which represents the decimal portion. You can then use two's-complement to sign the number.

Sign Bit [15]	Integer Bits [14:7]	Decimal Bits [6:0]

Addition and subtraction do not need any modifications to work, however, there are certain characteristics of fixed point numbers that need to be taken into account when doing multiplication. When doing multiplication it is important that both numbers are in the same format before the operation. You also need to manually compute the sign bit by doing a XOR operation and sign extending. A fixed point value also “grows” in both directions, this means the result of a fixed point multiplication has twice as many decimal bits and twice as many integer bits and you lose the sign bit.

With working fixed point operations and sin and cos values our first step was to make our tanks move in angles. To do this we had the left arrow key increase the angle and the right arrow key decrease the angle of the tank. We used inequalities to roll the value over in order to keep the angles between 0 and 360. We then use the current angle to compute an X velocity and Y velocity using our velocity step size and standard vector decomposition i.e. X velocity = step*cos(currentAngle). One the rising edge of the V-Sync signal we then update the X and Y position by adding the X and Y velocities. One thing to note is that we did need to extend the position bits to include decimal values in order to do the addition and then after the addition was completed we assigned the final position output to the integer portion of the number. This was so that we did not need to make any modifications to our collision and drawing hardware and our output position signal remained 10 bits.

After we gain the ability to travel in all different angles we now need the ability to have the sprite rotate into the corresponding angle in which we wish to travel. We do this with a rotation matrix shown below and two linear transformations. The basic idea is that we rotate the DrawX and DrawY coordinates at the angle to which we want the tank to rotate. We then take the rotated coordinates and feed them into the same inequalities which use the tank position and its size to determine if the color mapper needs to draw a pixel. In order to rotate the tank about its center we need to normalize DrawX and DrawY about the origin which is the top left corner of the screen. This means we subtract DrawX and DrawY by the current position. We can then plug these values along with the current angle into the rotation matrix. Finally we need to add the tank position back to the coordinates in order to end up in the correct position on the screen. This results in a tank that can rotate about the screen.



Rotating the DrawX and DrawY coordinates

$$\begin{bmatrix} cos\emptyset & -sin\emptyset & 0 \\ sin\emptyset & cos\emptyset & 0 \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x.cos\emptyset - y.sin\emptyset \\ x.sin\emptyset + y.cos\emptyset \\ 1 \end{bmatrix}$$

Basic expressions:

TankXRotated = (DrawX-TankXPos)Cos(CurrentAngle) - (DrawY-TankYPos)Sin(CurrentAngle) + TankXPos;

TankYRotated = (DrawX-TankXPos)Sin(CurrentAngle) + (DrawY-TankYPos)Cos(CurrentAngle) + TankYPos;

System Verilog used to compute rotated coordinates including all of the sign extending and bit shifting.

```
DrawXe2[31:0] = {6'b0, DrawX, 16'b0};
DrawYe2[31:0] = {6'b0, DrawY, 16'b0};
// sign extend cos sin
cos2e[31:0] = {{12{cos2[7]}}, cos2[7:0], 12'h0};
sin2e[31:0] = {{12{sin2[7]}}, sin2[7:0], 12'h0};


Tank2Xsp[31:0] = DrawXe2 + ~TankX2e+1'b1;
Tank2Ysp[31:0] = DrawYe2 + ~TankY2e+1'b1;


XMCsign2 = Tank2Xsp[31]^cos2e[31];
YMCsign2 = Tank2Ysp[31]^cos2e[31];
XMSsign2 = Tank2Xsp[31]^sin2e[31];
YMSsign2 = Tank2Ysp[31]^sin2e[31];

XmultCos2[63] = XMCsign2;
XmultSin2[63] = XMSsign2;
YmultCos2[63] = YMCsign2;
YmultSin2[63] = YMSsign2;


XmultCos2[62:0] = Tank2Xsp[30:0]*cos2e[30:0];
XmultSin2[62:0] = Tank2Xsp[30:0]*sin2e[30:0];
```

```
YmultCos2[62:0] = Tank2Ysp[30:0]*cos2e[30:0];
YmultSin2[62:0] = Tank2Ysp[30:0]*sin2e[30:0];


DrawXs2[15:0] = {{6{XMCsign2}}, XmultCos2[41:32]} + {{6{~YMCsign2}},
~YmultSin2[41:32]}+1'b1 +BallX2;

DrawYs2[15:0] = {{6{XMSsign2}}, XmultSin2[41:32]} +
{{6{YMCsign2}},YmultCos2[41:32]} + BallY2;

DrawXs2Prime = DrawXs2[9:0];
DrawYs2Prime = DrawYs2[9:0];
```

Sprite Drawing:

Our game needed very few sprites, just a title screen, win screens and the sprites for the tanks. In order to implement the static sprites we used some inequalities to determine based upon DrawX and DrawY if we need to draw a sprite. If we need to draw a sprite we just normalize the DrawX and DrawY coordinates to the top left of the screen by subtracting the image position and adding the half of the image size. We then index the correct sprite ROM with the formula ADDR = DrawX + DrawY*ImageWidth. We used a palette approach so our sprite ROM gave us an index into our 3 bit color palette. The RGB values are then checked with the game state signals before being displayed.

