ECE 385

Fall 2023

Tanks Arcade Game

Final Project

Ashley Herce & Rahul Grover
Lab Section: JW
Lab TA: Jinhua Wang

I. Introduction

To begin, we implemented the design for our game by using a Microblaze processor to handle operations that do not require high-performance such as basic input/output operations written in a higher-level coding language like C. Additionally, the Spartan 7 FPGA we utilized can be programmed to use the Microblaze processor to handle IO as specified by the functions written in C and extend this to include a USB and VGA system to handle more complex operations like drawing the movement/firing of multiple tanks through keyboard input on a HDMI output.

To successfully draw the sprites for our two tanks with appropriate coloring as well as our background image, we used COE files and a color palette to properly link each pixel to the appropriate color fill. These drawn tanks can then be fired with ammunition and move through the map except through the borders and walls. Our hardware logic contains appropriate collision detection by creating hitboxes around the tanks and ammunition to determine if the game needs to be reset/ended or if a player should lose a life.

II. Written Description & Diagrams of Tanks System

Module descriptions:

Following are module descriptions for the top level (not including the modules encapsulated in our MicroBlaze block diagram):

Module: `ball.sv`

Input(s): `Reset, frame_clk, [31:0] keycode, [9:0] DrawX, [9:0] DrawY,`

Output(s): `[9:0] Tank1_X, [9:0] Tank1_Y, [9:0] Tank2_X, [9:0] Tank2_Y, [12:0] offset, [12:0] offset2, [9:0] missile1_X,[9:0] missile1_Y, [9:0] missile2_X, [9:0] missile2_Y, missile1_on, missile2_on, [2:0] player1Life, [2:0] player2Life, [12:0] life_1_offset, [12:0] life_2_offset, game_over, [2:0] winner`

Description: The module declares multiple local variables which represent the missile and tanks position/point (center, leftmost, rightmost, topmost, bottommost) on the x and y axis and the missile and tanks size on the x and y axis as well as the size. In an `always_ff` block, the module will reset the game when the `Reset` signal is high to its default state where both tanks are in the top corners.It will also reset the game when the `game_over` bit is high. The `game_over` bit is set based on if life count variables for player 1 or player 2 are above 3200 bits. Otherwise, it will allow the two players to fire and move their tanks based on where it is on the screen and change directions based on conditional `keycode` options. Then the module will update the game with each tank's movement being represented on the screen based on the motion in the x and y direction and current coordinates of everything.

Purpose: The module serves as the control for how the tanks move on the screen and allows us to detect collisions as well as set the default position of everything when the game resets. The game handles multiple conditional cases of keycodes. We utilize 16 of the 32 available keycodes to handle movements of each player using the WASD and JIKL keys as well as Q and U for firing. Based of the conditional keycodes available and the conditionals used we can only handle firing when players are both stationary.

Module: `Color_Mapper.sv`

Input(s): `[9:0] Tank1_X, [9:0] Tank1_Y, [9:0] Tank2_X, [9:0] Tank2_Y, [9:0] DrawX, [9:0] DrawY, clk, [12:0] offset, [12:0] offset2,[9:0] missile1_X,[9:0] missile1_Y, [9:0] missile2_X, [9:0] missile2_Y, [12:0] life_1_offset, [12:0] life_2_offset, missile1_on, missile2_on,`

Output(s): `[3:0] Red, [3:0] Green, [3:0] Blue`

Description: The module serves to determine multiple aspects of the game as listed: the current life for each tank and update the life counter to reflect a collision for each tank, update the position of each tank or missile fired from each tank, and finally update the background/map. It does so by parsing through each input to its corresponding rom or palette and then mapping the appropriate color to each pixel depending on where each instance (tanks/missiles/heart counter/map) is to accurately reflect a chance. The associated `Red`, `Green`, and `Blue` values are updated accordingly based on what is output by each rom or color mapper instance for each sprite. Conditionals determine which pixel is drawn from the corresponding rom, if the draw missile, background, life count or tank are high at the corresponding pixel then it is drawn to the screen in raster order. The module also instantiates the palettes for background, tank and life counts and accesses the IP's associated.

