ECE 385

Spring 2023

Final Project Report

Battle Tanks 1990

Name: Akshat Singh, Adnan Challawala

Lab Section: SL

TA: Shitao Liu

Introduction:

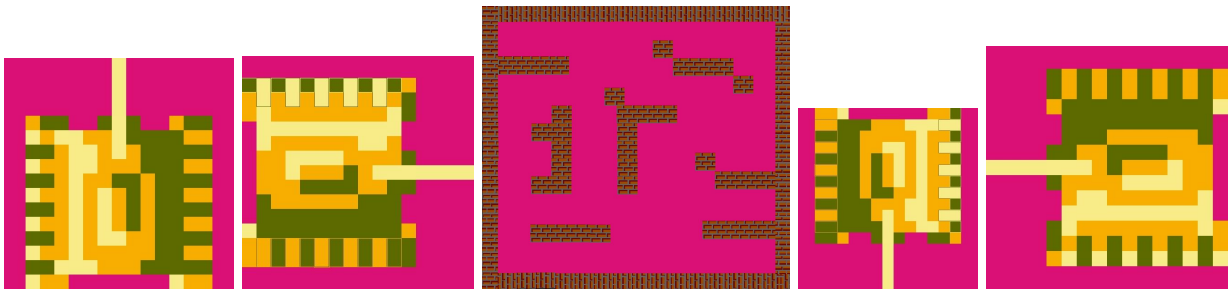
Through the course of this lab we aimed to build a retro game called Battle city which took inspiration from an original game of Battle Tanks 1990. It's a multiplayer game which focuses on the fact that the two players will go around shooting each other with the multiple bullets that are there at their disposal with the tanks losing their health everytime a bullet touches them and the first the tank to lose all their health loses the game. The game at its basis is focused around VGA display and mixing game logic to plan and design the entire game. To start with the design, we started looking at lab 6.2 code and modified it to meet our requirements. We decided to plan our design to first focus on getting the game logic for one tank and then expand to the second player.

Our design is based on the SoC supported by a DE-10 Lite FPGA board. The physical movement and collision detection aspects of the game were handled by hardware modules defined in .sv files. The DE-10 board's on-chip memory was used to store the attributes of tanks, obstacles and other sprites The NIOS II CPU was used to interface with the USB keyboard in order to access the standard gaming "WASD" keys for the first tank and then the 'arrow' keys for the second tank along with the spacebar

and enter for firing bullets. The main point of interaction between the FPGA board and user controls were taken care of by the C code that was written to store the keycodes that came in and to display the characters and sprites on the monitor using a VGA cable.

Basic features and graphics:

These are the sprites used in the creation of our game. The pink backgrounds were taken out by using conditional statements that only allow for RGB to be set where a pink color's RGB values in hex aren't detected.



Implementation:

Initial setup:

The first modifications we started doing was by making the ball not bounce off the walls and then we started to make the ball only once when we touched the keyboard and not continuously move in the direction of the last keycode pressed. Transition to the new code wasn't too bad as we had to modify the old case statements and then make a few changes to account for the negative motion when touching the bounds of the wallpaper. As we got that to work, that was the basic movement of our tank.We decided to continue using the ball from lab 6.2 and get our game logic right before we could proceed to put sprites into our project. The next task that we focused on was to get our tank to start shooting just one bullet as soon as we pressed a spacebar. The bullet was a major design implementation because we wanted the ball to move around the tank until the spacebar is pressed and when the command to shoot a bullet is passed on, the bullet should move in

the direction of where the tank points to. Further we didn't want our bullets to bounce, we wanted them to disappear when it collided with the walls or another tank.

Bullet generation:

To start with the bullet we replicated the lab 6.2 code just to make a small ball and then we defined the bounds for the ball so as soon as it touches the boundary the pixel gun at that point should stop drawing the bullet and instead draw the background again. The first major design choice that we made was the way we implemented the direction of the bullet. As we wanted the bullet to move in the direction of the tank, we could have directly made the bullet follow the direction of the tank but this would cause the bullet to change direction when the tank is moved as well. So to deal with the problem, we made a variable called "direction" and "bullet_direction" which would be the last keycode pressed before the bullet was fired and this wouldn't change its value until the bullet goes off the screen or disappears. The presence of the bullet on the screen was driven by a signal called drawbullet_flag, and understanding how to toggle this signal on/off depending on the game environment took a fair bit of thinking. We implemented this feature quite early in our development timeline, and our next challenge was adding 3 bullets per tank. Therefore, we created a counter and several conditional constructs that drove 3 signals: drawbullet_flag, drawbullet2_flag and drawbullet3_flag. The snippets below gives a basic idea of how these flags were driven and how directions were set:

```
//////////////////////////////////bullet 1 position///////

 if (drawbullet_flag == 1)
 begin
    Bullet_Y_Pos <= (Bullet_Y_Pos + Bullet_Y_Motion);  // u
    Bullet_X_Pos <= (Bullet_X_Pos + Bullet_X_Motion);
 end

 else
 begin
    Bullet_Y_Pos <= tankY;  // Update bullet position with
    Bullet_X_Pos <= tankX;  // SEEING IF USING THE TANKX/TA
 end

 //////////////////////////////////bullet 1 direction////

 if (drawbullet_flag == 1)
 begin
    bullet_direction <= bullet_direction;  // Update tank p
 end

 else
 begin
    bullet_direction <= direction;
 end
```

```
8'd44 : begin  // changed this to X to check if there is a problem with

    if (count == 4'd1)

    begin
        drawbullet_flag = 1'b1;

        if(bullet_direction == 2'b10) // s (down)
            begin
            Bullet_X_Motion <= 0;
            Bullet_Y_Motion<= 6;
            end


        if(bullet_direction == 2'b11) // d (right)
            begin
            Bullet_X_Motion <= 6;
            Bullet_Y_Motion<= 0;
            end
        if(bullet_direction == 2'b00) // w (up)
            begin
            Bullet_X_Motion <= 0;
            Bullet_Y_Motion<= -6;
            end

        if(bullet_direction == 2'b01) // a (left)
            begin
            Bullet_X_Motion <= -6;
            Bullet_Y_Motion<= 0;
            end
    end
```

If collisions between the bullets and any map features or the enemy tank were registered, we simply set the bullet flags back to 0. This was the only togglable signal we implemented without explicitly using an FSM.

Multiple keycode implementation:

After we got our bullet to follow the tank and shoot with a different keycode, we decided to advance onto shooting and moving at the same time. Previously in the Lab 6.2 code, we had only 1 PIO block assigned to execute and process 1 keycode at a time. To move and shoot we would need 2 PIO blocks. So we added another PIO block in the platform designer for this. After we added, we had to make slight modifications in the C code to account for the extra keycode press. The change in the C code was in adding another parameter setKeycode and getKeycode functions. After we instantiated those changes onto our top level and we also made changes in our ball.sv file to take care of the hierarchy of keycode presses. Once we got that to work, we were able to execute 2 keycode presses at the same time, which involved shooting and moving at the same time.

Implementing the second tank:

As we got our game logic down, we decided to extend the same for the second tank, so we could focus on collisions, sprites, aesthetics and game flow after that. For the implementation for the second tank we extended the same logic by making a new module and .sv file and copy, pasting the onto the new file. The only changes that we made were the keycodes that were assigned for the movement for the second tank. Instead of “WASD " controlling the tank, the movements of the second tank were in the hands of the arrow keys and the shooting for the second tank was handled by the ‘enter’ key. Once we extended that, we next moved on increasing the number of keycodes that can be pressed

at the same time. This was necessary for multiplayer functionality as it would allow movement and shooting for both the tanks at the same time, without having to execute keycodes one at a time. This was done in a similar fashion as discussed earlier, where we instantiated new PIO blocks in the platform designer and in our top level along with changes in the C code. Once we got that done, we also had to make changes in both of the tank files by removing the hierarchy / order in the way the keys are pressed. This was done by putting OR statements on the keycodes pressed. After all these changes we had two tanks on our screen which were able to shoot each bullet.

