ECE 385

Fall 2025

Final Project - Astro Party

Final Project Report
Astro Party

Ram Reddy (ramr3) & Ishaan Desai (idesa3)

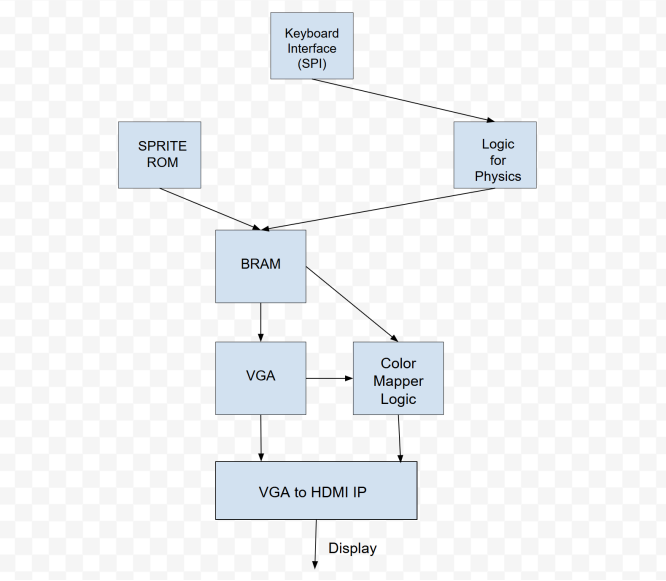
Lab TA: Lily Wan

Introduction

For our final project, we designed and implemented Astro Party, a fast paced two player game on the FPGA. We used the MicroBlaze processor, the MAX3421E USB host controller, and the text-mode graphics controller. The MAX3421E enables keyboard peripheral support for player controls, while the text-mode graphics controller provides visual output to the screen.

In Astro Party, two players are each attempting to eliminate their opponent three times to win the match. Players control their ships using keyboard inputs: WASD keys for Player 1 and arrow keys for Player 2. The W and Up arrow keys move forward in the direction that you are facing. The A, D, Left arrow, and Right arrow keys rotate the ships counterclockwise and clockwise. The S and Down arrow keys fire bullets in the direction the ship is facing. Each ship can have only one bullet active at a time. The game features dynamic sprite rotation using transformation matrices, collision detection between bullets and ships, walls in the arena that make it harder to hit your opponents, as they can block bullets, and a kill counter display showing each player's progress toward victory. We implemented a title screen that appears when the device is programmed, a pause function accessible during gameplay, and a win screen that displays when one player achieves three eliminations. The Enter key is used to start the game from the title screen. The Esc key is used whenever you want to pause the game. The R key is used to advance from the win screen to a rematch. Finally, the Q key is used when you want to advance from the win screen to the title screen to quit the game.

Written Description of Final Project System



Block Diagram for Final Project

1. Overall Project Top Level Design Description

In the Astro Party project, the system is an HDMI graphics controller that is driven by a custom AXI4 peripheral and MicroBlaze. The AXI4 peripheral connects the keyboard, the MicroBlaze, and the HDMI graphics controller all together. The MicroBlaze handles a majority of the game state logic and stores important game information; there are some memory maps registered mapped through the AXI4 bus.The game is interfaced by a keyboard where the MicroBlaze reads input and calculates important game information. This information which is mapped to the AXI can be read by the HDMI graphics controller. The controller uses the calculated information to draw the various sprites and text onto the screen using a color mapper. The sprites are stored in an internal BRAM in the IP.

2. Overall HDMI Graphics IP Description

The HDMI Graphics controller IP is an AXI4 peripheral that produces full graphics on the display through HDMI. The HDMI graphics controller IP contains 20 control registers. Only 15 of the control registers are used as the IP has extra for future possible game features. The registers on the IP are ship positions, bullet positions, player kill stats, game states, and sprite rotation values. On every pixel, the controller uses the DrawX and DrawY signals to determine the pixel to be drawn. Based on the current X and Y values and the game information states in the control registers sprites, game screens, and kill counters are all drawn. In the color mapper based on the DrawX and DrawY signals, sprites, stars, and game screens are drawn. The controller integrates the VGA module, sprite ROM, and VGA to HDMI IP, control registers, and the AXI4 bus to be able to display game graphics on the screen.

3. Game State Implementation Description

The game uses a Finite State Machine in C to control its logic. It switches between Title, Playing, Paused, and Win states by updating a GAME_STATE register. In the Title state, it waits for the "Enter" key to start the game. During Playing, it runs the game loop, handling player movement, bullet management, and collision detection. Collisions trigger actions like increasing kill count or transitioning to the Win state after three kills. This approach ensures real time updates for sprite positions, angles, and sprites. This ensures that game logic with drawing capabilities with efficient and fast rendering.

