ECE 385
Fall 2023

Final Project

Jack Walberer
Advait Renduchintala
Section JH

Introduction:

For our project we decided to make space invaders, the classic arcade game. The goal of the game is to destroy all aliens. The player controls the spaceship at the bottom of the screen and shoots at the aliens controlled by the computer. The aliens shoot back and the player loses if they get shot. All our code was in system verilog except for the code needed to control the USB keyboard which we needed to control the game. Sound, game logic, sprites and all other game components are in system verilog. Randomization was needed to control when the aliens shoot.

Written Description and Diagrams of Microblaze System:

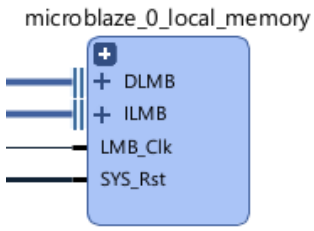
Microblaze:

MicroBlaze allows us to perform control logic applications. In our case it helps by acting as the USB driver.

Clocking Wizard:

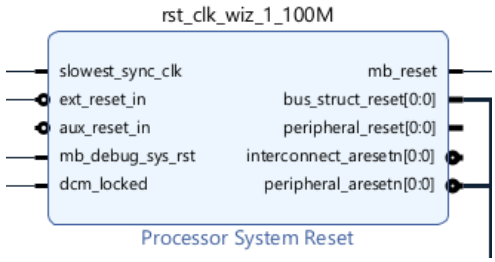
The Clocking Wizard creates clock sources and optimizes and generates the HDL, ensuring efficient and stable clock networks in FPGA designs.

Microblaze_0_local_memory:



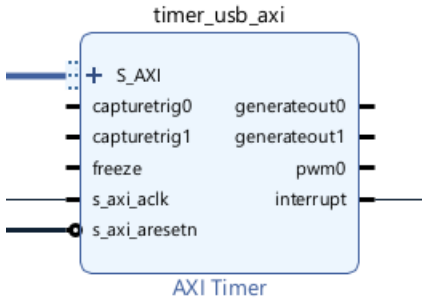
This local memory provides the MicroBlaze processor with fast and direct access to data for instruction and data storage. It plays a crucial role in improving the performance of the processor by reducing the need to access slower external memory sources.

Rst_clk_wiz_1:



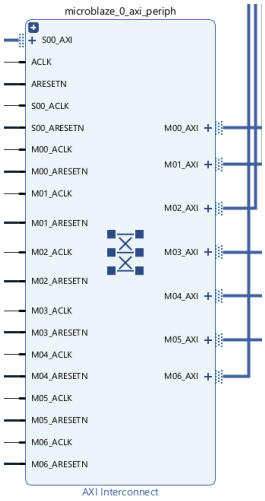
`Rst_clk_wiz_1` is the reset signal associated with a clock wizard. This signal is used to initialize or reset the clock wizard to ensure stable and predictable clock generation and configuration.

Timer_usb_axi:



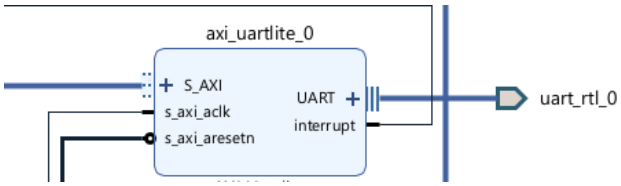
`Timer_usb_axi` refers to a timer module that's interfaced through the AXI protocol. This timer can be used to generate precise delays, measure intervals, or trigger events, and can also communicate with other AXI-compliant modules or processors in the system.

Microblaze_0_axi_peripheral:



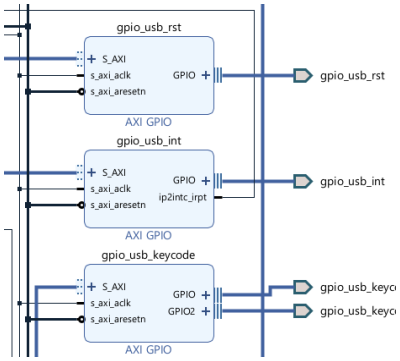
`Microblaze_0_axi_peripheral` is the AXI associated with the MicroBlaze. This interface facilitates communication between the MicroBlaze processor and various peripheral devices or memory controllers. It ensures that data transfers, memory accesses, and control signals can move efficiently between the processor and the connected peripherals.