In order to display our rotated tanks, instead of using the regular DrawX and DrawY coordinates we use the rotated coordinates which we calculated earlier.

System Verilog example for fixed position inequality:

```
else if(t1wscreen)
                begin
                        if ((DrawX[9:0] >= 10'd320 - 10'd80) &&
                        (DrawX[9:0] <= 10'd320 + 10'd80) &&
                        (DrawY[9:0] >= 10'd240 - 10'd45) &&
                        (DrawY[9:0] <= 10'd240 + 10'd45))
                        begin
                                Red_New = PalletW[PalletIWS1[2:0]][23:16];
```

```
                Green_New = PalletW[PalletIWS1[2:0]][15:8];
                Blue_New = PalletW[PalletIWS1[2:0]][7:0];
        end
        else
        begin
                Red_New = 8'hFF;
                Green_New = 8'hFF;
                Blue_New = 8'hFF;
        end
end
```

Maze Generation:

Another core component within the game is the mazes that the tanks compete within. Due to the abundance of maze generating algorithms available we decided to implement randomly generated mazes. The algorithm we chose is the recursive backtracking algorithm because it generates a maze where every point can be reached, and the paths are very long which we thought would make for fun gameplay. The algorithm was also fairly simple to implement in C which would allow us to quickly port it to the NIOS II. This is a basic description of the operation of the algorithm.

1) Declare a 2D array for the amount of horizontal and vertical cells that will represent the maze grid.
2) Pick a random cell in the maze grid to start, mark the current cell you are at as visited and push this cell onto the stack.
3) Check the North, East, South and West neighboring cells to see if they have been visited, push each unvisited cell onto a neighbor stack.
4) Pick a random available direction to travel in from the stack and create a path from the cell you are on to the cell you are going to travel too.
5) “Travel” to the cell and set it as visited, and push it onto the stack. You can now clear the neighbor stack and repeat from step 3.
    a) If you find that there are no available neighbors you need to pop off of the main stack and repeat from step 3 with a previous cell

b) The algorithm terminates when you have visited all of the cells

We found an implementation of this algorithm in C++ by One_Lone_Coder and used a majority of his logic, however, there was quite a bit of porting to do in order to have the algorithm work on the NIOS II and with our hardware. We first converted the algorithm to C, so that it would be able to run alongside the USB keyboard polling on our NIOS II. This required us to write our own stacks, and paired stacks (stack of structs) within C. One_Lone_Coder also had his own console engine which displayed the maze, and this engine would allow you to inflate the cells by an arbitrary amount while keeping the walls the same size. We used the inflating logic in order to write 0s and 1s into a screen buffer to represent walls or empty space. This screen buffer had a final size of 160x120. We then used the VGA Text Interface from lab 7 and resized the VRAM struct so that it would allocate memory in order to fit our new maze data. We essentially have 160 x 120 characters each of which are 4 x 4 pixels that we can write to and a 1 denotes a maze wall and a 0 denotes free space. We then had to resize our VRAM in order to accept the new data and store it in registers so it can be easily accessed by our collision logic. We also needed to modify the drawing algorithm from lab 7 since each 32 bit register stores 32 characters, we have more characters and the characters are now different sizes.

We also leverage the screen buffer in order to draw scores for each player. We do this by storing the score and using it to look up the corresponding number which was drawn in 0s and 1s. Finally we draw the score in the correct places for each play within the maze screen buffer.

Final Expressions for drawing the maze and scores:

```
Byte_ADDR[14:0] = drawxsig[9:2]+drawysig[9:2]*160;
Word_ADDR[9:0] = Byte_ADDR[14:5];//How to index the ram
//Checking if each pixel is a wall or not
currentMaze = Maze_Reg[Word_ADDR][~Byte_ADDR[4:0]];
```

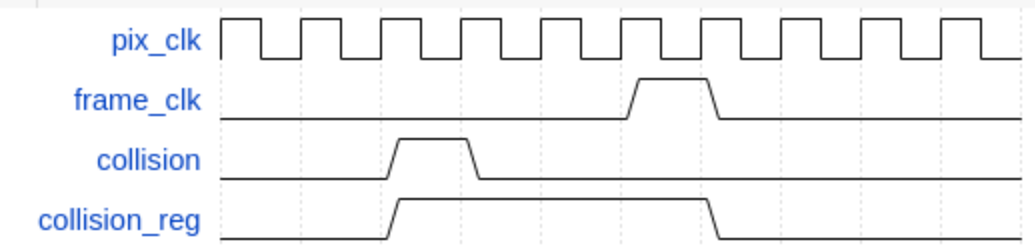
Collisions:

Detecting the collisions of objects with the maze walls is another key component of the game. The collisions are important not only for the tanks but also for the bullets. The bullets

collide and deflect their movement according to the position of the wall relative to the bullet. The initial plan for detecting collisions was to use the x and y position of the object and according to those pixel positions calculate the corresponding entry in the bitfield. If the entry in the bitfield was 1 we would have collided with a wall. However, we realized that that implementation was not robust because it used a lot of multipliers and FPGA resources. Additionally, having multiple objects that collided with the walls meant that we needed multiple instantiations of that module which would increase our compilation time significantly.

We decided to implement the collisions with a method similar to what color mapper does. When we access the current entry in the bitfield, which is used to draw the maze, we at the same time access an entry up (relative to the current entry), an entry down, left and right. If the entry up or down is 1 then there is a vertical wall and if the entry left or right is 1 then there is a horizontal wall. Inside the module we check if DrawX and DrawY are inside the frame of the object, a signal objectOn is assigned. We check whether both objectOn and the current entry of the maze is 1. This means that a collision is detected. However, although this implementation was more robust, it led to clock issues since we were detecting the collisions in the pixel clock. The signals MazeUp, MazeDown,MazeRight and MazeLeft were being calculated each pixel as well. However, the tank and bullet module were updating the positions in the frame clock. By the time the data was read by the frame clock, the value telling us whether we had a collision in the frame is lost.

We had to cross data from the fast pixel clock domain to the slow frame clock domain. Therefore, we built an always_ff block which in the positive edge of the pixel clock checks if we have detected a collision. In that case we save the pulse, together with MazeUp, MazeDown,MazeLeft and MazeLeft. If wall_on goes low, then we save the previous data. The data is reset on the next frame, when drawX and drawY is 0. There is another always_ff block which reads the data from the previous flip flop in the negative edge of the clock. Therefore, this way we have stretched all the pulses that we need to be read on the frame clock and solved the Crossing Clock Domain Issues.



The bullets and the tank can move in any direction which increases the number of conditional statements we had to consider significantly. To reduce the above, we pass the x step and y step motion of the object as input. Therefore, if the object is moving up and left we need to check only for a right wall or a top wall. If the object is moving up and right we need to check for a left wall or a top wall and so on. The module that detects the collision with the walls outputs 4 signals which tell us whether the wall is in the top, bottom, right or left relative to the object.

Each tank and bullet take these signals as input and update their x and y position accordingly. For the tank, whenever there is a wall detected, we complement the x and y motion. For the bullets we need to make sure, the collisions are realistic and they get deflected at a different angle depending whether the wall is on the top, bottom, left or right. If there is a top wall or a bottom wall then only the y motion is complemented. If there is a left or a right wall then the x motion is complemented.

The collisions of the bullets with the tanks were fairly more straightforward, as we just checked whether the x and y position of any of the active bullets is within the frame of the tank. In that case the module outputs a signal signifying that the tank is shot.

Written Description of all .sv Modules

Module: vga_controller.sv

Inputs: Clk, Reset

Outputs: hs, vs, pixel_clk, blank, sync, [9:0] DrawX, DrawY

Description: The VGA controller module is responsible for producing the timing signals Horizontal Sync(hs) and Vertical Sync(vs) which are needed by the VGA monitor. It outputs the current position of the electron beam at each pixel time (DrawX and DrawY).

Purpose: Producing timing signals needed by any VGA monitor and outputs the position of the beam which is then used by other modules

Module: HexDriver.sv

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: This module takes the data from the registers as input and shows it in the hexadecimal LED s of the FPGA.

Purpose: Driver for the hex displays to convert 4-bit data to the appropriate 7-bit value.

Module: game_states.sv

Inputs: CLK, RESET, tank1shot, tank2shot, maze_ready,[31:0] keycode,[1:0] game_reset

Outputs: title, t1wscreen, t2wscreen, [1:0] game_end

Description: This module contains the state machine for the project. It is responsible for the transitions between the different states of the game and the corresponding output signals for each state ,which then decide what we draw into the screen at certain times. The maze_ready, keycode and the game_reset signal are PIOS where Nios II writes into since part of our game logic is handled within the software.

Purpose: Game's state machine

Module: sinCos.sv

Inputs: [5:0] AngleI,

Outputs: [7:0] sin, cos

Description: This module is the sin and cos look up table which is used to lookup the corresponding sin and cos for each angle. It is used for the rotation,the movement of the tanks in different angles and calculating the movement of the bullets.