4. Sprite Implementation Description

The graphical system uses a Color Mapper module that determines pixel data by interfacing with a BRAM which has pre-loaded sprite data via a .coe file. To manage various game elements such as a title screen, player ships, and UI components, we use the color mapper to calculate precise memory address offsets based on the base starting index of each sprite and its local coordinate mapping relative to the current DrawX and DrawY positions. For the player ships, the color mapper incorporates a rotation transformation module that translates sprite location onto coordinates into rotated coordinates, allowing for 360 movement. We also have priority drawing which will draw based on priority of sprites, backgrounds etc. This ensures that objects like bullets and ships are rendered on top of the background stars and walls, while making sure that the color mapper dims the entire scene.

5. Rotation Implementation Description

To incorporate rotations for sprites into the game involved both MicroBlaze and color mapper integration. First, in the MicroBlaze specific rotation angles are saved as part of the game state of each sprite. The angles go from 0-360 degrees where pressing A or ← will subtract 20 degrees from the saved angle rotation and pressing D or → will increase the angle by 20 degrees. The angle wraps around when the angle becomes less than 0 or greater than 360. Then every game calculation cycle in the MicroBlaze which is around every 10 frames of output in the IP, the rotations are saved into the memory mapped registers. Then within the HDMI AXI, the rotation angles are converted to a multiple of 20 from 0-18 as specific angles cannot be calculated in the FPGA. In order to solve this problem we hardcoded possible rotation values from the possible 0-18 rotation states. For each rotation state, let's say for example 20 degrees this is cos(20) = 0.9396 and sin(20) = 0.3420. Then for these calculated angles we multiplied by some power of 2 in order to get a whole number we used 2^14 so cos(20) becomes 15395 and sin(20) becomes 5604. Then for any given X and Y for the sprite pixel we perform the 2d rotation matrix multiplication to get the new location of the X and Y and right shift back by 2^14 in order to get the exact unscaled location. The 2d rotation matrix is [[cos,sin],[-sin,cos]]. Below is example code for the rotation conversion

```
initial begin
    cos_lut[0] = 16'sd16384; sin_lut[0] = 16'sd0;
    cos_lut[1] = 16'sd15395; sin_lut[1] = 16'sd5604;
    ...
end


logic signed [15:0] cos_val, sin_val;
logic signed [26:0] temp_x, temp_y;
logic [4:0] safe_index;
```

```
    always_comb begin
        if (rotation_index >= 18)
            safe_index = 0;
        else
            safe_index = rotation_index;

        cos_val = cos_lut[safe_index];
        sin_val = sin_lut[safe_index];

        temp_x = (dx * cos_val) + (dy * sin_val);
        temp_y = (-dx * sin_val) + (dy * cos_val);

        sprite_x = temp_x >>> FRAC_BITS;
        sprite_y = temp_y >>> FRAC_BITS;

        in_bounds = (sprite_x >= -16 && sprite_x <= 16 &&
                     sprite_y >= -13 && sprite_y <= 13);
    end
```

6. Pausing Game Description

When the ESC key is pressed during gameplay, it transitions to the PAUSED state and updates the game_state control register to 2. While in the PAUSED state, the game logic stops updating, but the ESC key continues to be monitored to be able to resume the game. On the hardware side, when game_state equals 2, the color_mapper applies rendering by drawing two white vertical pause bars at screen center with the highest priority over all other elements. All game elements receive a dimming effect by bit-shifting their color values right by 1. This creates a clear visual indication that the game is frozen while maintaining the visibility of the game.

7. Collision Detection Description

The collision detection code is run on the MicroBlaze system, For each key there is a specific velocity generated by the physics system mentioned in section 10 below. With the newly generated velocities we convert the velocities to displacement and then use the X and Y positions to know if we collided with pre generated walls on the map. If any of the ships hit the walls, the velocities become 0, so the ship will no longer move. If the player wants to move they will need to rotate and gain velocity again. Below is an example code for checking if we are in a wall

```
bool is_in_wall(double x, double y) {
    if (x >= 310.0 && x <= 330.0) {
```

```
        if (y < 180.0 || y >= 300.0) {
            return true;
        }
    }

    if (y >= 230.0 && y <= 250.0) {
        if (x < 260.0 || x >= 380.0) {
            return true;
        }
    }
    return false;
}
```

Another collision detection is for the bullets which check if they are within the hit boxes of the ships. This is very similar to the wall collision detection where we have a box around the ship's current location to detect any bullet hits. If a bullet is hit then we change the game state.