Axi_uartlite:



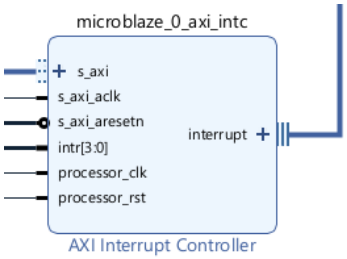
`Axi_uartlite` is a core that communicates using the AXI protocol in FPGA designs. It provides basic UART functionality, enabling serial communication between the FPGA and external devices or systems through the AXI interface.

GPIO:



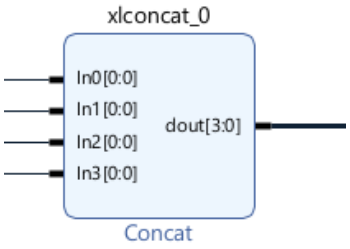
These 3 GPIOs allow us to connect the keyboard to the FPGA so that we can input in keyboard commands.

AXI_interrupt_controller:



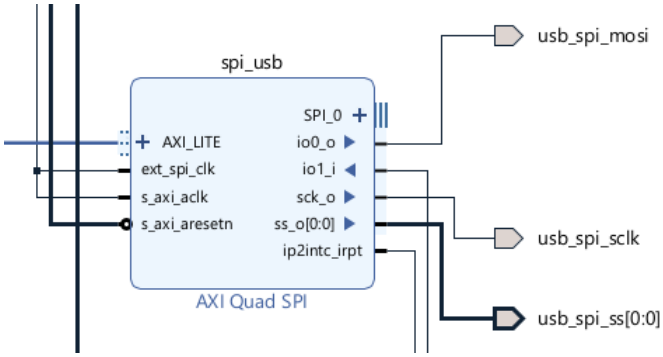
The `AXI_interrupt_controller` is a module that manages and prioritizes multiple interrupt sources when interfaced through the AXI protocol. It combines interrupt signals from various peripherals and provides a centralized point for the processor or other logic to handle and acknowledge these interrupts.

Concat:



This simple block takes 4 inputs from the GPIO and UART and then feeds into the interrupt controller so that the signal can be sent.

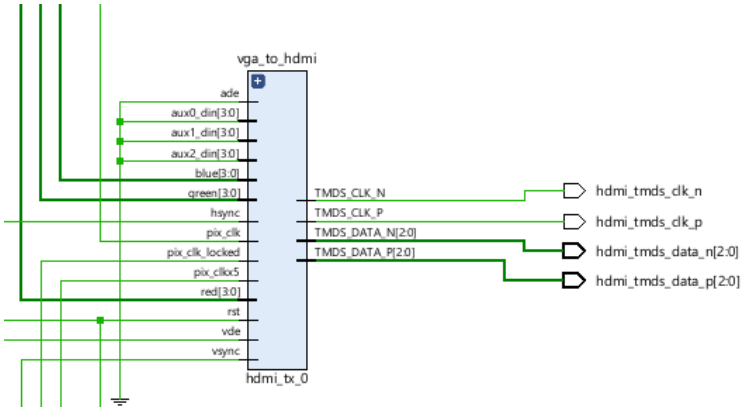
AXI_quad_SPI:



The `AXI_quad_SPI` is a module that facilitates communication with SPI devices using the AXI protocol. Unlike standard SPI, which uses a single line for data transmission, this module allows for faster data transfers. This module manages SPI transactions, ensuring that data is correctly sent and received between the FPGA and connected SPI devices through the AXI interface. It is also a synchronous serial protocol used to communicate with the MAX3421E.

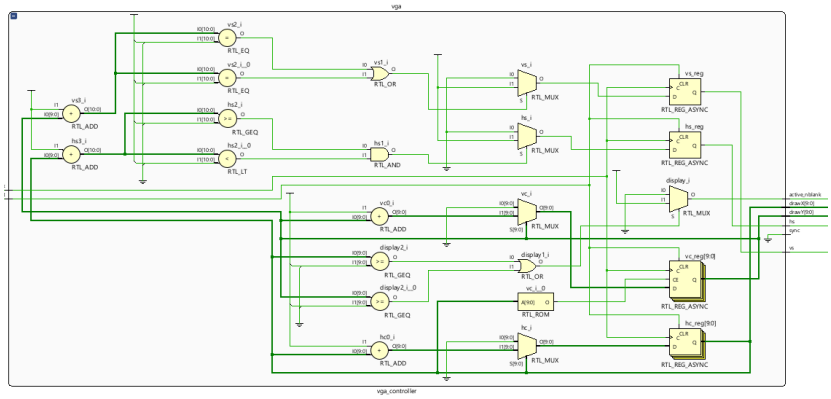
Module Descriptions:

Vga_to_hdmi (Very Similar to lab 6.2):