Sprites:

We dealt with the majority of our sprite logic in the color mapper.sv file. The sprites played an important role in the aesthetics of our game. The sprites were generated with the help of Ian's shelper tool. We used the sprites for our tanks, background, main menu, and end screen. The sprites were further integrated with an FSM to generate an animation. The sprites were generated using a python script, where we entered the image size along with the bit size to represent the number of colors that were present in the image that we wanted to create the sprite into. An integral part of each sprite that we made was to color each sprite background into hot pink so we could have sharp images of the sprite. Once we put the image into the python script, we would get a color palette .rom and .mif file which were used to print the sprite onto our screens. The .mif file was used to specify the color index of each and every pixel that the sprite was going to cover. The color index was for the rom file which contained the actual RGB value for each and every color that was used in the sprite.The rom and palette modules that we would add to our color mapper would be instantiated as examples in the example file. An illustration of the rom_module for that image, which dealt with the pertinent memory allocations of that particular image, would be found in the rom file. The palette file would then produce the colors necessary to create the image. From that point forward, anytime we needed to draw a picture, we would take the red, green, and blue colors from the palette file for that particular image. When an image needed to be drawn, we would assign rom to that position and separately set the Red, Green, and Blue colors for the sprites' size boundaries

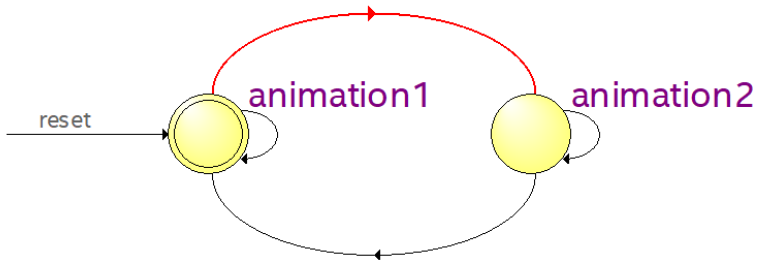
to match those colors' values. All the instantions for the sprites were made in the main color_mapper.sv file. After looking at the .mif file, we got an idea to extend this .mif file to be used for collision detection, instead of using color detection in general.

Collision detection:

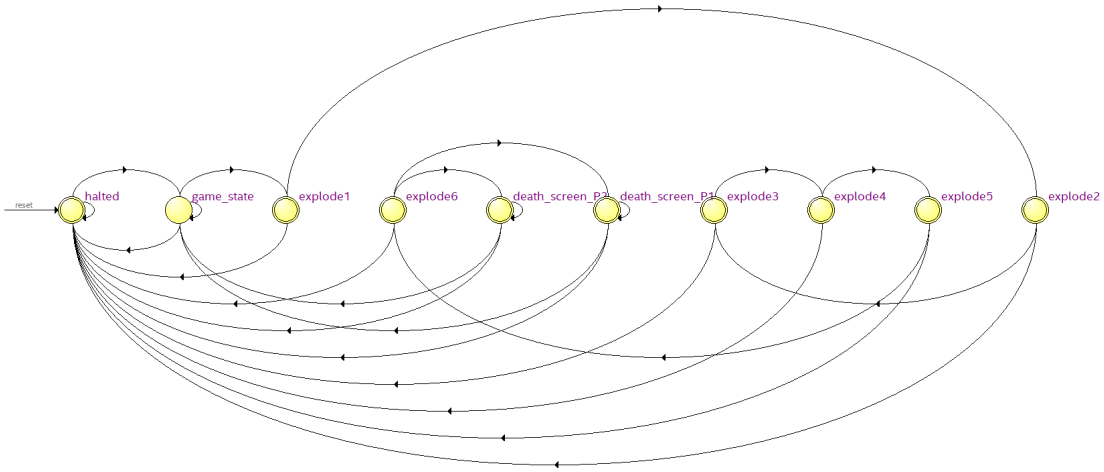
Collision detection was an integral part of our final project. It would not only control the outcome of our game, but it was also necessary to maintain everything within the bounds of the screen. As mentioned before, as a design choice we decided to implement a bitmap instead of color values and coordinates for our collision detection. We decided to use this because we felt we would not only get more precise results in terms of collision detection but our code would also be much more efficient and neat. Initially we created a map in terms of a sprite to generate blocks and boundaries. As we generated a sprite for that, we got a .mif file for it which we modified for our needs. Instead of using multiple integers to represent the different colors in our sprite, we modified into 0's and 1's where 1 represented all the obstacles and 0's represented the free space for the tanks to move along. These values for individual pixels allowed us to make our collisions to be very accurate. Once we got the map and collision detection in, we had to make 4 hit points on your tank to allow for collision in all the directions. At any given point only 2 hit points were activated and we would instantiate a bitmap for each of the activated hitpoint, which would make our collisions even more precise. It worked in such a way that whenever either of the hitpoints encountered a 1 from our bitmaps, a collision flag would be set to 1, this flag would ensure that the tank can't move in that direction by setting a slight bounce in the opposite direction. We did the exact same procedure for our bullet to obstacle collision. For our bullet to tank collision, we used a slightly different approach by using coordinates and vicinity of the pixel gun drawing the tank to make sure if the ball is within the box of the tank, a collision flag is raised and the health of the tank is reduced.

FSM design:

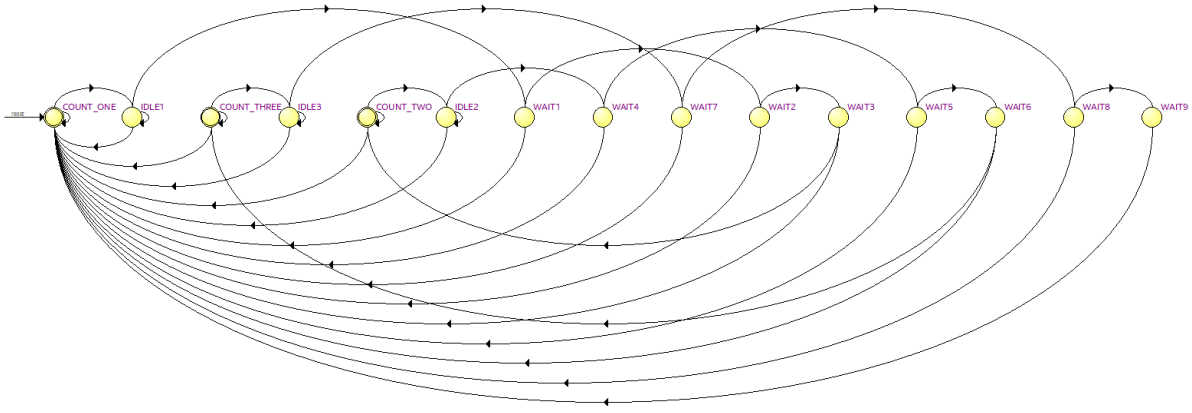
Our game required extensive use of finite state machines in order to toggle signals and add various functionalities. The first one was the **animator FSM,** which had the alternated between two tank sprites while a keycode was held down. This FSM was driven with the use of a 1 Hz clock, which was produced by passing our 50 MHz clock into a module called slow clock and using basic counting logic to produce a positive edge every 5,000,000 cycles. These signals were then set to our color mapper module, where both tanks were given access to the alternate sprites.



Next, we implemented our main FSM: the **game state machine**. This FSM allowed us to move between the menu screen, game screen and player death screens depending on the relevant game updates. This FSM also handled the animation of an explosion, which again involved interfacing alternating sprite signals in the color mapper module right before the displaying of the player death screens. The movement between different screens was done by nesting different sections of the color mapper that handled the production of the **red, green and blue** bit values within conditional statements that activated depending on the state. For example, if the game_on signal went low and death_screenP1 signal went high, the RGB bit values would be loaded with the palette of the P1 death screen. This FSM was driven with the VGA_VS clock of 60 Hz which gave a perfect transition time while allowing for the explosion to be visible near the end of the game.



One of the most challenging implementations in our game was that of multiple bullets. We decided to have 3 bullets being fired and as discussed before, this used an FSM acting as a counter with count states 1, 2 and 3. This FSM was driven by VGA_VS as well and had multiple wait states between count states in order to prevent a rapid transition which could cause strange latch behavior and the firing of 2 bullets if 2 counts were registered in a single transition. Rather than instantiating two instances of the same counter module for both tanks, the specific keycodes required the creation of 2 counters.



These were the FSMs explicitly generated by us. A lot of implicit combinational logic also emulates FSMs as evidenced by several categories in our final project's state machine viewer (such as the setting of bullet directions and flags). Working on this project helped us truly understand the power of a finite state machine in generating a synthesizable "toggle" signal, as hardware cannot synthesize a simple block of high level code that triggers an if statement when a signal goes high and then immediately sets the signal back to some other value.

Positive edge detector and slow clock:

```
always_ff @(posedge clock) begin
    if (reset) posedge_detected <= 0;
    else if (health != 6'd0) posedge_detected <= collision && !signal_in_prev; // press down, so collision goes high and signal in goes high too
    signal_in_prev <= collision; // but then immediately, signal in prev goes high so posedge detected goes loww
end
// and when collision goes back down, posedge remains 0
```

We ran into issues attempting to prevent multiple runs of an operation even if the signal was triggered for a few microseconds, since some of our clocks run at speeds greater than 25 MHz. Our design used an assortment of clock signals, including VGA_VS (60 Hz), vga_clk (25 MHz) and MAX10_CLK1_50 (50 MHz). While these clocks served most modules well, we had to generate a slow clock for certain instances using a counter. These two basic hardware modules aided multiple aspects of our design.