Purpose: Sin and Cos Lookup tables

Module: Tank1.sv

Inputs: Reset, frame_clk,hit,isWallBottom,isWallTop,isWallRight,isWallLeft,[1:0] game_end, [7:0] sin, cos,[31:0] keycode,[9:0] spawn_pos,[9:0] TankX, TankY, TankS,TankXStep,TankYStep

Outputs: ShootBullet,[5:0] Angle

Description: This module calculates the movement of the tanks in different key presses. Pressing key A rotates the tank counterclockwise, D rotates the tank clockwise, W moves the tank straight, S moves the tank backwards and Q turns the shoot_bullet signal high. The calculation of the X step and Y step motion is done by taking the angle of the tank and using fixed-point for the calculations. In addition to the above the isWallBottom,isWallTop,isWallRight and isWallLeft input signals give information if the tank is collided with a wall and update the movement of the tank accordingly by complementing the previous movement.

Purpose: Contains the logic for the rotation and the movement of the tank in different angles and the shooting control logic.

Module: Tank2.sv

Inputs:Reset, frame_clk,hit,isWallBottom,isWallTop,isWallRight,isWallLeft,[1:0] game_end, [7:0] sin, cos,[31:0] keycode,[9:0] spawn_pos,[9:0] TankX, TankY, TankS,TankXStep,TankYStep

Outputs: ShootBullet,[5:0] Angle

Description: This module calculates the movement of the tanks in different key presses. The left arrow rotates the tank counterclockwise, the right arrow rotates the tank clockwise, the up arrow moves the tank straight, the down arrow moves the tank backwards and the space bar makes the shoot bullet signal 1. The calculation of the X step and Y step motion is done by taking the angle of the tank and using fixed-point for the calculations. In addition to the above the isWallBottom,isWallTop,isWallRight and isWallLeft input signals give information if the tank is collided with a wall and update the movement of the tank accordingly by complementing the previous movement.

Purpose: Contains the logic for the rotation and the movement of the tank in different angles and the shooting control logic.

Module: bullet.sv

Inputs: Reset, frame_clk,hit,isWallBottom,isWallTop,isWallRight,isWallLeft,create,[1:0] game_end,[9:0] tankX,tankY, [7:0] sin, cos, [31:0] keycode

Outputs: is_bullet_active, [9:0] BulletX, BulletY, BulletS,BulletXStep,BulletYStep,[15:0] bulletTimer

Description: This module calculates the movement of the bullet.Whenever the shootBullet signal from one of the tank modules goes high a bullet is created. At the moment of the bullet creation, the sin and cos values from the tanks are saved, is_bullet_active goes high and then fixed point is used to calculate the x step and y motion of the bullets. The position of the bullet is calculated by using the same x step and y motion until a collision is detected and then the step motion changes. If a wall is detected in the top or in the bottom then the y motion is complemented and if a wall is detected in the left or in the right the x motion is complemented. This makes the collision of the bullets with the walls realistic as the bullets are deflected with an angle. There is a timer within the module which gets incremented once the bullet is active. When the timer gets above a certain value then the is_bullet_active goes low. This signal is then used by the color mapper module to decide whether to draw the bullet. The timer and the bullet disappearance allows us to reuse the bullets once they disappear.

Purpose: Contains the logic for the movement of the bullet

Module:angleMux

Inputs: [5:0] Angle, [7:0] sin,cos,

Outputs: [7:0] newSin, newCos

Description: Since the sin cos lookup table only calculates the angles as if they are in the first quadrant of a normal coordinate system, this module takes the sin and cos values output -ed by the sinCos.sv and takes into consideration the new coordinate system where (0,0) is in the top left of the screen, the value of x increases as we move to the left and the value of y increases as we go down the screen.

Purpose: Calculates the new sin and cos values considering our new coordinate system.

Module: CollisionWall

Inputs:[9:0]objectX,objectY,objectS,X_Motion,Y_Motion,DrawX,DrawY, frame_clk,Reset, pixel_clk,hit,currentMazePrime,MazeUpPrime,MazeDownPrime,MazeLeftPrime, MazeRightPrime

Outputs: isWallBottom,isWallTop,isWallRight,isWallLeft

Description: This module is responsible for detecting the collision of an object with the walls. Since taking the position of each object and using that to access the bitfield used a lot of the resources of the FPGA, we decided to detect the collisions in the pixel clock and save it to be read in the frame clock. These signals are then used to decide the position of the wall relative to the object.