Purpose: The module serves to determine the color of each pixel on the screen and map each appropriate instance to its corresponding color palette to allow a background and foreground color for each tank, missile, heart counter (the life counter for each tank starts at 3 hearts), and the overall map which is the remainder of the screen.

Module: `tank_rom.xci`

Input(s): `addra[12:0], clka`

Output(s): `douta[1:0]`

Description: The XCI contains the parameters to apply the source code for the tank1_rom

Purpose: It generates an IP which initializes the COE file to generate a tank image. The tank image is what is used for our players, it holds a 40x1600 image, we access only 40x40 of the image at once allowing our player's image to change orientation when a keycode is pressed. The IP is used for both player1 and player 2

Module: `tank1_life.xci`

Input(s): `addra[11:0], clka`

Output(s): `douta[0:0]`

Description: The XCI contains the parameters to apply the source code for the tank1_life

Purpose: It generates an IP which initializes the COE file to generate an on VGA screen image of a life count consisting of 3 hearts. We access 60x20 image and output the associated color to draw to the screen either a 0 or a 1 to represent a red heart or the white background

Module: tank2_life_rom.xci

Input(s): `addra[11:0], clka`

Output(s): `douta[0:0]`

Description: The XCI contains the parameters to apply the source code for the tank1_life

Purpose: It generates an IP which initializes the COE file to generate an on VGA screen image of a life count consisting of 3 hearts. We access 60x20 image and output the associated color to draw to the screen either a 0 or a 1 to represent a red heart or the white background

Module: `background_rom.xci`

Input(s): `addra[18:0], clka, dina[0:0]`

Output(s): `douta[0:0]`

Description: The XCI contains the parameters to apply the source code for the background

Purpose: It generates an IP which initializes the COE file to generate an on VGA screen image of a background

Module: `HexDriver (hex.sv)`

Input(s): `clk, reset, [3:0] in[4]`

Output(s): `[7:0] hex_seg, [3:0] hex_grid`

Description: The module is designed to output a 4-digit 7-segment display for hexadecimal values. It takes input as a 4-bit nibble value `in[4]` which represents a hex digit and drives the corresponding segments of the seven-segment display to display the digit sequentially.

Purpose: The hex unit displays the content of register A and register B in hexadecimal

Module: `nibble_to_hex (hex.sv)`

Input(s): `[3:0] nibble`

Output(s): `[7:0] hex`

Description: The module uses a combination logic block `always_comb` to convert 4 bit binary `nibble` input to 8 bit `hex`. Additionally it resets the `hex` display in an `always_ff` block when reset is high

Purpose: The module is designed to convert a 4 bit binary nibble to 8-bit hexadecimal representation. The module is used to display the hexadecimal values on our hex displays on the FPGA.

Module: `mb_usb.bd (not a .sv file it is a block diagram of our MicroBlaze architecture)`

Input(s): `clk_100Mhz, gpio_usb_int_tri_i, reset_rtl_0, uart_rtl_0_rxd, usb_spi_miso`

Output(s): `gpio_usb_keycode_0_tri_o, gpio_usb_keycode_1_tri_o, gpio_usb_rst_tri_o, uart_rtl_0_txd, usb_spi_mosi, usb_spi_sclk, usb_spi_ss`

Description: This is the MicroBlaze block design final that instantiates the MicroBlaze soft processor and all the required peripherals.

Purpose: It implements the full MicroBlaze system on chip including processor, memories, peripherals, interconnect and I/O.

Module: `clk_wiz_0.xci (not a .sv file it is an IP)`

Input(s): `clk_in1, reset`

Output(s): `locked, clk_out1, clk_out2`

Description: The XCI contains the parameters to apply the source code for the clk_wiz

Purpose: It generates the various clock frequencies like 100MHz system clock which is required for HDMI

Module: `VGA_controller.sv`

Input(s): `pixel_clk, reset`

Output(s): `hs, vs, active_nblank, [9:0] drawX,` `[9:0] drawY`

Description: The module handles multiple syncs of the vertical and horizontal settings based on `DrawY` and `DrawX` using `always_ff` blocks on the `pixel_clk` and `reset` signals. It will run the horizontal counter when the reset vertical counter is incremented. Horizontal sync pulse is 96 pixels long and signal is registered to ensure clean output waveform while vertical sync pulse is 2 lines long . It will only display pixels between horizontal 0-639 and vertical 0-479 and the signal is registered within the DAC chip (combinational logic)