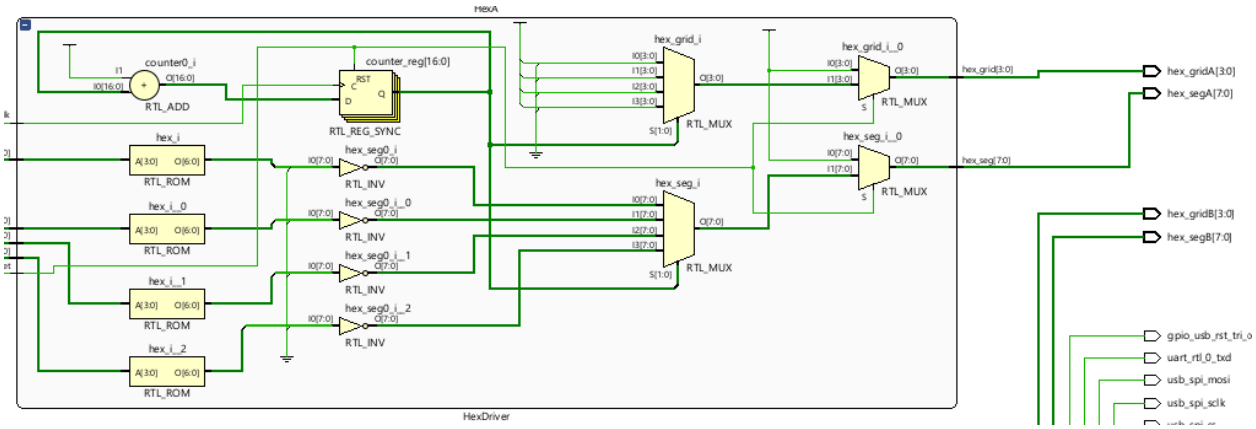
VGA to HDMI conversion involves taking an analog VGA signal, digitizing it, and then transmitting it as a digital HDMI signal. This converts VGA inputs (pixel data, sync signals) to HDMI outputs (TMDS , clock, etc.).

Vga (Very Similar to lab 6.2):



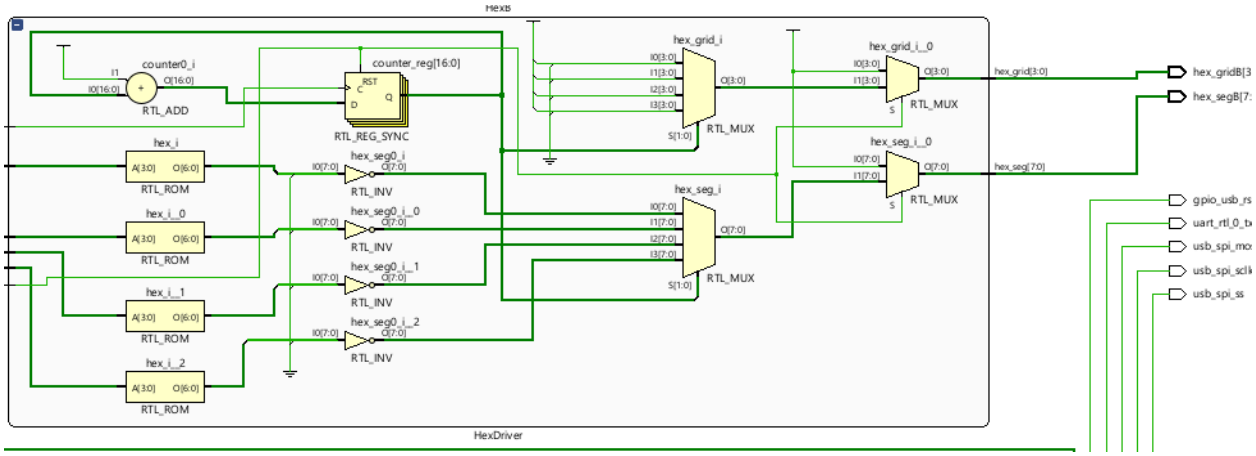
This VGA controller generates synchronization and timing signals for a 640x480 VGA display based on a 50 MHz pixel clock. It keeps track of horizontal (hc) and vertical (vc) pixel positions using counters, producing horizontal (hs) and vertical (vs) sync pulses at specific intervals to signal the start of a new line or frame. The controller outputs pixel coordinates (drawX and drawY) and a display signal (active_nblank) to determine if a particular pixel should be displayed or if it's within the blanking interval.