Purpose: Taking the step motion of an object, it calculates whether there is an wall in the top, bottom, right or left of the object

Module: TBCollision

Inputs:[9:0]Tank_X_Pos,Tank_Y_Pos,Tank_Size,Bullet1_X_Pos,Bullet1_Y_Pos,isBullet1Active [9:0]Bullet2_X_Pos,Bullet2_Y_Pos,isBullet2Active,[9:0]Bullet3_X_Pos,Bullet3_Y_Pos,isBullet 3Active,[9:0]Bullet7_X_Pos,Bullet7_Y_Pos,isBullet7Active,[9:0]Bullet8_X_Pos,Bullet8_Y_Po s,isBullet8Active,[9:0] Bullet9_X_Pos,Bullet9_Y_Pos,isBullet9Active,

Outputs: TBCollided

Description:This module takes as input the tank position,the x and y position of all the bullets and whether the bullets are active or not. If the x and y position of the bullet are within the frame of the tank, then a collision has happened and the TBCollided signal goes high.

Purpose: Detects whether any of the active bullets in the game collided with a tank.

Module:frameRAM.sv

Inputs: [2:0] data_In, [18:0] wraddress, raddress,we,Clk

Outputs: [2:0] data_Out

Description: This module instantiates the on chip memory where the indexing for the palettes of each sprite is stored.

Purpose: Stores the pallete’s index for the tank sprites and the title

Module: lab62.sv

Inputs : MAX10_CLK1_50, [1: 0] KEY, [9: 0] SW,[7: 0] HEX0, [7: 0] HEX1,[7: 0] HEX2,[7: 0] HEX3,[7: 0] HEX4, [7: 0] HEX5,
Outputs:[9: 0] LEDR,DRAM_CLK, DRAM_CKE, [12: 0] DRAM_ADDR, [1: 0] DRAM_BA, [15: 0] DRAM_DQ, DRAM_LDQM, DRAM_UDQM, DRAM_CS_N, DRAM_WE_N, DRAM_CAS_N, DRAM_RAS_N, VGA_HS, VGA_VS, [7: 0] VGA_R, [7: 0] VGA_G, [7: 0] VGA_B, [15: 0] ARDUINO_IO,ARDUINO_RESET_N
Description:Contains the instantiation of our platform designer SOC which contains the vga controller interface and allows us to draw on the screen.
Purpose: This is our top level module for the project, it allows our CPU to communicate with our drawing hardware.

Module: vga_text_avl_interface.sv
Inputs :CLK,RESET,AVL_READ,AVL_WRITE,AVL_CS, [3:0] AVL_BYTE_EN,[9:0] AVL_ADDR, [31:0] AVL_WRITEDATA,[31:0] keycode_signal, maze_ready,[1:0] game_reset,[19:0] spawn_pos,
Outputs: [1:0] game_end,[31:0] AVL_READDATA,[7:0] red, green, blue,hs, vs
Description: This module is the top level file for the VGA text mode custom ip component on Platform designer. It contains the signals that implement the Avalon MM slave port as input. The slave port allows us to use Nios II to write,read and modify the data for the maze in the registers via software. The module also contains the VGA port signals such as hs,vs, red,green and blue which allow us to display the data in the VGA monitors. All the game modules such as tanks,bullets, collision modules and color mapper are instantiated within this module.
Purpose:Top level file for the VGA text mode core component in Platform designer.

Module: color_mapper.sv
Inputs: module color_mapper ([9:0] BallX1, BallY1, DrawX, DrawY, Ball_size,BallX2,BallY2, Tank2Shot, Tank1Shot, blank, CLK, Sys_CLK,[7:0] sin2, cos2, sin1,cos1, maze,title, t1wscreen, t2wscreen, [7:0] Red, Green, Blue, [9:0] Bullet1X, Bullet1Y, Bullet1S,is_bullet1_active, [9:0]Bullet2X, Bullet2Y, Bullet2S,is_bullet2_active,[9:0] Bullet3X,Bullet3Y, Bullet3S, is_bullet3_active, [9:0] Bullet7X, Bullet7Y, Bullet7S,is_bullet7_active, [9:0] Bullet8X, Bullet8Y, Bullet8S, is_bullet8_active, [9:0] Bullet9X, Bullet9Y, Bullet9S, is_bullet9_active