Purpose: The module handles vertical and horizontal sync when setting a DrawyY and drawX. The VGA_controller module is design to draw the ball on the screen and termine the horizontal and vertical movement to draw the pixels accordingly

Module: `hdmi_tx_0.xci (not a .sv file it is an IP)`

Input(s): `ade, [3:0] aux0_din, [3:0] aux1_din, [3:0] aux2_din, [3:0] blue, [3:0] green, [3:0] red, hsync, pix_clk, pix_clk_locked, pix_clkx5, rst, vde, vsync`

Output(s): `TMDS_CLK_N, TMDS_CLK_P, [2:0] TMDS_DATA_N, [2:0] TMDS_DATA_P`

Description: The xci is an instance of the HDMI IP core

Purpose: It handles conversion of the VGA signals generated by the FPGA logic into HDMI format for output over the HDMI port

Module: `background_palatte.sv`

Input(s): `index`

Output(s): `[3:0] red, [3:0] green, [3:0] blue`

Description: Based on the current `index` inputted to the palette, the correct color (either white or pastel blue) is selected to color the background of the game. Based on the index, the `red`, `green`, and `blue` values are assigned a specific value which we selected for our colors.

Purpose: This module serves as a color lookup for the background ROM which allows us to create the map of our game.

Module: `tank_palatte.sv`

Input(s): `[1:0] index`

Output(s): `[3:0] red, [3:0] green, [3:0] blue`

Description: Based on the current `index` inputted to the palette, the correct color is selected to color player one's tank of the game. Based on the index, the `red`, `green`, and `blue` values are assigned a specific value which we selected for our colors.

Purpose: This module serves as a color lookup for the tank1 ROM which allows us to create player one's tank in our game.

Module: `tank2_palatte (tank_palatte.sv)`

Input(s): `[1:0] index`

Output(s): `[3:0] red, [3:0] green, [3:0] blue`

Description: Based on the current `index` inputted to the palette, the correct color is selected to color player two's tank of the game. Based on the index, the `red`, `green`, and `blue` values are assigned a specific value which we selected for our colors.

Purpose: This module serves as a color lookup for the tank2 ROM which allows us to create player two's tank in our game.

Module: `tank1_life_palatte (tank_palatte.sv)`

Input(s): `[1:0] index`

Output(s): `[3:0] red, [3:0] green, [3:0] blue`

Description: Based on the current `index` inputted to the palette, the correct color is selected to color player one's life display of the game. Based on the index, the `red`, `green`, and `blue` values are assigned a specific value which we selected for our colors.

Purpose: This module serves as a color lookup for the tank1 life ROM which allows us to create player one's life display in our game.

Module: `tank2_life_palette (tank_palatte.sv)`

Input(s): `[1:0] index`

Output(s): `[3:0] red, [3:0] green, [3:0] blue`

Description: Based on the current `index` inputted to the palette, the correct color is selected to color player two's life display of the game. Based on the index, the `red`, `green`, and `blue` values are assigned a specific value which we selected for our colors.

Purpose: This module serves as a color lookup for the tank2 life ROM which allows us to create player two's life display in our game.

Module: `tank_rom.sv`

Input(s): `clock, [10:0] address`

Output(s): `[1:0] q`

Description: Based on the `[10:0] address` inputted to the ROM, the corresponding value stored at the memory location specified by that address is stored in `[1:0] q`, which is set on the posedge of the `clock` signal.

Purpose: The purpose of this file is to get a value stored in the block memory at a specified address at each posedge of the clock cycle.

Module: `tanks_top_level.sv`

Input(s): `Clk, reset_rtl_0, gpio_usb_int_tri_i, usb_spi_miso, uart_rtl_0_rxd`

Output(s): `gpio_usb_rst_tri_o, usb_spi_mosi, usb_spi_sclk, usb_spi_ss, uart_rtl_0_txd, hdmi_tmds_clk_n, hdmi_tmds_clk_p, [2:0] hdmi_tmds_data_n, [2:0] hdmi_tmds_data_p, [7:0] hex_segA, [3:0] hex_gridA, [7:0] hex_segB, [3:0] hex_gridB`

Description: The module instantiates all blocks of the hardware design including `HexDriver, mb_block_i, clk_wiz, vga_controller, hdmi_tx_0, ball and color_mapper.`

Purpose: The module serves as our top-level design and takes in input and instantiates our design modules.