Health bar generation:

Two instantiations of this module stored the health of our tank, which was a variable directly sent to our color mapper in order to produce the bounds for the health bar on the screen. It made use of our positive edge detector to reduce the health of a tank by a predetermined amount when our **collision handler module** was triggered by an overlap between one of the tanks and bullets from the opposing tank.

```
always_ff @ (posedge clock) begin

   if (reset) begin
      health <= 6'd50;
      tank_dead <= 1'b0;

   end

   else if (posedge_detected && !shield_on) begin
   health <= health - 10;
   tank_dead <= 1'b0;
   end

   else if (health == 6'd0) begin
   health <= 6'd0;
   tank_dead <= 1'b1;
   end

end
```

Random block generation:

Before creating our own map using photoshop and bitmap logic, we were attempting to create a randomized map with a combination of breakable and unbreakable blocks. To this end, our initial design involved populating a 640 x 480 array with 0s, 1s, 2s and other numbers representing different block types in C, storing this array in memory, and interfacing it with our SystemVerilog code via the **avalon bus module** from Lab 7. We would then use these bits in our color mapper for the electron gun to decide which pixels to draw to the screen and subsequently generate our map. However, we encountered memory issues with this implementation and realized that the aesthetics of the generated map simply didn't work out the way we wanted it to.

Next, we tried generating random blocks on the screen using C code to produce a 20 bit random number of which 10 bits were used for the X coordinate and the other 10 bits were used for the Y coordinate. The C code was interfaced with the SystemVerilog code via a PIO block synthesizing a wire named randcoordwire. This method worked successfully but caused the block to jump around the screen as the C code kept running. We next attempted making a packed unpacked array of 20 registers and populating them with different randx and randy positions using a for loop. This allowed us to retain the position of the block as the registers filled up in a clock cycle, creating a stable output. However, we learnt that a for loop doesn't synthesize sequentially in SystemVerilog but

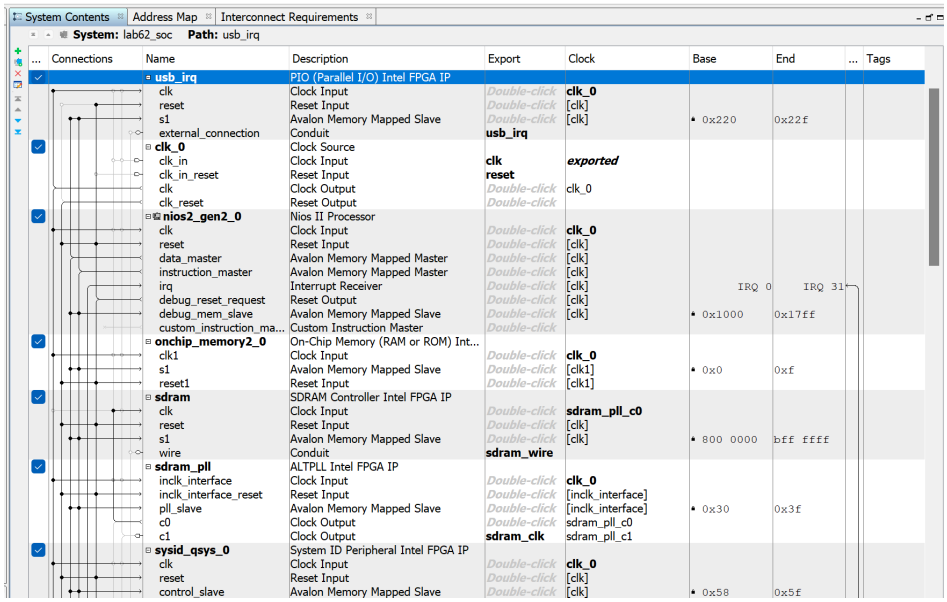
rather unfolds and receives the same signals in a clock cycle. This prevented us from populating all 20 registers with different coordinates. Between this issue and the potential overlap of block bounds, we decided that random map generation in this way would not be practical.

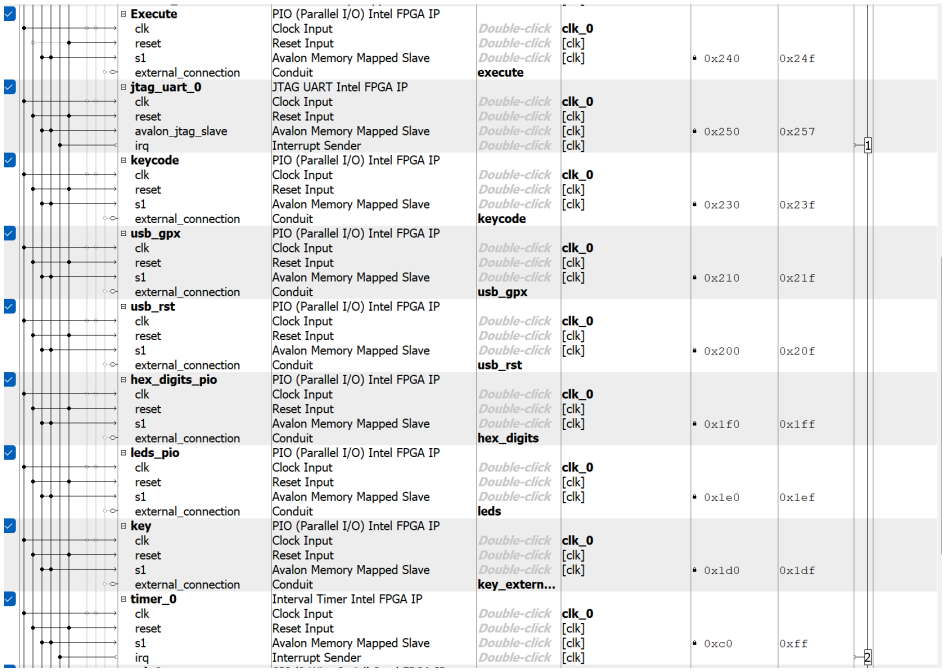
Ultimately, we decided to create an LFSR or **linear feedback shift register** to produce a single pair of bits to generate the coordinates for one randomly placed power-up sprite. An LFSR is a type of shift register that generates a pseudo-random sequence of binary digits. It works by shifting the contents of a register by one bit at each clock cycle, and then using a linear feedback function to determine the value of the input bit that will be added to the register. The feedback function is typically implemented using XOR gates, which take the outputs of certain taps in the register and perform an exclusive-or operation on them to generate the feedback bit. Specific implementations of an LFSR also rely on the various errors produced by flip flop race conditions as covered in the first lab for ECE 385. Our LFSR didn't produce the power-up sprite within the relevant bounds, but our random generation worked. The design was based on what we understood from a stackoverflow post, and represents one of the only viable ways to generate randomness in hardware.

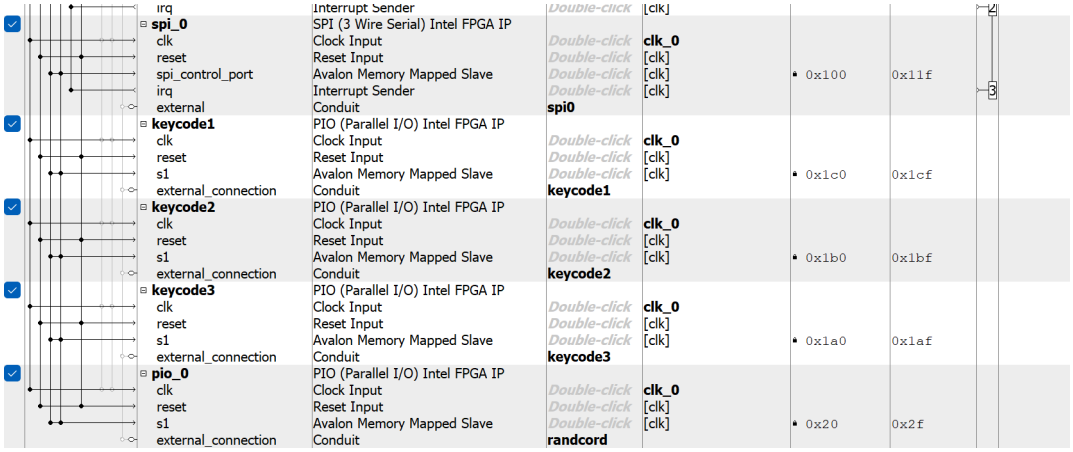
Shield generation:

As the final feature of the game, the shield module generated a 10 second shield which prevented the loss of any health for a specific tank while it lasted. The logic for shield acquisition was the only part of our design that used **color based collision detection**, where for a specific pixel, if either tank's draw signal and the randomly generated shield power up block's draw signal were simultaneously high, the shield acquisition signal was triggered. This drove a positive edge detector and subsequent timer running on the slow clock to prevent health loss for 10 seconds.