Outputs:[7:0] Red, Green, Blue

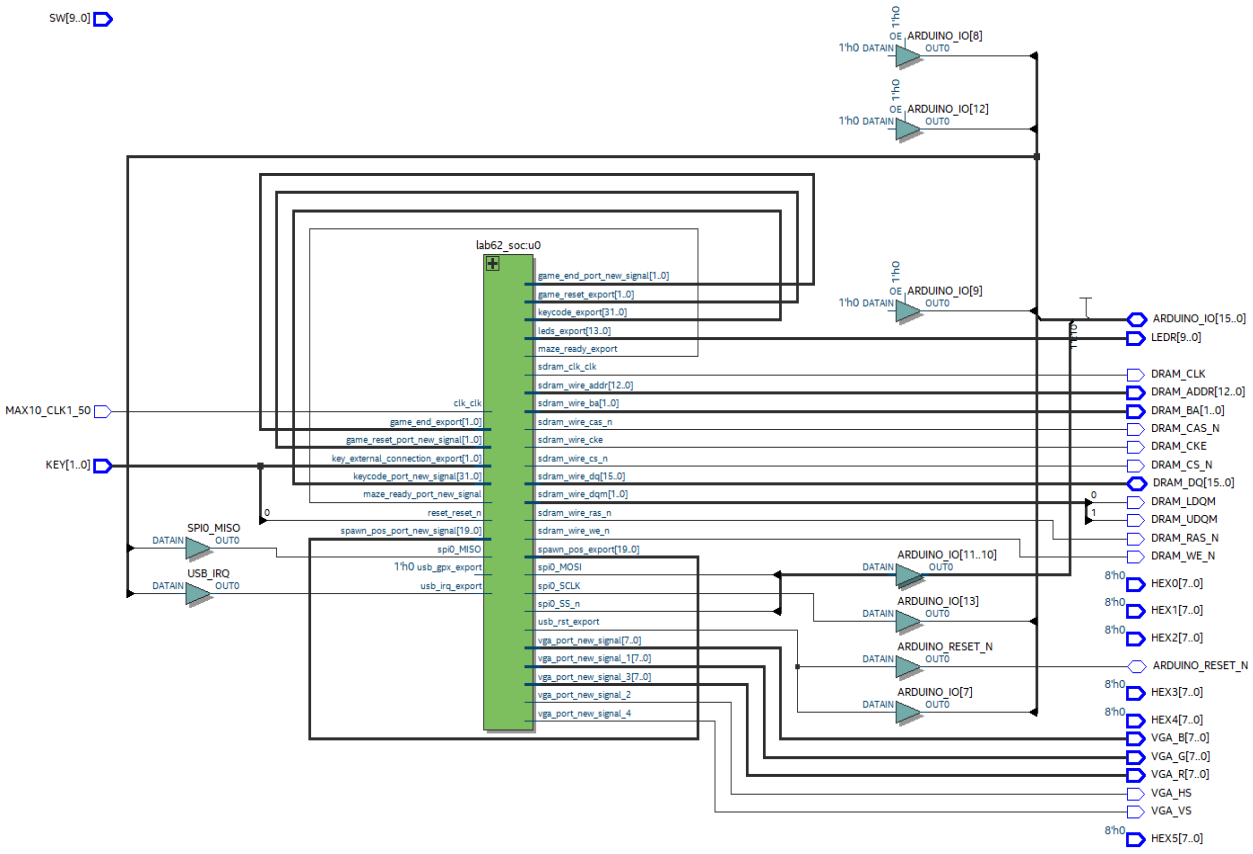
Description: This module takes as input DrawX, DrawY, the positions of each tank, each bullet and signals from the state machine. Based on the signals of the state machine we decide whether we draw the title screen, the game screen or the end screen. Within the game screen there are several conditions that give priority to the maze, bullets or the tank sprites. If the data that we receive when we access the registers that contain the maze is a 1 then we draw a wall otherwise we draw the background(or the sprites).

Purpose: It outputs the corresponding RGB signals to the monitor.

Resources:

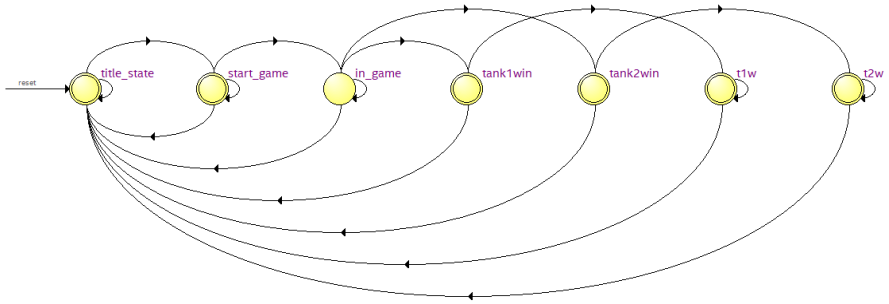
LUT	37,843 / 49,760 (76 %)
DSP	Embedded 9-bit multipliers 64 / 288 (22 %)
Memory(BRAM)	1,021,792 / 1,677,312 (61 %)
Flip-Flop	22209
Frequency	69.89 MHz
Static Power	97.46 mW
Dynamic Power	257.84 mW
Total Power	377.20 mW