8. Keyboard Implementation Description

The design receives keyboard inputs from a connected keyboard through the MAX3421E component and converts the keycodes of the pressed keys into movement of a ships on the screen, which you can move the ball up, down, left, and right. We use the MicroBlaze along with an on-chip memory block and a parallel I/O block to read and modify memory-mapped values. It is also used to implement the SPI protocol for communication with the USB host controller. the I/O uses the GPIO concepts to let it communicate with the MAX3421E chip, which would automatically receive USB input. By using GPIO, the MicroBlaze is able to send SPI signals using a library that we are given. This allows the system to interface with the USB pins that pass keycodes into the MicroBlaze. These keycodes are stored in registers that can be accessed by the Astro Party C program. The C program reads the first keycode in the list and checks it through a lot of if statements to determine if the ball should move up, down, left, or right for each sprite and each specific mapped key. The next motion of each ship is stored until the ball either hits a wall or a new keypress is detected. Ultimately, the USB chip is continuously polling the keyboard, and those keycodes are constantly being sent through the GPIO via SPI functions that tell the USB driver what to look for.

```
            MAX3421E_Task();
            USB_Task();
            if (GetUsbTaskState() == USB_STATE_RUNNING) {
                if (!runningdebugflag) {
                    runningdebugflag = 1;
                    device = GetDriverandReport();
```

```
            }
            // Keyboard polling
            if (device == 1) {
                rcode = kbdPoll(&kbdbuf);
                if (rcode == 0) {
                    printHex(kbdbuf.keycode[0] + (kbdbuf.keycode[1]<<8) +
                    (kbdbuf.keycode[2]<<16) + (kbdbuf.keycode[3]<<24), 1);
                }
            }
```

9. Physics Implementation Description

In the game, the physics engine runs on the Microblaze part. The ships are moved based on velocity and displacements. We use rotation matrices to convert angles into x and y displacements. We use functions like cos and sin to help calculate movement vectors. Ships have momentum, accelerating when keys are held and slowing down due to friction when released. So ships have a maximum of 10 pixel movement per frame and will slowly decrease over time when W is not pressed. The wall collisions are detected using a four corner system, halting movement on x and y axes independently upon impact. The bullets follow constant velocity motion in a set angle, with directions determined by the ship's rotation angle at firing. We implemented screen wrapping to allow ships to reappear on the opposite side when crossing boundaries. When bullets collide with players it is detected using distance calculations, avoiding square root operations for better performance on the system. Below is the code for bullet physics

```
// update bullet positions
for(int i = 0; i < MAX_BULLETS; i++) {
    if (bullets[i].active) {
        double old_x = bullets[i].x;
        double old_y = bullets[i].y;
        double new_x = old_x + bullets[i].x_vel;
        double new_y = old_y + bullets[i].y_vel;

        if (bullet_hits_wall(old_x, old_y, new_x, new_y)) {
            bullets[i].active = 0;
            continue;
        }

        bullets[i].x = new_x;
```

```
        bullets[i].y = new_y;

        if (is_out_of_bounds(bullets[i].x, bullets[i].y)) {
            bullets[i].active = 0;
        }
    }
}
```