PIO block module descriptions:


System Contents
Address Map
Interconnect Requirements
System: lab62_soc Path: usb_irq
... Connections Name Description Export Clock Base End ... Tags
usb_irq PIO (Parallel I/O) Intel FPGA IP
clk Clock Input Double-click clk_0
reset Reset Input Double-click [clk]
s1 Avalon Memory Mapped Slave Double-click [clk] 0x220 0x22f
external_connection Conduit usb_irq
clk_0 Clock Source
clk_in Clock Input clk exported
clk_in_reset Reset Input reset
clk Clock Output Double-click clk_0
clk_reset Reset Output Double-click
nios2_gen2_0 Nios II Processor
clk Clock Input Double-click clk_0
reset Reset Input Double-click [clk]
data_master Avalon Memory Mapped Master Double-click [clk]
instruction_master Avalon Memory Mapped Master Double-click [clk]
irq Interrupt Receiver Double-click [clk] IRQ 0 IRQ 31
debug_reset_request Reset Output Double-click [clk]
debug_mem_slave Avalon Memory Mapped Slave Double-click [clk] 0x1000 0x17ff
custom_instruction_ma... Custom Instruction Master Double-click
onchip_memory2_0 On-Chip Memory (RAM or ROM) Int...
clk1 Clock Input Double-click clk_0
s1 Avalon Memory Mapped Slave Double-click [clk1] 0x0 0xf
reset1 Reset Input Double-click [clk1]
sdram SDRAM Controller Intel FPGA IP
clk Clock Input Double-click sdram_pll_c0
reset Reset Input Double-click [clk]
s1 Avalon Memory Mapped Slave Double-click [clk] 800 0000 bff ffff
wire Conduit sdram_wire
sdram_pll ALTPLL Intel FPGA IP
inclk_interface Clock Input Double-click clk_0
inclk_interface_reset Reset Input Double-click [inclk_interface]
pll_slave Avalon Memory Mapped Slave Double-click [inclk_interface] 0x30 0x3f
c0 Clock Output Double-click sdram_pll_c0
c1 Clock Output sdram_clk sdram_pll_c1
sysid_qsys_0 System ID Peripheral Intel FPGA IP
clk Clock Input Double-click clk_0
reset Reset Input Double-click [clk]
control_slave Avalon Memory Mapped Slave Double-click [clk] 0x58 0x5f


Execute PIO (Parallel I/O) Intel FPGA IP
clk Clock Input Double-click clk_0
reset Reset Input Double-click [clk]
s1 Avalon Memory Mapped Slave Double-click [clk] 0x240 0x24f
external_connection Conduit execute
jtag_uart_0 JTAG UART Intel FPGA IP
clk Clock Input Double-click clk_0
reset Reset Input Double-click [clk]
avalon_jtag_slave Avalon Memory Mapped Slave Double-click [clk] 0x250 0x257
irq Interrupt Sender Double-click [clk]
keycode PIO (Parallel I/O) Intel FPGA IP
clk Clock Input Double-click clk_0
reset Reset Input Double-click [clk]
s1 Avalon Memory Mapped Slave Double-click [clk] 0x230 0x23f
external_connection Conduit keycode
usb_gpx PIO (Parallel I/O) Intel FPGA IP
clk Clock Input Double-click clk_0
reset Reset Input Double-click [clk]
s1 Avalon Memory Mapped Slave Double-click [clk] 0x210 0x21f
external_connection Conduit usb_gpx
usb_rst PIO (Parallel I/O) Intel FPGA IP
clk Clock Input Double-click clk_0
reset Reset Input Double-click [clk]
s1 Avalon Memory Mapped Slave Double-click [clk] 0x200 0x20f
external_connection Conduit usb_rst
hex_digits_pio PIO (Parallel I/O) Intel FPGA IP
clk Clock Input Double-click clk_0
reset Reset Input Double-click [clk]
s1 Avalon Memory Mapped Slave Double-click [clk] 0x1f0 0x1ff
external_connection Conduit hex_digits
leds_pio PIO (Parallel I/O) Intel FPGA IP
clk Clock Input Double-click clk_0
reset Reset Input Double-click [clk]
s1 Avalon Memory Mapped Slave Double-click [clk] 0x1e0 0x1ef
external_connection Conduit leds
key PIO (Parallel I/O) Intel FPGA IP
clk Clock Input Double-click clk_0
reset Reset Input Double-click [clk]
s1 Avalon Memory Mapped Slave Double-click [clk] 0x1d0 0x1df
external_connection Conduit key_extern...
timer_0 Interval Timer Intel FPGA IP
clk Clock Input Double-click clk_0
reset Reset Input Double-click [clk]
s1 Avalon Memory Mapped Slave Double-click [clk] 0xc0 0xff
irq Interrupt Sender Double-click [clk]



1) **Jtag_uart_0:** This module was instantiated in the platform designer to allow for easy debugging of the C code on the Eclipse by allowing us to communicate with the Command line terminal and commands. The JTAG UART is a hardware device which allows for communication between the FPGA and other devices. The JTAG UART module allows for bi-directional communication between the device and an external device or computer
2) **Keycode:** The keycode module is used for instantiating the keyboard hardware device which is used as a communication input device into the FPGA. It enables the processor to retrieve unique codes that correspond to a pressed key on a USB keyboard.
3) **Keycode1:** The keycode module is used for instantiating the keyboard hardware device which is used as a communication input device into the FPGA. It enables the processor to retrieve unique codes that correspond to a pressed key on a USB keyboard.
4) **Keycode2:** The keycode module is used for instantiating the keyboard hardware device which is used as a communication input device into the FPGA. It enables the processor to retrieve unique codes that correspond to a pressed key on a USB keyboard.
5) **Keycode3:** The keycode module is used for instantiating the keyboard hardware device which is used as a communication input device into the FPGA. It enables

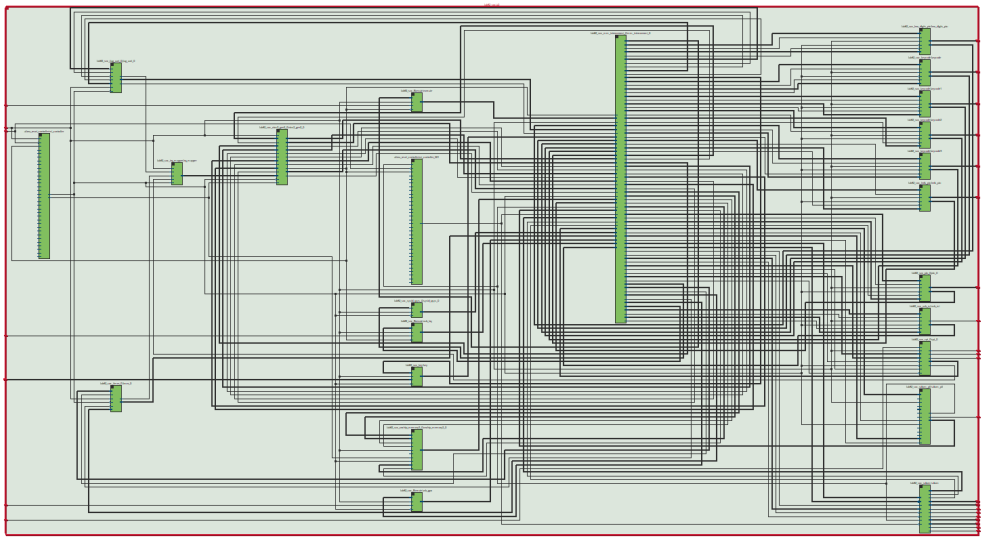
the processor to retrieve unique codes that correspond to a pressed key on a USB keyboard.

6) **usb_irq:** This module is required to for the interaction between the keyboard and the FPGA as it sends an interrupt signal which is used by the USB device to notify the host (computer) about an event that requires attention, such as the insertion of a new device or the completion of a data transfer.
7) **usb_gpx:** This is another module which is required for communication between the FPGA and the keyboard, GP Stands for USB General Purpose Input/Output. It provides a set of pins that can be configured by the device designer for various purposes, such as controlling LEDs, reading switches, or interfacing with sensors.
8) **Usb_rst**: A signal used to reset the USB device or put it in a low-power mode. It is typically activated by the host, either through a software command or a hardware pin. This is needed to utilize the USB keyboard with the FPGA.
9) **HEX_digits_pio:** This module is used for instantiating the Hex displays on the FPGA which are responsible for displaying the ASCII value for each character that is pressed on the keyboard. By utilizing a 16-bit parallel interface, the transfer of multiple digits can be accomplished simultaneously, resulting in faster and more efficient communication.
10) **Timer_0**: Used to send signals to the FPGA so that NIOS II can check time passed.
11) **Spi_0**: This module is instantiated to allow the devices to communicate over the API protocol. More specifically it allows the slave and master channels to coo=mmunicate over MISO and MOSI datapaths
12) **LED:** The LED was defined as a PIO(Parallel I/O) with specifications as an output. The LED was used to display the output from the accumulator which was calculated by taking in the inputs from the switches.
13) **Switches:** Similar to the previous component the switches were defined as PIO(Parallel I/O) but they were used to take data from the user by flipping the switches that are present on the DE-10 Lite FPGA board.