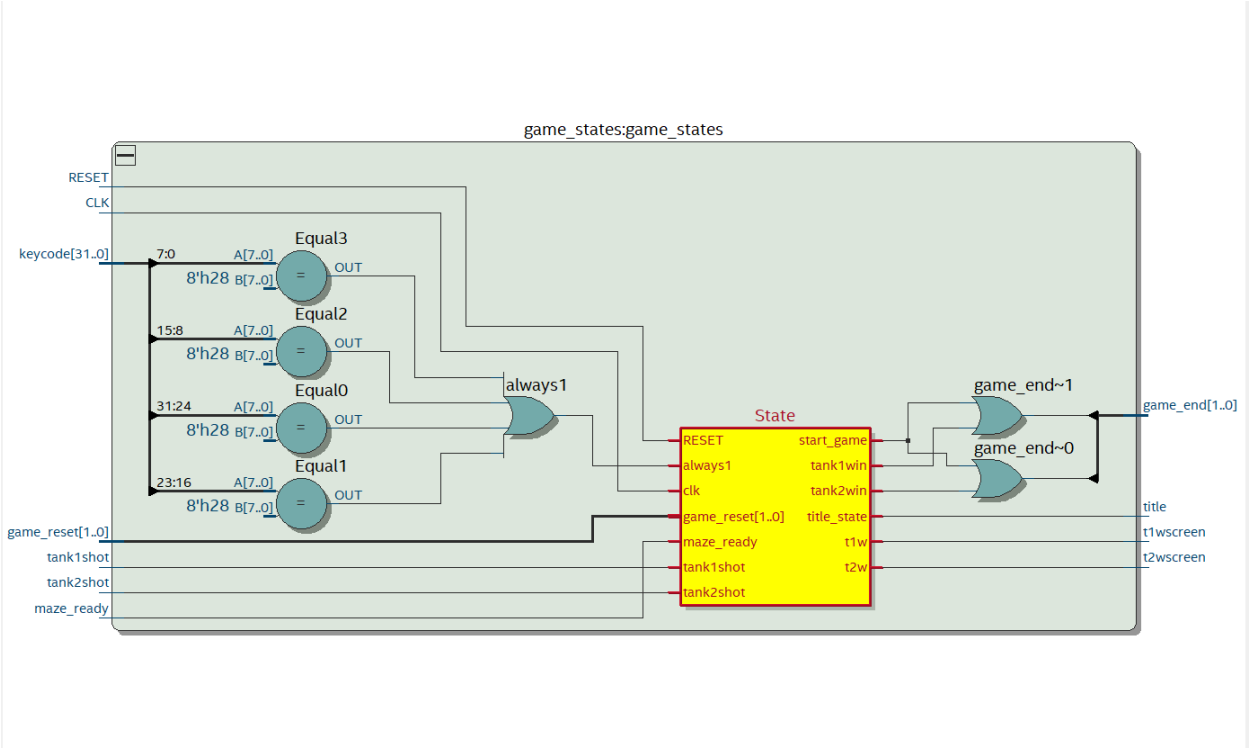
RTL Diagram


SW[9..0]
MAX10_CLK1_50
KEY[1..0]
lab62_soc:u0
clk_clk
game_end_export[1..0]
game_reset_port_new_signal[1..0]
key_external_connection_export[1..0]
keycode_port_new_signal[31..0]
maze_ready_port_new_signal
reset_reset_n
spawn_pos_port_new_signal[19..0]
spi0_MISO
1'h0 usb_gpx_export
usb_irq_export
game_end_port_new_signal[1..0]
game_reset_export[1..0]
keycode_export[31..0]
leds_export[13..0]
maze_ready_export
sdram_clk_clk
sdram_wire_addr[12..0]
sdram_wire_ba[1..0]
sdram_wire_cas_n
sdram_wire_cke
sdram_wire_cs_n
sdram_wire_dq[15..0]
sdram_wire_dqm[1..0]
sdram_wire_ras_n
sdram_wire_we_n
spawn_pos_export[19..0]
spi0_MOSI
spi0_SCLK
spi0_SS_n
usb_rst_export
vga_port_new_signal[7..0]
vga_port_new_signal_1[7..0]
vga_port_new_signal_3[7..0]
vga_port_new_signal_2
vga_port_new_signal_4
SPI0_MISO
USB_IRQ
ARDUINO_IO[8]
ARDUINO_IO[12]
ARDUINO_IO[9]
ARDUINO_IO[11..10]
ARDUINO_IO[13]
ARDUINO_RESET_N
ARDUINO_IO[7]
ARDUINO_IO[15..0]
LEDR[9..0]
DRAM_CLK
DRAM_ADDR[12..0]
DRAM_BA[1..0]
DRAM_CAS_N
DRAM_CKE
DRAM_CS_N
DRAM_DQ[15..0]
DRAM_LDQM
DRAM_UDQM
DRAM_RAS_N
DRAM_WE_N
HEX0[7..0]
HEX1[7..0]
HEX2[7..0]
ARDUINO_RESET_N
HEX3[7..0]
HEX4[7..0]
VGA_B[7..0]
VGA_G[7..0]
VGA_R[7..0]
VGA_HS
VGA_VS
HEX5[7..0]