Following are module descriptions for our block diagram of the implemented Microblaze architecture (mb_usb):

Module: `microblaze_0_0 (MicroBlaze)`

Input(s): `Interrupt, Debug, Clk, Reset`

Output(s): `DLMB, ILMB, M_AXI_DP`

Description: The microblaze_0 is the soft processor IP core

Purpose: The module is the MicroBlaze soft processor (CPU) There are multiple preset configurations such as the Microcontroller, real time processor and application processor (the lab uses microcontroller). There are 32-bit instruction sets and general purpose registers and a 32-bit address bus.

Module: `mdm_1_0 (MicroBlaze Debug Module)`

Input(s): `N/A`

Output(s): `MBDEBUG_0, Debug_SYS_Rst`

Description: the mdm_1 is a MicroBlaze Debug Module

Purpose: The mdm_1 provides debug access to the MicroBlaze system. It supports software debugging totals up to 32 MicroBlaze processors and support for synchronized control of multiple MicroBlaze processors. It has configurable software access to debug functionality through the AXI Lite interface.

Module: `clk_wiz_1_0 (Clocking Wizard)`

Input(s): `clk_in1 (clk_100MHz), Reset_ah`

Output(s): `clk_25Mhz, clk_125MHz, locked`

Description: The clk_wiz_0 is a Clocking Wizard IP block.

Purpose: Generates the 100MHz system clock and other clocks like for HDMI from the single 100MHz input.

Module: `microblaze_0_local_memory`

Input(s): `DLMB, ILMB, LMB_Clk, SYS_Rst`

Output(s): N/A

Description: The microblaze_0_local_memory is a Block RAM IP configured as data and instruction memory.

Purpose: Provides program and data storage for the MicroBlaze processor.

Module: `rst_clk_wiz_1_100M (Processor System Reset)`

Input(s): `slowest_sync_clk, ext_reset_in (reset_rtl_0), mb_debug_sys)rst, dcm_locked`

Output(s): `mb_reset, bus_struct_reset[0:0], interconnect_aresetn[0:0], peripheral_aresetn[0:0]`

Description: The rst_clk_wiz_1_100M generates a reset signal synchronized to the 100MHz clock.

Purpose: Resets the MicroBlaze system synchronously to avoid timing issues.

Module: `microblaze_0_axi_intc (AXI Interrupt Controller)`

Input(s): `s_axi, s_axi_aclk, s_axi_aresetn, intr[3:0], processor_clk,processor_rest`

Output(s): `interrupt`

Description:The microblaze_0_axi_intc is an Interrupt Controller IP block

Purpose: The interrupt controller IP processes interrupts from peripherals and informs the MicroBlaze of interrupt requests

Module: `microblaze_0_axi_periph (AXI Interconnect)`

Input(s): `S00_AXI, ACLK, ARESETN, S00_ACLK, M00_ACLK,M00_ARESTN, M01_ACLK, MO1_ARESETN, M02_ACLK, M02_ARESETN, MO3_ALCK, M03_ARESETN, MO4_ARESETN, MO4_ALCK, MO5_ACLK, M05_ARESETN, M06_ALCK, MO6_ARESETN`

Output(s): M00_AXI, M01_AXI, M02_AXI, M03_AXI, M04_AXI, M05_AXI, M06_AXI

Description: The microblaze_0_axi_periph is an AXI Interconnect block.

Purpose: The AXI interconnect connects various AXI peripheral buses to the master AXI bus from the MicroBlaze.

Module: `timer_usb_axi (AXI Timer)`

Input(s): `S_AXI, s_axi_aclk, s_axi_aresetn`

Output(s): `interrupt`

Description: The timer_usb_axi is an AXI Timer peripheral.

Purpose: The peripheral is used to implement timeouts needed for the USB protocol in software, where the MicroBlaze can use it to count elapsed time.