14) **Clk_0:** This is a 50 MHz clock from the FPGA that is used to synchronize a clock and a reset which are used to clock and reset other modules.
15) **Nios2_gen2_0:** This module is used to instantiate the module that is used to define the NIOS II processor that we use to perform various tasks like writing C/C++ on eclipse to making these modules on the platform designer.This makes it easy for software developers to write and debug code for the processor, reducing the time and effort required for development. One of the key advantages of the NIOS II processor is its ease of use.
16) **sdram:** SDRAM is the memory control module which is responsible for providing an interface between the processor and SDRAM itself. It used to reserve memory blocks, initialize data and provide functionality for initial read and write functions as well. When the Nios II processor needs to access data from the SDRAM, it sends a read request to the memory controller. The memory controller retrieves the requested data from the SDRAM and sends it back to the processor. Similarly, when the processor needs to write data to the SDRAM, it sends a write request to the memory controller, which writes the data to the appropriate memory location. The SDRAM's speed and capacity can affect the overall performance of a Nios II system.
17) **Sdram_pll:** The SDRAM clock is linked to a clock signal that is intentionally delayed by 1 ns. This delay is implemented to compensate for any phase discrepancies that may arise between the master and slave clocks.
18) **Sysid_qsys_0:** This is used for checking errors in the connections from the hardware to software components that are connected.
19) **pio_0:** This PIO block was used for generating a random 20 bit number and sending interfacing our hardware with the C code via “randcoordwire”.

SystemVerilog files and module descriptions:

1. finalproject.sv :



Module: finalproject

Purpose: This is the **top level module,** tying together all the modules of our design by instantiating all of them and establishing the relevant connections between them. The input and output signals also contain Arduino class signals which interfaces with the USB drivers generated by the C code.

Inputs and outputs:

```
////////// Clocks //////////
input     MAX10_CLK1_50,

////////// KEY //////////
input    [ 1: 0]   KEY,

////////// SW //////////
input    [ 9: 0]   SW,

////////// LEDR //////////
output   [ 9: 0]   LEDR,
```

- //////// HEX ////////
- output [7: 0] HEX0,
- output [7: 0] HEX1,
- output [7: 0] HEX2,
- output [7: 0] HEX3,
- output [7: 0] HEX4,
- output [7: 0] HEX5,

//////// SDRAM ////////

- output DRAM_CLK,
- output DRAM_CKE,
- output [12: 0] DRAM_ADDR,
- output [1: 0] DRAM_BA,
- inout [15: 0] DRAM_DQ,
- output DRAM_LDQM,
- output DRAM_UDQM,
- output DRAM_CS_N,
- output DRAM_WE_N,
- output DRAM_CAS_N,
- output DRAM_RAS_N,

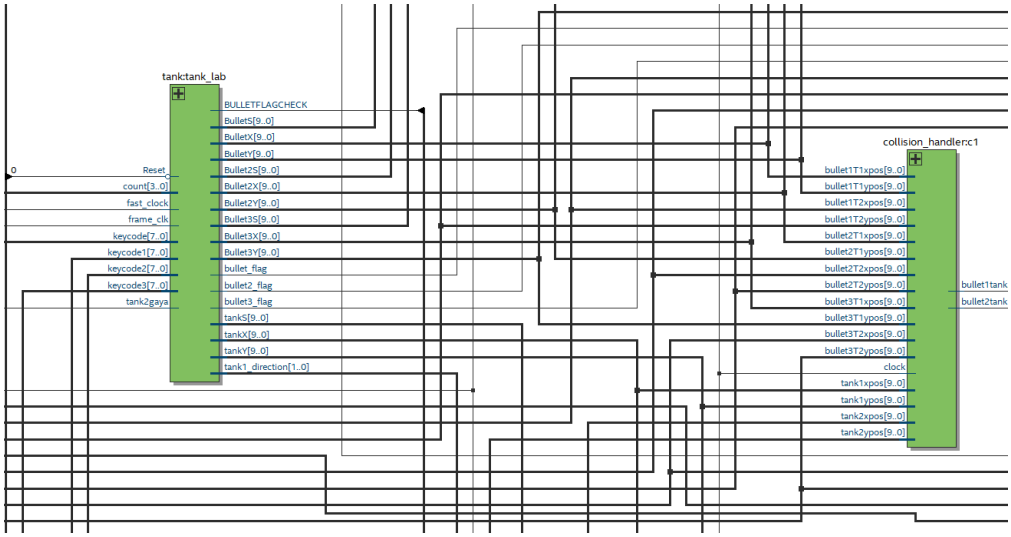
//////// VGA ////////

- output VGA_HS,
- output VGA_VS,
- output [3: 0] VGA_R,
- output [3: 0] VGA_G,
- output [3: 0] VGA_B,

//////// ARDUINO ////////

- inout [15: 0] ARDUINO_IO,
- inout ARDUINO_RESET_N

2. ball.sv:



Module: tank

Inputs: Reset, frame_clk, fast_clock, tank2gaya,

[7:0] keycode, keycode1, keycode2, keycode3,

[3:0] count

Outputs: bullet_flag, bullet2_flag, bullet3_flag,

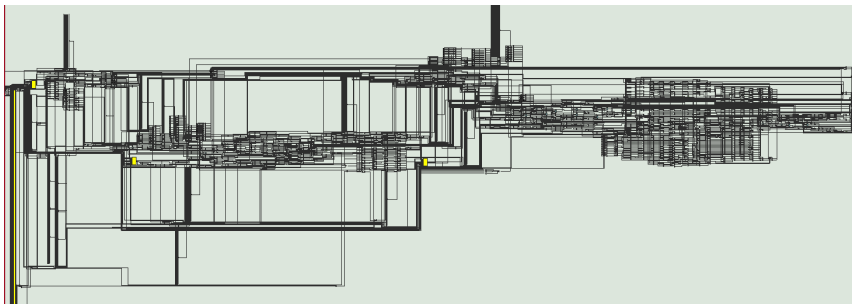
tank1_direction,

[9:0] tankX, tankY, tankS,

[9:0] BulletX, BulletY, BulletS, Bullet2X, Bullet2Y, Bullet2S, Bullet3X, Bullet3Y, Bullet3S, BULLETFLAGCHECK

Purpose: This module is our masterpiece. An absolute ode to the world of design. Our brainchild and our proudest innovation. Spanning a whopping 1,272 lines, it handles all the logic for tank 1's movement via the WASD and spacebar keys alongside handling the production and direction of 3 bullets as discussed in the "Implementation" section of our report.

3. ball2.sv:



Module: tank2

Inputs: Reset, frame_clk, fast_clock, tank2gaya,

[7:0] keycode, keycode1, keycode2, keycode3,

[3:0] count

Outputs: bullet_flag, bullet2_flag, bullet3_flag,

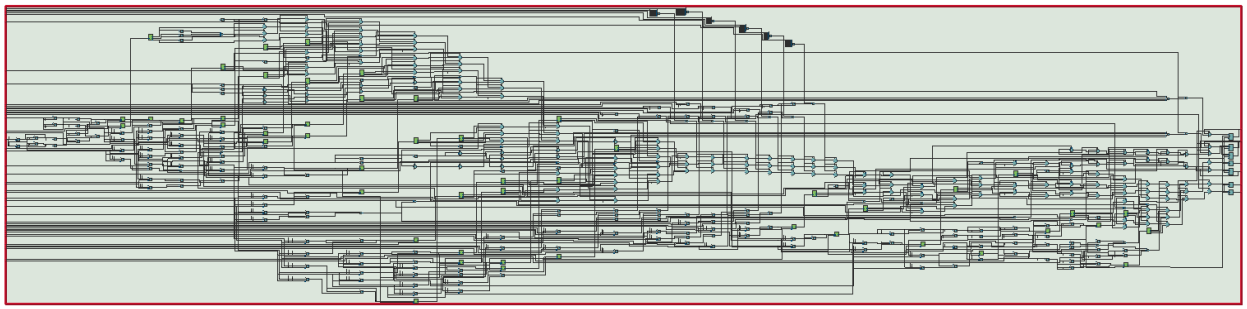
tank2_direction,

[9:0] tankX, tankY, tankS,

[9:0] BulletX, BulletY, BulletS, Bullet2X, Bullet2Y, Bullet2S, Bullet3X, Bullet3Y, Bullet3S, BULLETFLAGCHECK

Purpose: Handles all the logic for tank 2's movement via the arrow keys and enter key alongside handling the production and direction of 3 bullets as discussed in the "Implementation" section of our report.