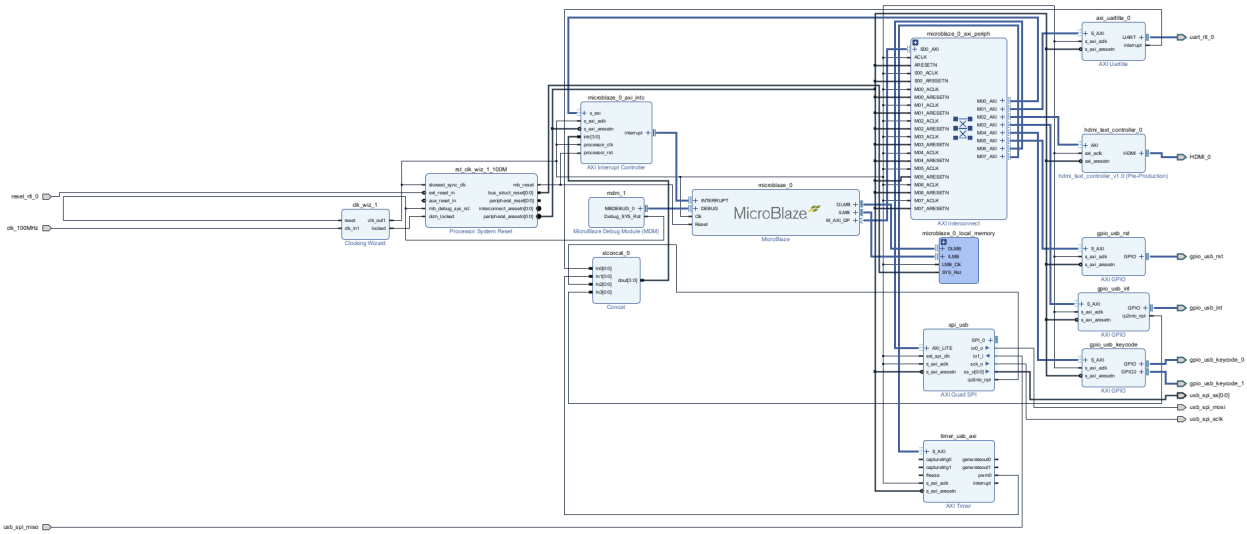
10. Bullets Implementation

The game maintains an array of up to 20 bullets in software, each storing position, velocity, alive status, and the player. When a player presses the fire button, S or Down arrow, it ensures only one bullet spawns at the ship's center with velocity calculated from the ship's rotation angle. Each frame, active bullets move according to their velocity and check for wall collisions. If the bullets hit walls or leave the screen, they get deactivated. Collision detection uses distance calculations between bullet positions and ship centers with a threshold. The software sends the first two active bullets' positions to hardware through the bullet1X, bullet1Y, bullet2X, and bullet2Y control registers. The color mapper module then draws these bullets as small yellow circles using distance calculations, rendering them during both playing and paused states.

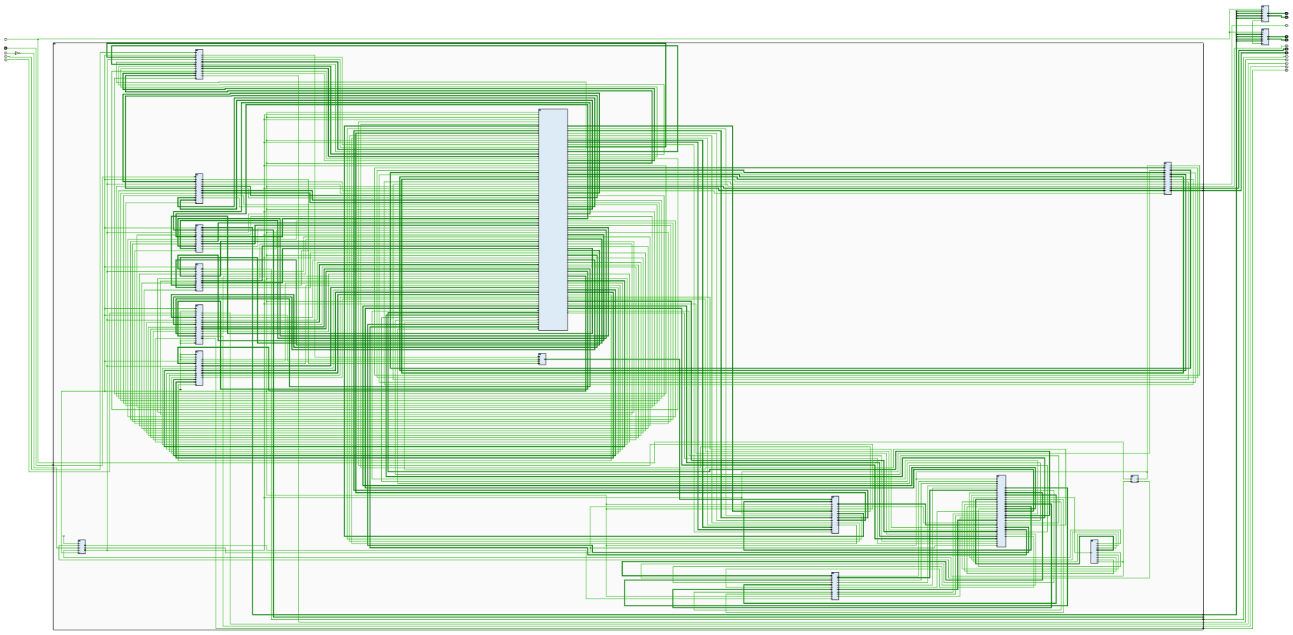
11. Respawn State Implementation

When a player is hit by a bullet, the software sets their dead flag, moves their position off-screen, resets speed to zero, calculates a respawn frame timestamp, and increments the opponent's kill count. All of this information is sent to the hardware. In each frame, the code checks if a dead player's respawn time has arrived by comparing the current frame count to their stored respawn frame. When it reaches that time, it clears the dead flag and resets the position to the respawn location of each player. During the respawn delay, the hardware receives the off-screen coordinates through BallX, BallY, Ball2Y, and Ball2Y control registers, causing the color mapper's boundary checks to fail and preventing the ship sprite from rendering. When the player respawns, and the valid coordinates are sent back to hardware, the ship reappears at its starting position.