Module: `microblaze_0_xlconcat (Concat)`

Input(s): `timer_usb_axi interrupt, axi_uartlite_0 interrupt, spi_usb interrupt (ip2intc_irpt), gpio_usb_int interrupt (ip2intc_irpt)`

Output(s): `dout[3:0] (connected to intr[3:0])`

Description: The microblaze_0_xlconcat is a concatenation IP block.

Purpose: The IP concatenates the interrupt signals from the multiple peripheral inputs into a single bus to connect to the interrupt controller.

Module: `axi_uartlite_0 (AXI Uartlite)`

Input(s): `S_AXI, s_axi_aclk, s_axi_aresetn`

Output(s): `UART (uart_rtl_0) , interrupt`

Description: The axi_uartlite_0 module is a UART Lite peripheral.

Purpose: The UART lite module allows the MicroBlaze to send/receive data over the serial port to the PC and is used for printf debugging. The main uses for it is to connect the Microcontroller Bus Architecture to the AXI and provide asynchronous serial data transfer.

Module: `gpio_usb_rst (AXI GPIO)`

Input(s): `S_AXI, a_axi_aclk, s_axi_aresetn`

Output(s): `GPIO (gpio_usb_rst)`

Description: The gpio_usb_rst is an AXI GPIO module configured as a 1-bit output.

Purpose: Generates the reset signal for the MAX3421E USB controller so the MicroBlaze can reset it as needed.

Module: `gpio_usb_int (AXI GPIO)`

Input(s): `S_AXI, a_axi_aclk, s_axi_aresetn`

Output(s): `GPIO(gpio_usb_int), ip2intc_irpt`

Description: The gpio_usb_int is an AXI GPIO module configured as a 1-bit input.

Purpose: 1 bit input used to receive the interrupt signal from the MAX3421E. This allows the MAX3421E to notify the MicroBlaze of events like USB enumeration or received data

Module: `gpio_usb_keycode (AXI GPIO)`

Input(s): `S_AXI, a_axi_aclk, s_axi_aresetn`

Output(s): GPIO (gpio_usb_keycode_0), GPIO2 (gpio_usb_keycocde_1)

Description: The gpio_usb_keycode is an AXI GPIO module configured as a dual 32-bit channel output.

Purpose: The AXI GPIO is used to send the USB keycode from the MicroBlaze to the top level so it can be displayed on the serial console for debugging

Module: `spi_usb (AXI Quad SPI)`

Input(s): `AXI_LITE, ext_spi_clk, s_axi_aclk, s_axi_aresetn`

Output(s): `sck_o (usb_spi_sclk) , ss_o[0:0] (usb_spi_ss),io0_o, io1_i, ip2intc_irpt`

Description: The spi_usb module instantiates the AXI Quad SPI IP core and connects it to the I/O pins to communicate with the MAX3421E USB controller chip. It handles the low-level SPI protocol and timing requirements.

Purpose: provide the SPI communication interface between the Microblaze processor and the MAX3421E USB controller chip. It connects the SPI signals (SLCK, MISO, MOSI, and SS) and provides the 4 read and write functions to abstract SPI operations to register and FIFOs on the MAX3421E.

Explanation of Game

Our game utilizes two peripherals, one USB keyboard (MAX3421E) and an HDMI monitor connecting to the Spartan-7 FPGA through ports. The game starts with Player One's tank in the top left and Player Two's tank in the top right. Both tanks are 40x40 pixels drawn facing the left. Tanks can be controlled using the W-A-S-D and

I-J-K-L respectively as arrow keys and fire a missile from their tanks with Q and U. Additionally, both players have a health bar of three hearts that are located at the bottom of the screen; A player loses a heart if their tank gets hit by a missile from the opposing tank. The game ends when a player reaches zero hearts and the corresponding winner (Player 1 or 2) is displayed on the FPGA's LED displays to indicate the game has ended. A 1 is displayed corresponding to player 1 winning and a 2 for player 2 winning. Additionally, a "F" is displayed to the HEX to signal "game over" and the game must be reset to start again. To reset the game back to its original state tBTN0 must be pressed.