4. Color_Mapper.sv:



Module: color_mapper

Inputs:

input [9:0] DrawX, DrawY,

input [9:0] T1Bullet1X, T1Bullet1Y, T1Bullet1S, T1bullet1flag,
input [9:0] T1Bullet2X, T1Bullet2Y, T1Bullet2S, T1bullet2flag,
input [9:0] T1Bullet3X, T1Bullet3Y, T1Bullet3S, T1bullet3flag,
input [9:0] T2Bullet1X, T2Bullet1Y, T2Bullet1S, T2bullet1flag,
input [9:0] T2Bullet2X, T2Bullet2Y, T2Bullet2S, T2bullet2flag,
input [9:0] T2Bullet3X, T2Bullet3Y, T2Bullet3S, T2bullet3flag,
input logic vga_clk, blank, game_on, explosion1_on, explosion2_on, death_screenP1, death_screenP2,
input logic tank1spritedetector, tank2spritedetector,
input logic tank1gaya, tank2gaya,
input logic [5:0] healthoftank1, healthoftank2,
input [1:0] tank1_Direction, tank2_Direction,
input [9:0] tankX, tankY, tank_size,
input [9:0] tank2X, tank2Y, tank2_size,
input logic [2:0] CM_sprite_signal,
input logic [19:0] randcoordwire,
input logic obstacle_EN, reset

Outputs:

output logic [7:0] Red, Green, Blue,
output logic tank1shieldacquired, tank2shieldacquired

Purpose: One of the most important modules in our design, spanning a whopping 1,567 lines and handling all the logic for the production of output [3:0] Red, Green and Blue: the 4 bit values producing the color for our pixels on the screen via the VGA_controller module through a connection in the finalproject module. A basic example of how drawing is handled is as follows: the difference between tank 1's X and Y positions and the DrawX/DrawY positions indicating the position of the electron gun is used to determine whether or not you are within tank 1's position

on the screen. If this is true, then the tank1_on signal goes high (actually tank1_up, tank1_down, tank1_left or tank1_right depending on what direction is being received from tank1) and tank1 is drawn on the screen. Certain other sprites are drawn only when a combination of sprite_on and another flag are high, such as in the case of the bullets where a variable like T1bulletflag provides the fire signal to begin the actual rendering of the bullet. The clock driving these assignments is VGA_Clk, which allows for smooth screen rendering. All drawing logic is encapsulated within an **if (blank)** statement and levels of conditional logic check whether to draw the game contents, death screens or menus (all covered within "Implementation"). This module also produces tank1shieldacquired and tank2shieldacquired by checking for the color based collision between either tank and the shield sprite and sends it to the top level for use by other modules.

5. HexDriver.sv:

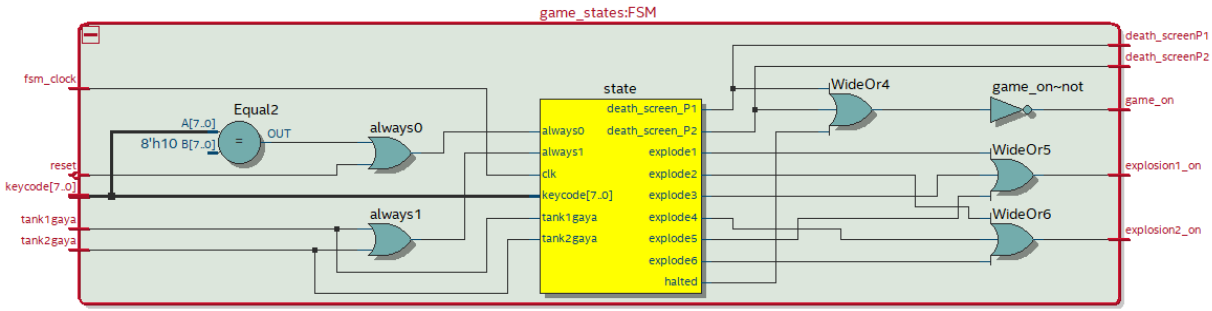
   Module: HexDriver

   Inputs: [3:0] In0

   Outputs: [6:0] Out0

   Purpose: This module contains case statements to make initiations that show how each hex value in this module corresponds to the hex value which would be seen on the DE-10 board.

6. Gamestatemachine.sv:



Module: game_states

Inputs:

input tank1gaya, tank2gaya, fsm_clock, reset,

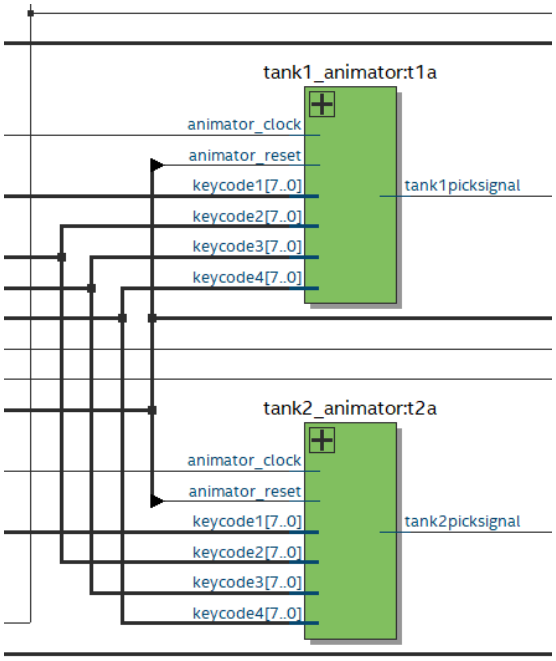
input [7:0] keycode

Outputs:

output game_on, explosion1_on, explosion2_on, death_screenP1, death_screenP2

Purpose: As explained in "Implementation", this module determines the state of the game and cycles between screens depending on collision signals or various keycode inputs: M for menu, Y for restarting after a death, Enter for starting the game from the menu screen, and O for resetting the tanks' positions, bullets and health bars.

7. Animator_fsms.sv:



Module: tank1_animator

Inputs:

input logic animator_clock, animator_reset,

input logic [7:0] keycode1, keycode2, keycode3, keycode4

Outputs:

output logic tank1picksignal

Purpose: Cycles between tank 1 sprites as explained in “Implementation”.

Module: tank2_animator

Inputs:

input logic animator_clock, animator_reset,

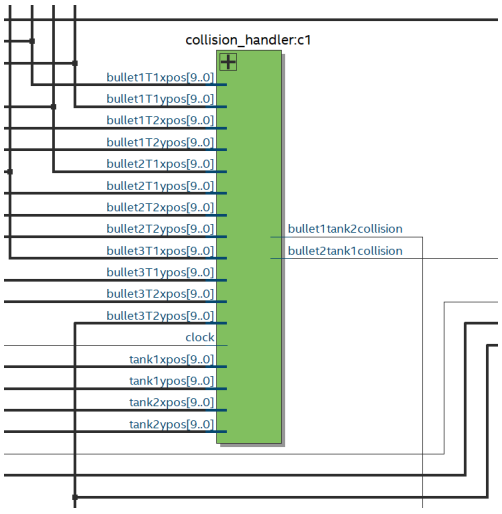
input logic [7:0] keycode1, keycode2, keycode3, keycode4

Outputs:

output logic tank2picksignal

Purpose: Cycles between tank 2 sprites as explained in “Implementation”.

8. collision_handler.sv:



Module: collision_handler

Inputs:

input logic [9:0] bullet1T1xpos, bullet1T1ypos, bullet2T1xpos, bullet2T1ypos, bullet3T1xpos, bullet3T1ypos, bullet1T2xpos, bullet1T2ypos, bullet2T2xpos, bullet2T2ypos, bullet3T2xpos, bullet3T2ypos, tank1xpos, tank1ypos, tank2xpos, tank2ypos,

input logic clock

Outputs:

output bullet1tank2collision, bullet2tank1collision

Purpose: Handles collisions by checking if bullets are within bounds of tanks. High accuracy and responsible for triggering various signals via the top level.

9. clock_divider.sv:

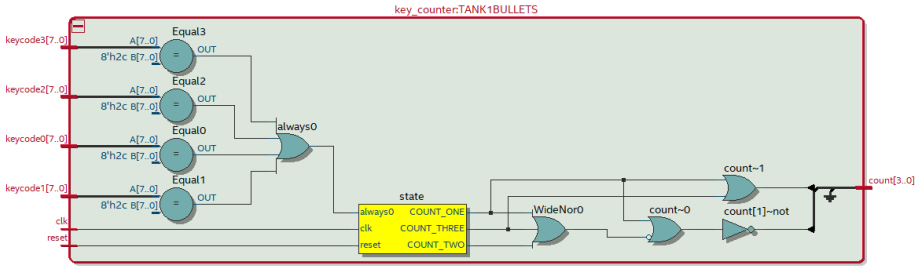
   Module:

   Inputs:

   Outputs:

   Purpose: Produces a slow clock as explained in “Implementation”.