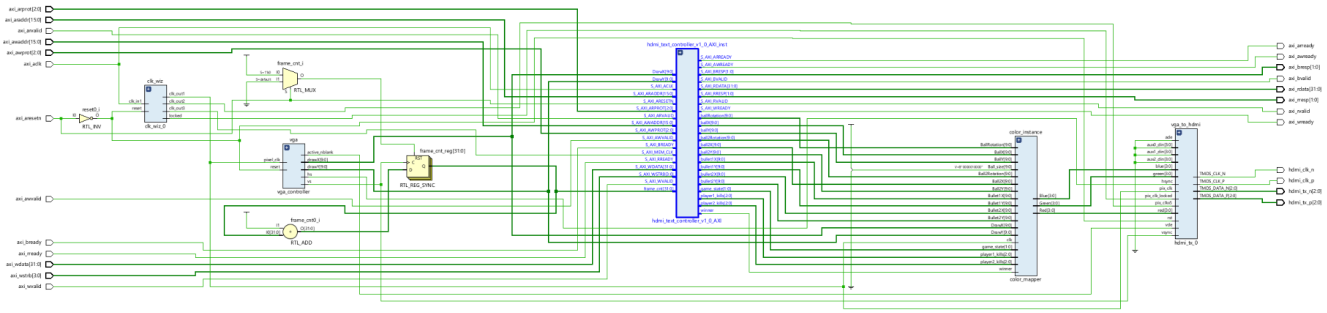
Block Diagram



Block Design Diagram of Astro Party



Top-Level Diagram of Astro Party



Top-Level Diagram of Text-Mode Graphics Controller

Module Descriptions

Module: mb_usb_hdmi_top
Inputs: Clk, reset_rtl_0, [0:0] gpio_usb_int_tri_i, usb_spi_miso, uart_rtl_0_rxd
Outputs: 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: This is the top level of the display processor and is the top level to the entire processor.
Purpose: The purpose of this module is to define all the overarching inputs, outputs, submodules and IP of the microblaze system.

Module: hex_driver
Inputs: clk, reset, in[4]
Outputs: [7:0] hex_seg, [3:0] hex_grid
Description: This module converts the inputs to hex values which can be displayed on the LEDs on the board.
Purpose: This allows for A and B to be displayed on the FPGA board.

Module: vga_controller
Inputs: pixel_clk, reset
Outputs: hs, vs, active_nblank, sync, [9:0] drawX, [9:0] drawY
Description: This module contains a positive edge clock that updates the pixel information on the display.
Purpose: The module is used to correctly update and store the VGA.

Module: color_mapper

Inputs: [9:0] BallX, [9:0] BallY, [9:0] Ball2X, [9:0] Ball2Y, [9:0] Bullet1X, [9:0] Bullet1Y, [9:0] Bullet2X, [9:0] Bullet2Y, [9:0] DrawX, [9:0] DrawY, [9:0] Ball_size, [9:0] BallRotation, [9:0] Ball2Rotation, [1:0] game_state, [2:0] player1_kills, [2:0] player2_kills, winner
Outputs: [3:0] Red, [3:0] Green, [3:0] Blue
Description: This module is the color mapper for determining the RGB output for each pixel based on its coordinates, game state, and object data. It checks whether a pixel overlaps sprites or game objects, fetches sprite data from ROM, applies ship rotation using rotation_transform modules, and filters transparent colors.
Purpose: The module handles different game states, enforces sprite priority, and provides consistent and correct colored graphics for the display.

IP: HDMI/DVI Endocer
Module: hdmi_tx_0
Inputs: pix_clk, pix_clkx5, pix_clk_locked, rst, [3:0] red, [3:0] green, [3:0] blue, hsync, vsync, vde, [3:0] aux0_din, [3:0] aux1_din, [3:0] aux2_din, ade
Outputs: TMDS_CLK_P, TMDS_CLK_N, [2:0] TMDS_DATA_P, [2:0] TMDS_DATA_N
Description: This IP module is used to convert the VGA signal to HDMI.

IP: AXI UART Lite
Module: axi_uartlite_0
Description: The IP module is used to UART interface with AXI4 support which allows interrupts with serial communication.

IP: AXI GPIO
Module: gpb_usb_int, gpb_usb_keycode, gpio_usb_rst
Description: This IP module gives configurable general purpose input and output to the AXI4 which is used to manage external signals and peripherals.

IP: MicroBlaze Processor
Module: microblaze_0
Description: This IP module gives the overarching RISC processor which is specifically optimized for FPGAs. This controls and manages all the other components within the block design.

IP: Microblaze Debug Modules
Module: mdm_1
Description: This IP allows for debugging on the microblaze module which allows for setting debug points and register access.

IP: Microblaze Local Memory

Module: microblaze_0_local_memory
Description: This IP gives local memory to the microblaze which is essentially VRAM. This allows for storage and access of data.

IP: AXI Timer
Module: timer_usb_axi
Description: This IP provides a configurable timer to the AXI interface which can serve as an interrupt or signals within the AXI bus.

IP: AXI Interconnect
Module: microblaze_0_axi_periph
Description: This IP is essentially the interconnect for the AXI bus, which can communicate between the Microblaze and outside peripherals like the GPIO and UART.