Our design utilizes on-chip memory rather than registers which requires us to use C code to handle the way we read/write to the BRAM. We can then access data through the AXI bus on our Microblaze architecture and handle things like drawing out our COE files/heart system with the appropriate designs, accessing keycodes clicked from our USB keyboard, and eventually writing the logic for handling the higher functionality aspects of our game in VHDL. Furthermore, based on the keycodes that are clicked, our file ball.sv contains a large chunk of our game logic that uses these keycodes to determine the direction of the tank and stop it from going through walls or borders. This is done by checking the x and y coordinates of each tank and the direction it is going in based on the keyboard input. A similar implementation exists for the missiles fired by the tanks to ensure all elements of the game are interacting with the 640x480 map appropriately.

Our game consisted of four kinds of collision detected which superseded each other to logically determine which collisions would take priority over others. Based on the current clock cycle, the first thing that would be checked is if a missile from Tank 1 collided with Tank 2 or vice versa and then reduce a life of the corresponding tank. In the instance that both tanks hit each other within the same clock cycle, both players would lose a life. The second type of collision that would be detected is if both tanks collided which would result in both players losing a life. Third, we check if a tank is colliding with a wall in which case the keycode input for movement is ignored unless it is in a direction facing away from the wall. Finally, each tank's current location is checked against the map borders to ensure that they are unable to go off the edge. Each missile's coordinates are also checked to see if they collide with a wall or map border and return the ammunition to the respective tank.

The purpose of our VGA is to paint a screen consisting of a matrix of pixels row by row. For our game, we can create more advanced sprites with our VGA controller by specifying the dimensions and color of the pixels in our BRAM through a COE file. The refresh rate of our screen will be 60 Hz, which means that our VGA will generate a signal 60 times per second to account for any updates given to it. Our VGA controller communicates similarly where it synchronizes all the clocks for our tanks, background, ammunition, and color mapper so that all elements update together. Additionally, the VGA controller and Color Mapper are set to communicate with each other via the drawX and drawY which specify how the objects should be drawn by the pixels as well as what the colors should be. If the tank attempts to move or fire a missile, the Color Mapper receives this information and updates the ball's pixel coordinates accordingly.