10. bulletcounter.sv:



Module: key_counter

Inputs:

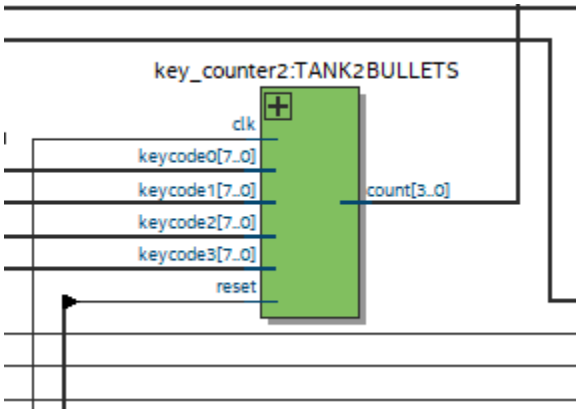
input clk, reset,

input [7:0] keycode0, keycode1, keycode2, keycode3

Outputs:

output [3:0] count

Purpose: Cycles between counts from 1 to 3 for indicating which bullet to fire as explained in “Implementation”, depending on the spacebar key for tank 1.



Module: key_counter2

Inputs:

input clk, reset,

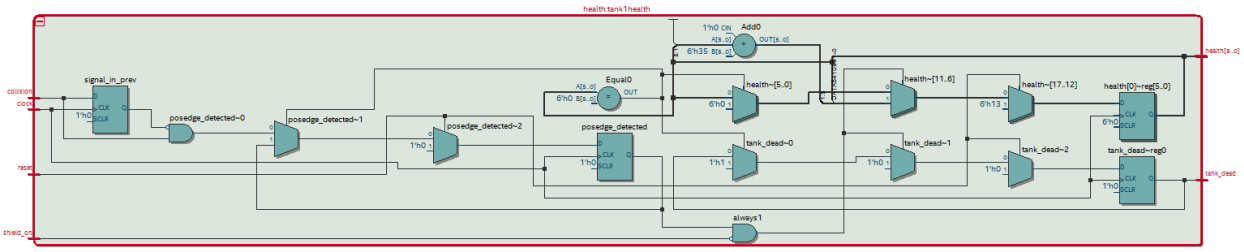
input [7:0] keycode0, keycode1, keycode2, keycode3

Outputs:

output [3:0] count

Purpose: Cycles between counts from 1 to 3 for indicating which bullet to fire as explained in “Implementation”, depending on the spacebar key for tank 2.

11. health.sv:



Module: health

Inputs:

input collision, clock, reset, shield_on,

Outputs:

output logic [5:0] health,

output tank_dead

Purpose: Controls the health of our tanks depending on the shield status as explained in “Implementation” and produces a tank_dead signal to end the game via the game_state module if a tank’s health falls to 0.

12. shield.sv:

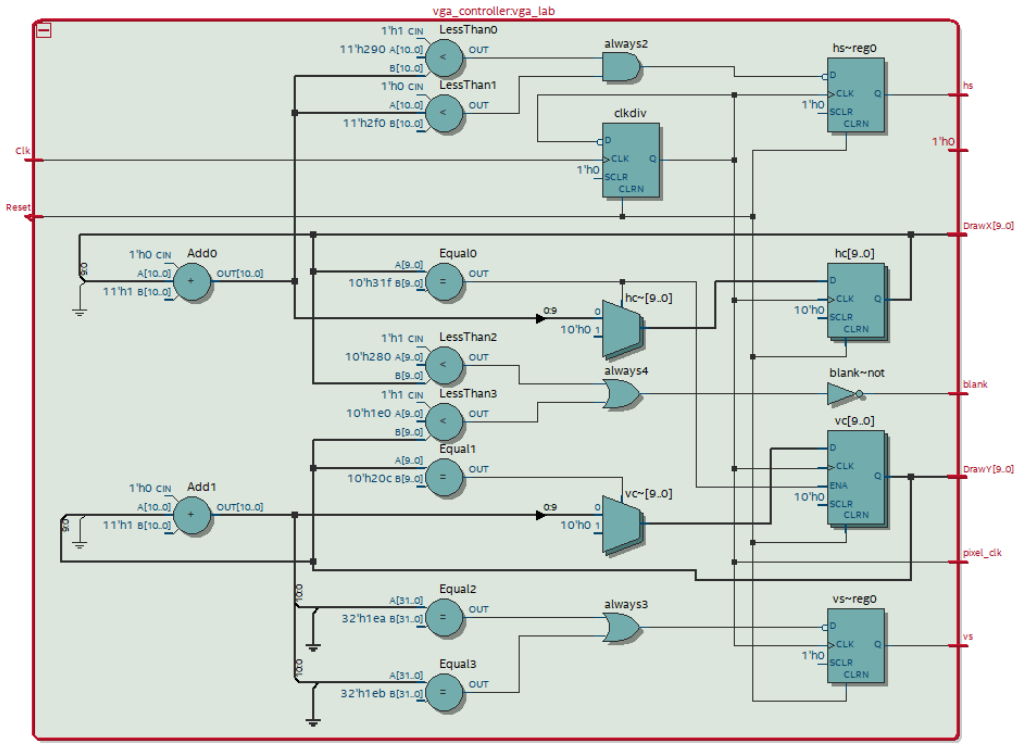
Module: shield

Inputs: reset, clock, shield_acquired

Outputs: shield_status

Purpose: Generates a shield for a tank as explained in “implementation”.

13. VGA_controller.sv:



Module: VGA_controller

Purpose: The purpose of this SV file was to instantiate the functionality of VGA controller along with the specification of the horizontal and vertical sync

Inputs: CLK, Reset

Outputs: Logic [9:0]DrawX, DrawY

Logic hs, vs, blank, sync, pixel_clk

Description: This module is responsible for handling the horizontal and vertical sync of each screen and setting each pixel with the help of each electron beam gun behind the screen.

14. lfsr.sv:

    Module: lfsr

    Inputs: Clk, reset, [19:0] seed

    Outputs: [19:0] out

    Purpose: Produces a 20 bit random number as explained in "Implementation". Several lfsr instantiations chained together in the top level via the seed and out signal in order to generate randomness.

15. lab6_constraints.sdc:

    Purpose: This file sets delays in the outputs for the correct time signals and allows us to generate frequency information in the compilation report.

16. lab62soc.v:

    Purpose: The platform design generates these hardware components, and allows for connection between the hardware, keyboard, and peripherals.

17. Final_project_palette.sv:
    - Final_tank_down_palette.sv
    - Final_tank_up_palette.sv
    - Final_tank_left_palette.sv
    - Final_tank_right_palette.sv
    - Final_tank_down2_palette.sv

- Final_tank_up2_palette.sv
- Final_tank_left2_palette.sv
- Final_tank_right2_palette.sv
- Finalmap_palette.sv
- Mainmenu1_palette.sv
- Bitmap_palette.sv
- P1_win_palette.sv
- P2_win_palette.sv
- explosion2_palatte.sv
- explosion3_palette.sv

Purpose: In our final_project_example file, we used sprites that are associated with specific color palettes. These palettes are defined in the zelda_palette module and each of them contains a 16 x 12 bit array called 'palette'. This array has columns for 'red', 'green', and 'blue' variables that determine the colors of the sprite. By providing an 'index' variable to the module, we can access and loop through the palette array to retrieve the three colors. These logic output values are then used to draw the sprite on the VGA display.

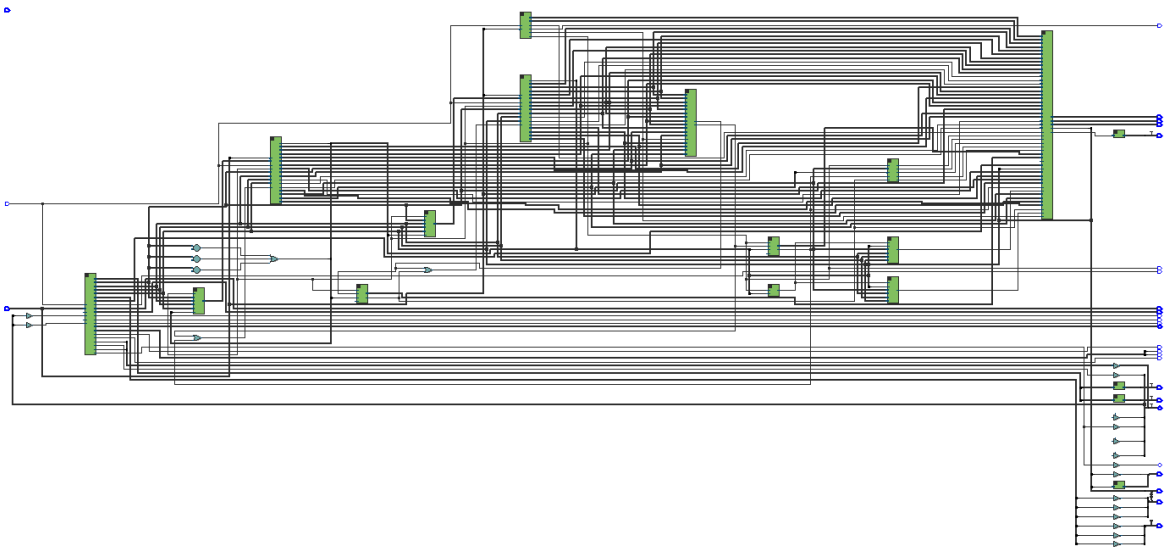
18. Final_project_rom.sv:
    - Final_tank_down_rom.sv
    - Final_tank_up_rom.sv
    - Final_tank_left_rom.sv
    - Final_tank_right_rom.sv
    - Final_tank_down2_rom.sv
    - Final_tank_up2_rom.sv
    - Final_tank_left2_rom.sv
    - Final_tank_right2_rom.sv
    - Finalmap_rom.sv
    - Mainmenu1_rom.sv
    - Bitmap_rom.sv