IP: Processor System Reset
Module: rst_clk_wiz_1_100M
Description: This IP is intended to give reset signals for the full Microblaze system.This ensures that the powerup and reset sequences are all using the same reset signal.

IP: AXI Interrupt Controller
Module: microblaze_0_axi_intc
Description: This IP is intended so that outside peripherals can let the know Microblaze know that they are ready to be handled. This will control which peripheral gets precedence over others.

IP: Concat
Module: xlconcat_0
Description: This IP module concatenates many interrupt signals into one so that it can be handled as one signal by the interrupt controller.This is useful when there are limited usable pins.

IP: AXI Quad SPI
Module: spi_usb
Description: This IP is the SPI which handles the serial communication between devices. It is the master device that controls the clock and MOSI.

Module: hdmi_text_controller_v1_0
Inputs: axi_aclk, axi_aresetn, [C_AXI_ADDR_WIDTH-1 : 0] axi_awaddr, [2 : 0] axi_awprot, axi_awvalid, [C_AXI_DATA_WIDTH-1 : 0] axi_wdata, [(C_AXI_DATA_WIDTH/8)-1 : 0] axi_wstrb, axi_wvalid, axi_bready, [C_AXI_ADDR_WIDTH-1 : 0] axi_araddr, [2 : 0] axi_arprot, axi_arvalid, axi_rready

Outputs: hdmi_clk_n, hdmi_clk_p, [2:0] hdmi_tx_n, [2:0] hdmi_tx_p, axi_awready, axi_wready, [1 : 0] axi_bresp, axi_bvalid, axi_arready, [C_AXI_DATA_WIDTH-1 : 0] axi_rdata, [1 : 0] axi_rresp, axi_rvalid
Description: This handles all the important instantiations for the IP and outputs the correct graphics and colors on the display using the color mapper module. It also increments the frame count with a positive-edge-triggered clock.
Purpose: This is the top-level module of the IP and instantiates all the important modules like AXI, HDMI, color mapper, etc.

Module: hdmi_text_controller_v1_0_AXI
Inputs: [9:0] DrawX, [9:0] DrawY, [31:0] frame_cnt, S_AXI_ACLK, S_AXI_MEM_CLK, S_AXI_ARESETN, [C_S_AXI_ADDR_WIDTH-1 : 0] S_AXI_AWADDR, [2 : 0] S_AXI_AWPROT, S_AXI_AWVALID, [C_S_AXI_DATA_WIDTH-1 : 0] S_AXI_WDATA, [(C_S_AXI_DATA_WIDTH/8)-1 : 0] S_AXI_WSTRB, S_AXI_WVALID, S_AXI_BREADY, [C_S_AXI_ADDR_WIDTH-1 : 0] S_AXI_ARADDR, [2 : 0] S_AXI_ARPROT, S_AXI_ARVALID, S_AXI_RREADY
Outputs: S_AXI_AWREADY, S_AXI_WREADY, [1 : 0] S_AXI_BRESP, S_AXI_BVALID, S_AXI_ARREADY, [C_S_AXI_DATA_WIDTH-1 : 0] S_AXI_RDATA, [1 : 0] S_AXI_RRESP, S_AXI_RVALID, [9:0] ballX, [9:0] ballY, [9:0] ballRotation, [9:0] ball2X, [9:0] ball2Y, [9:0] ball2Rotation, [9:0] bullet1X, [9:0] bullet1Y, [9:0] bullet2X, [9:0] bullet2Y, [1:0] game_state, [2:0] player1_kills, [2:0] player2_kills, winner
Description: This module takes care of the SPI communication and also all the different states for accessing data. This also takes care of writing to the control registers. This module also instantiates the BRAM so that the color data can be read.
Purpose: This serves as the bus between the master and slave devices. This communicates between SPI between devices.

IP: blk_mem_gen_0
Module: color_pallete_mem
Description: This IP allows the user to use memory outside of the VRAM. The BRAM can store more data than the VRAM can.