Game States (Probably explain the PIOS)

As shown in the state diagram we had 7 states, which represented the title screen, in game, tank 1 round win, tank 2 round win, tank 1 final win and tank 2 final win. When the game boots up you enter the title screen which sends a signal to the color mapper to display the title. The players are then prompted to start the game with the enter key and the game will stay in the title screen until the enter key is pressed. Once the enter key is pressed the game_end signal outputs a 3 which is then read by the NIOS and signifies that a new maze must be written into the VRAM and the score must stay the same. The tank modules also receive this signal and disable all user input and set their positions to opposite corners of the map while the game_end>0. Once the NIOS is finished writing the score and the maze into the VRAM a new round is able to start, so the NIOS then toggles the Maze_Ready signal high in order to signal the next state transition. We then transition to the in_game state and toggle game_end to 0 and gameplay proceeds until a tank is hit. If tank 1 is hit we move to the tank 2 round win state and toggle the game_end signal to 2, disabling all player movement, resetting start positions and signaling to the NIOS that tank 2 has won a round. We do this same procedure if tank 2 is hit except the game_end is toggled to 1. The NIOS then increments the correct score based upon the game_end signal and checks if the score is less than the winning threshold of 4. If less than a new maze is generated, the updated score is drawn, the Maze_Ready signal is toggled high and we transition back to the in_game state. If either player's score is higher than 3 then that player has won a round and the NIOS sends a signal to the hardware to transition to a win state. We then transition to the correct win state and toggle the correct winning signal high in order to signal to the color mapper to draw the win screen for that player. When the player presses enter all scores are reset and the game goes to the in_game state effectively restarting a new match.





Bugs, Errors, Issues, Debugging:

Signs, flashing, rotation, hardcode for debugging, Crossing data between, clock domain, passing through some of the walls, blocking and non-blocking

As we began implementing the fixed-point logic, we did not obtain the expected result regarding the movement of the tank. We were able to get the tank moving northwest and southeast, which led us to the conclusion that multiplication with the sin and cos always gave us

positive values. After conducting some research, we discovered that the sign bit was lost during the multiplication process. Therefore, we needed to make sure that we performed the XOR operation on the sign bits each time we multiplied 2 numbers, and we needed to assign this bit after each operation manually.

After implementing the logic to rotate the tanks by first rotating DrawX and DrawY we noticed that the rotation was centered somewhere in the corners of the screen. Although the tank rotated correctly in terms of angle, it did not rotate around its own center. The issue in the calculation of our rotated DrawX and DrawY was that we were actually just calculating the projections of (DrawX - TankX) and (DrawY-TankY). We were not considering the fact that the TankX and TankY had to be added back to the equation after the projections were calculated. Adding the above mentioned variables to our calculation fixed the rotation issues.

Another issue with the tank rotation was that after a 360 degree rotation there was a flashing in the entire screen. In order to debug this, we tried changing the angle calculation conditions in the tank module and saw the changes in the screen. We realized that the comparator was not properly checking the condition when angle became less than 0. This was fixed by declaring the angles as integers instead of logic type.

In addition to the above, crossing clock domains became an issue when checking for collisions with the walls. As we initially started implementing the collision logic, as described above in the collision part of the report, we were planning on using the position of the objects to access the bitfield and decide whether a collision had occurred. The above method required a lot of dividers and multipliers since we had several objects colliding with the walls in the game and our compilation time was increased significantly. We decided to approach this differently and we decided to detect collisions in a similar way to how color mapper draws on each pixel. After writing the module we realized that there were no collision detections at all. After making sure that we had passed the right signals to the module, we hardcoded some of the output signals to see if the movement was affected. The movement of the bullets changed, therefore we came to the conclusion that everything is right with how the signals are passed to the modules. The issue was that we were detecting the collisions in the pixel clock and the movement of the object was updated in the frame clock. For this reason we were not able to detect the collisions. As mentioned in the collision module, we had to stretch the signals so that they could be read in the frame clock.

Final Remarks:

Our design successfully implemented the basic gameplay features of the original tank trouble game as mentioned earlier. We enjoyed the final project format and each learned a lot about system verilog and hardware design. If we had some more time we wish we could have implemented some power ups, sound, and a few animations. Overall, however, the gameplay is smooth with no major or game breaking bugs and most importantly it is fun to play!

Final Product