- P1_win_rom.sv
- P2_win_rom.sv
- Explosion2_rom.sv
- Explosion3_rom.sv

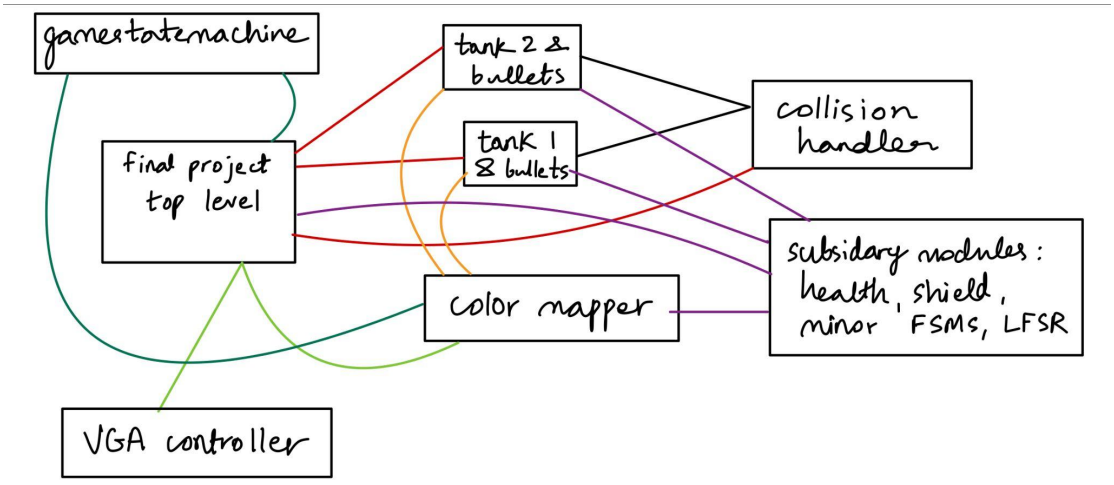
Purpose: The module in question stores the read-only memory (ROM) for each of the sprites used in our game. Each sprite's memory is stored in a memory array and initialized with its own .mif file. The ROMs have a clock input and an address input to access the correct memory location in the array. The output, q, holds the contents of the memory array at the specific address value being accessed, which can then be used in other modules.

Top-level diagram:

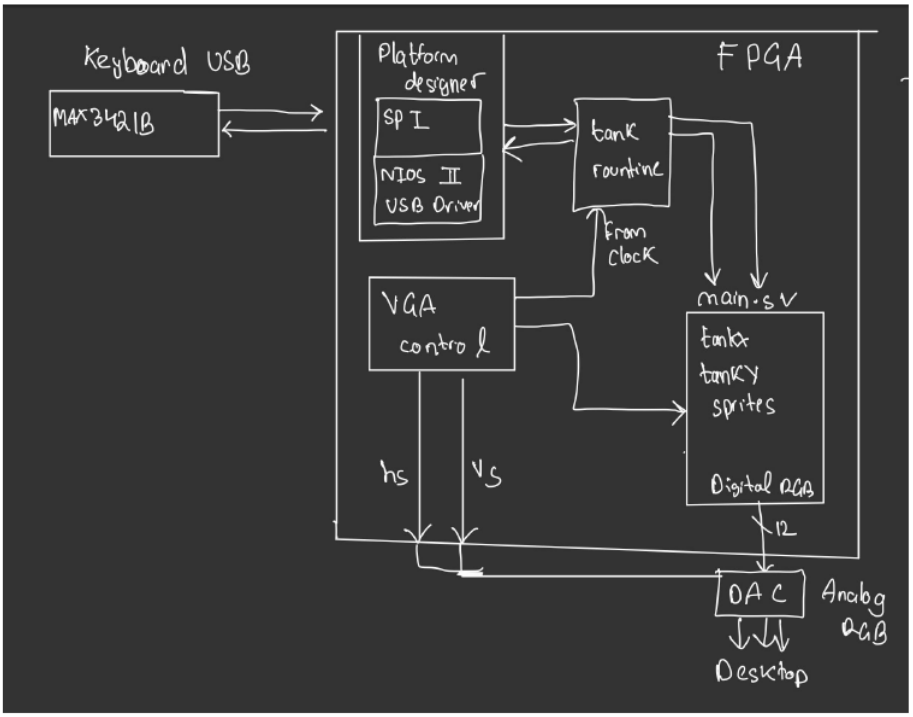
Overview of synthesized “Battle Tanks 1990” hardware:



The modules section of the report contains detailed schematic for individual modules and paints a clearer picture of module connections. For greater legibility, here is a basic overview of connections between main components:



Initial block diagram produced for final project proposal:



Design resource table:

LUT	15,100
DSP	0
Memory (BRAM)	1,079,936
Flip-flop	3,082
Frequency	100.36 MHz
Static power	96.18 mW
Dynamic power	0.86 mW
Total power	106.36 mW

As we can see, our rather large design is actually relatively efficient with a reasonable power consumption and 64% memory usage even after multiple bitmap instantiations. The biggest factor that enabled us to use multiple bitmaps is scaling down the resolution of bitmaps to 200x200 and mapping the proportions of our 640x480 screen to this scaled down version. Furthermore, another major optimization strategy for our bitmaps was using a multiplexer to pick 2 hit points at a time. The following code snippet displays this design:

```
//
   // C1//////C2
   // //////////
   // //////////
   // //////////
   // C3//////C4

//
   assign C1bitmapaddress = (((tankX * 320) / 640) + ((tankY * 240) / 480) * 320);
   assign C2bitmapaddress = ((((tankX + 32) * 320) / 640) + ((tankY * 240) / 480) * 320);
   assign C3bitmapaddress = (((tankX * 320) / 640) + (((tankY + 32) * 240) / 480) * 320 );
   assign C4bitmapaddress = ((((tankX + 32)* 320) / 640) + (((tankY + 32)* 240) / 480) * 320);

   always_comb begin
   if(tank1_direction == 2'b00)
      begin
      mapaddress1 = C1bitmapaddress;
      mapaddress2 = C2bitmapaddress;
      end
   else if(tank1_direction == 2'b01)
      begin
      mapaddress1 = C3bitmapaddress;
      mapaddress2 = C1bitmapaddress;
      end
   else if(tank1_direction == 2'b10)
      begin
      mapaddress1 = C3bitmapaddress;
      mapaddress2 = C4bitmapaddress;
      end
   else
      begin
      mapaddress1 = C4bitmapaddress;
      mapaddress2 = C2bitmapaddress;
      end

   end

   bitmap_rom tank1_collision1(.clock(fast_clock),.address(mapaddress1),.q(collisionsignal1));
   bitmap_rom tank1_collision2(.clock(fast_clock),.address(mapaddress2),.q(collisionsignal2));
```

Apart from this, multiple instantiations and using FSMs to drive signals for multiple modules optimized area and dynamic power usage.

Conclusion:

Through the course of this lab and class we were able to explore and design digital systems. We were able to apply the concepts we learnt during the class in our final project, Battle city. From the application of VGA to designing FSM's of our project, we were able to apply real life concepts into our project. This helped us gain deep insights into good and bad digital designs and in the process it has made us better engineers. We thoroughly enjoyed learning about the intricate details of these systems and how they can be expanded to be applied to solve real life problems through a FPGA. In the process it helped us develop design thinking, teamwork and application skills.

We were given the opportunity to interact with great TA's and CA's along the process which helped us to complete the project in the given time frame. The class as a whole was really well documented but we believe the addition of written quizzes would help the students better understand the concept and the application of those concepts. Along the process we encountered problems but while solving them we learnt more details about the Quartus program.