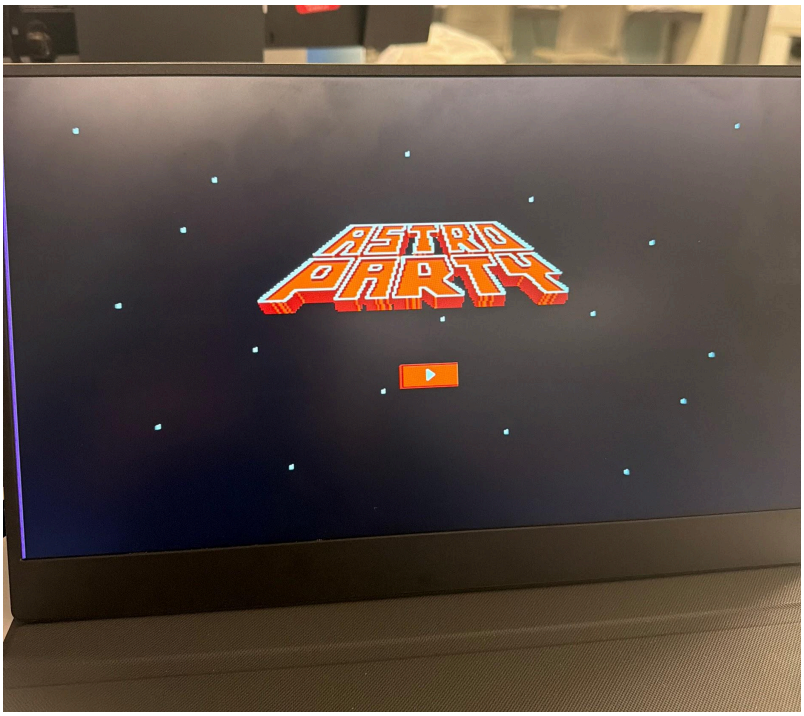
IP: Clocking Wizard
Module: clk_wiz_0
Inputs: clk_in1, reset
Outputs: clk_out1, clk_out2, clk_out3, locked
Description: This IP is used to control the clock signal.

IP: sprite_rom
Module: sprite_memory

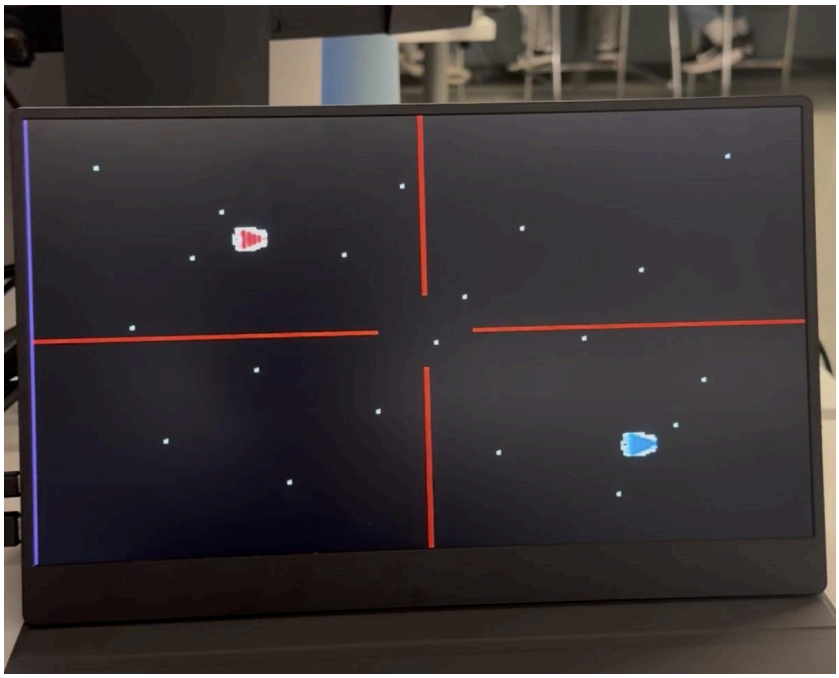
Description: This IP allows the user to use memory outside of the VRAM. This IP stored all of our sprite rom data. It was done by creating a read-only BRAM and loading a coe file with all of the sprite rom data that we needed.

Module: rotation_transform
Inputs: [4:0] rotation_index, [10:0] dx, [10:0] dy
Outputs: [10:0] sprite_x, [10:0] sprite_y, in_bounds
Description: This module rotates a sprite by mapping screen pixels back to the original sprite using an inverse rotation matrix. It accepts a rotation angle and a pixel offset from the sprite center, and uses sine and cosine lookup tables to compute the corresponding sprite coordinates and bounds check.
Purpose: It enables real-time sprite rotation without needing to store multiple pre-rotated images, rotating coordinates with fixed-point math.