III. Top Level Block Diagram/State Diagram

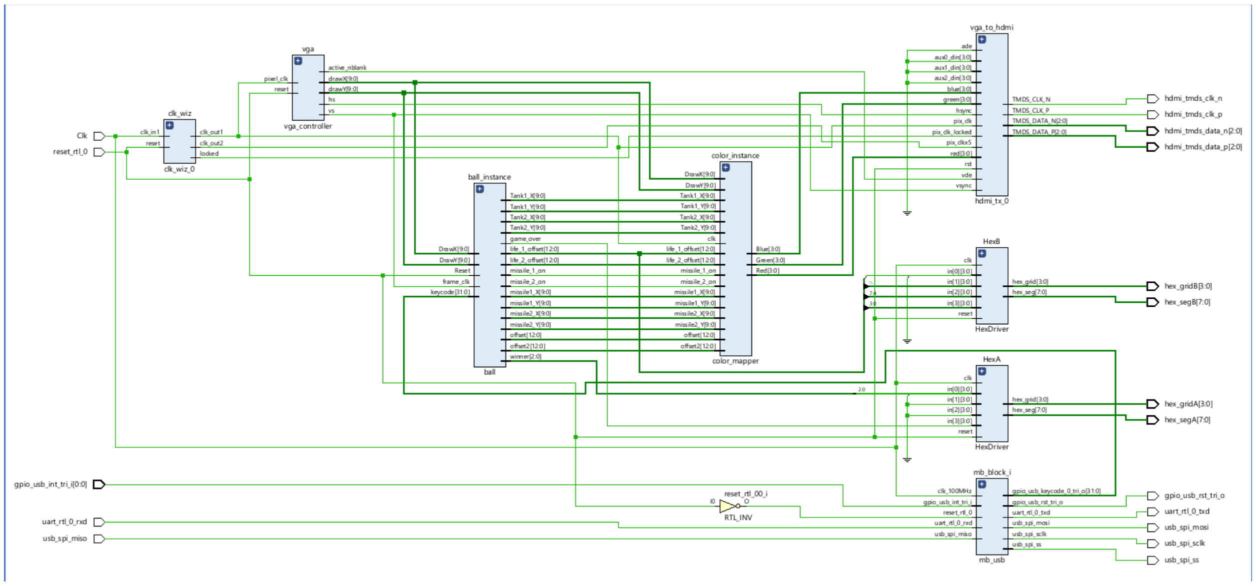
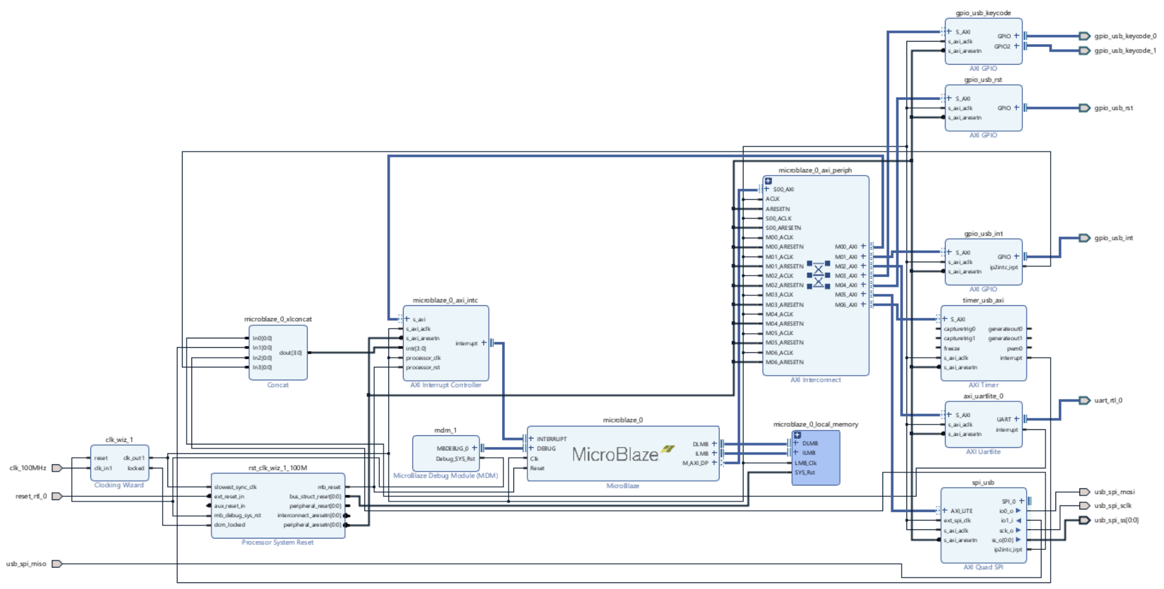


Figure 1: Top Level Block Diagram (Vivado)



Mb_block Diagram (lab 6.2)

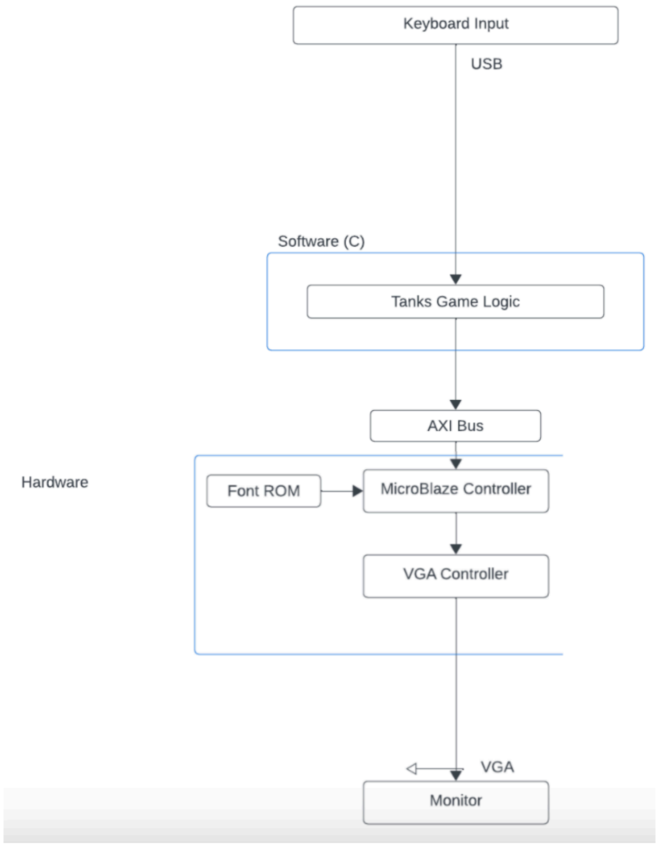


Figure 2: Block Diagram of Tanks Game

IV. Simulation/Waveforms

All testing was done visually using an HDMI monitor and vitis. No simulation waveforms used to test code functionality. Debugging done using hex drivers and debug output during bitstream generation.