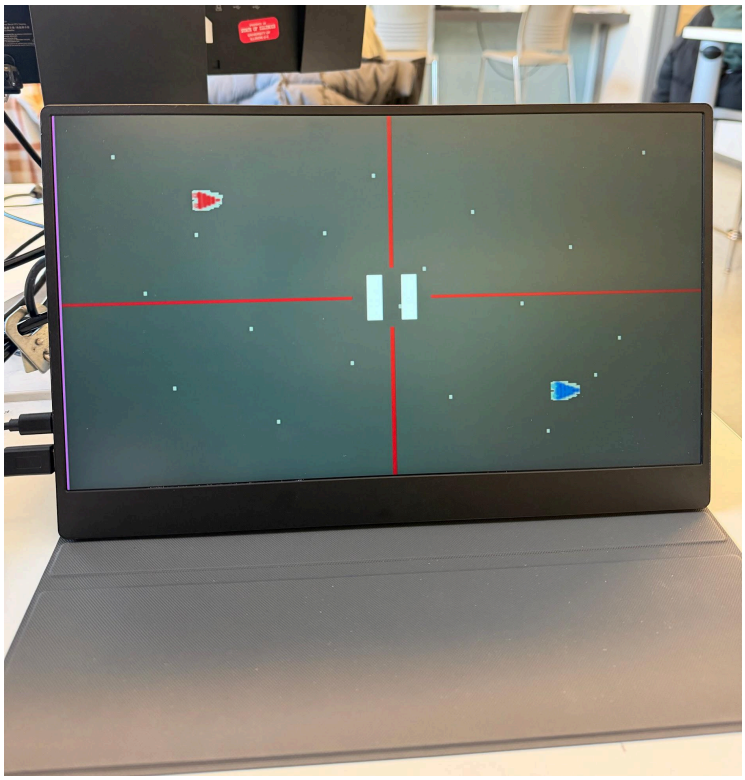
Images



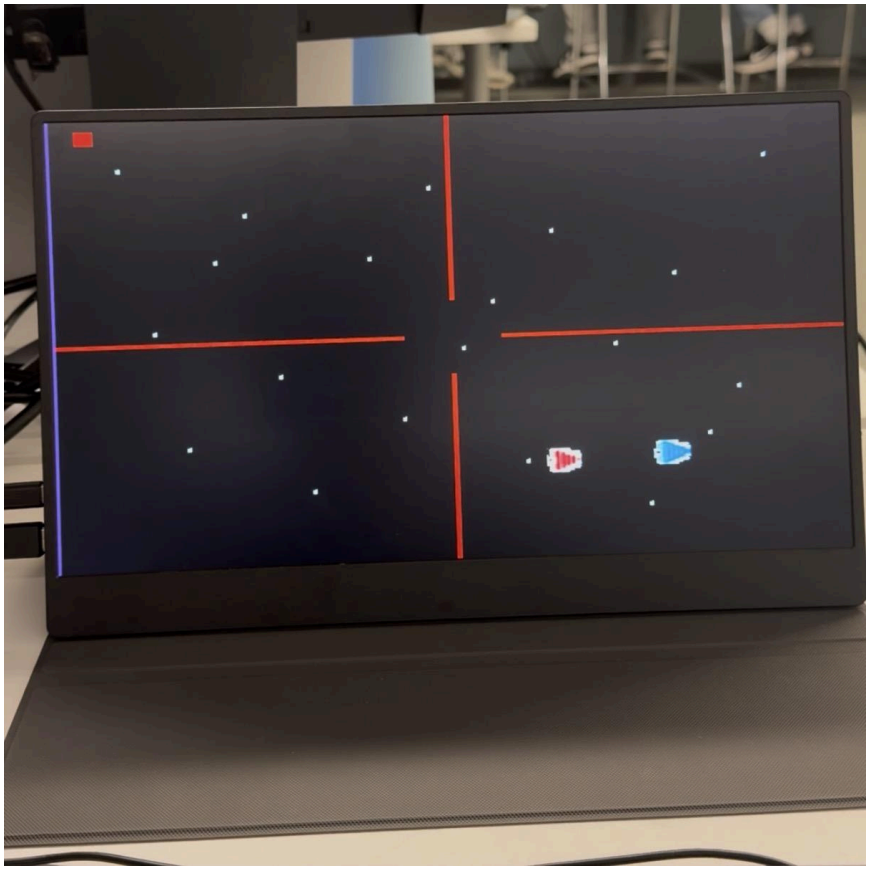
Title Screen



Start Screen



Pause Screen



Screen with Kill Counter Shown

Design Resources and Statistics Table

Final Project Design Statistics	
LUT	4159
DSP	36
Memory (BRAM)	63
Flip-Flop	3281
Latches	0
Frequency	34.62 MHz
Static Power (W)	0.078
Dynamic Power (W)	0.414

Total Power (W)	0.492

From the table we computed the frequency by using the equation f = 1/(10ns - WNS).

Conclusion

In conclusion, our ECE 385 final project involved designing and implementing Astro Party, a fast-paced two-player space combat game on an FPGA. By integrating the MicroBlaze processor, MAX3421E USB host controller, and a text-mode graphics controller, we successfully combined hardware and software to create an interactive gaming system. We implemented core game mechanics such as ship movement and rotation, bullet firing with collision detection, wall obstacles, and a kill-count system. We were able to implement sprite rotations using transformation matrices, and keyboard controls to be able to move, turn, and shoot the ships. We also added a title screen, pause functionality, and a win screen. The challenges we encountered, such as real-time rendering, system integration, and debugging, improved our understanding of FPGA-based design. Overall, this project served as a valuable learning experience in designing a functional and interactive game on an FPGA.