V. Software component of the final project

To access the keys that are clicked on the keyboard (MAX3421E), there are four signals called CLK, MOSI, MISO, and SS that are transferred through the USB. The two control signals called MOSI and MISO correspond to master-out slave-in and master-in slave-out respectively. Put simply, this means that they take care of data transmission between the Microblaze and the MAX3421E device. Additionally, all data transmission is synchronously connected to the CLK specified by the Microblaze through the CLK signal in the USB. Finally, the purpose of SS is that it allows you to select a specific slave device when it is a logic zero and must be unique to each device whereas other signals may be shared.

This overall process of transmitting the data is referred to as an SPI protocol which we were required to implement to interact with our peripheral keyboard. As a result, we wrote functions in C that allow us to read/write using the MOSI and MISO signals to the memory storage on the Microblaze architecture on the FPGA. We were able to achieve this by writing four functions that took care of reading or writing a selected value to a specified register in C. A description of each of these functions is given below.

C Code Function Descriptions

void MAXreg_wr(BYTE reg, BYTE val) This function writes a single byte value to a register on the MAX3421E. It selects the MAX3421E chip, writes the register address and value via SPI, reads the return code, and deselects the MAX3421E.

BYTE* MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data) This function writes multiple bytes to sequential registers on the MAX3421E starting from a provided register address. It selects the chip, writes the starting register address, writes the

provided data bytes via SPI, reads the return code, and deselects the chip. It returns a pointer to the memory location after the last written byte.

BYTE MAXreg_rd(BYTE reg)

This function reads a single byte from a register on the MAX3421E. It selects the chip, writes the register address to read, reads the register value via SPI, and deselects the chip. It returns the read byte.

BYTE* MAXbytes_rd(BYTE reg, BYTE nbytes, BYTE* data)

This function reads multiple sequential bytes from the MAX3421E starting at a provided register address. It selects the chip, writes the starting register address, reads the data bytes via SPI, and deselects the chip. It stores the read bytes in the provided data array and returns a pointer to the location after the last read byte.

C functions and files were taken from lab 6.2 as well as mb block design taken from lab 6.2. All C files and additional functions are given and used in the Final project design. Additional code for our project was written in our top level sv, ball.sv and color_mapper.sv.

VI. Documentation

Document & Design Resources and Statistics

LUT	4016/32600
DSP	23/120
WNS (ns)	-1.184
Memory (BRAM)	19.5/75
Flip-Flop	2982/65200
Latches*	0
Frequency	89.413 Mhz

Static Power	0.076 W
Dynamic	0.406 W
Total Power	0.482 W

To calculate Frequency we use the equation: $F\ (Mhz)\ =\ 1000/(T\ -\ WNS)$

Our design utilized on-chip memory so we did not require many LUT or Flip flop elements and power. However, our design handled multiple equations to calculate drawing to the screen resulting in some negative slack but overall the design statistics are relatively positive.

VII. Conclusion

Our game was able to work for the most part, although our collision detection for Player 1 was slightly buggy with the hearts not reducing properly sometimes. However, the latter three types of collision worked and movement and firing for each of the tanks also worked as intended. Additionally, the end game display worked as intended by displaying the correct player and resetting the game to the beginning state was fully functional.

Some improvements we could have made to our game are refining our current collision detection as it was buggy at times by doing some additional testing and seeing where our logic may be flawed. In terms of visual changes, we could have added more intractability to our map such as more walls or a more defined winner/loser message that appears on the HDMI display rather than using the FPGA. Overall, the functionality was at a playable stage by the end of our project and improvements can always be made to make the game more enjoyable.

Improvements to our design could have been made using a state machine to handle multiple keycodes as well as life count and all our features but was not implemented and handled at the start of the game resulting in issues later in the project as elements which could have been handled in an FSM was handled brute force. Elements such as multiplayer design could have been handled in an FSM allowing for firing and movement simultaneously.