Hex_A (Very Similar to lab 6.2):



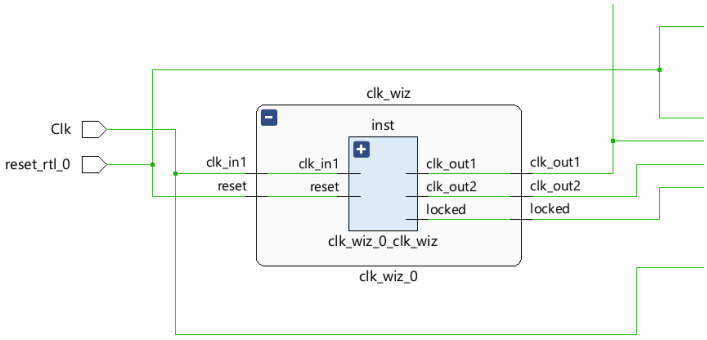
Contains the bit to hex conversion for all 16 bit values, and takes the value to be put on the fpga, then converts it to hex and displays it.

Hex_B (Very Similar to lab 6.2):



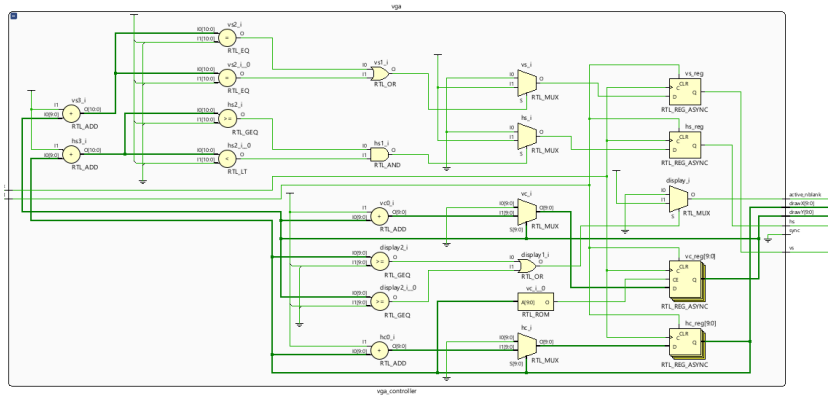
Contains the bit to hex conversion for all 16 bit values, and takes the value to be put on the fpga, then converts it to hex and displays it.

Clk_wiz (Very Similar to lab 6.2):



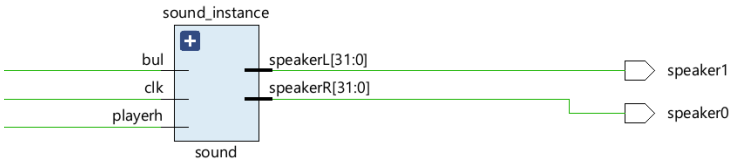
We imported this clocking wizard from Vivado's library, as it simplifies the task of generating and configuring clocking logic for FPGAs. It allows us to create clock outputs from various input sources and also helps by having high-accuracy clocks essential for high-speed designs and interfaces.

Vga (Very Similar to lab 6.2):



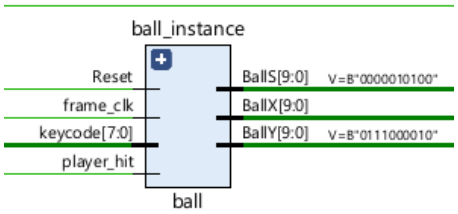
This VGA controller generates synchronization and timing signals for a 640x480 VGA display based on a 50 MHz pixel clock. It keeps track of horizontal (hc) and vertical (vc) pixel positions using counters, producing horizontal (hs) and vertical (vs) sync pulses at specific intervals to signal the start of a new line or frame. The controller outputs pixel coordinates (drawX and drawY) and a display signal (active_nblank) to determine if a particular pixel should be displayed or if it's within the blanking interval.

Sound:



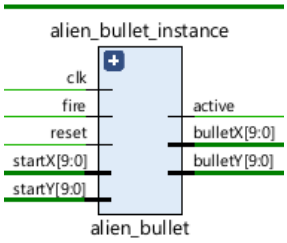
This project makes sound using a counter and logic to output different notes. A sound is made when the player shoots and continues to play until the projectile the player shot either hits an alien or misses and goes off the screen. Sound also plays on the game over screen. Sounds are made by sending frequencies to the speaker out pins. The frequencies are gotten from reading a counter, the counter is updated on a 25MHz clock, so reading the 14th bit of the clock for example would play a frequency of 25MHz/2^14, which is in the human audio range. We made the sounds used by some if and always_ff logic to play different notes depending on the clock.

Spaceship (called ball because code is referenced from lab 6.2):



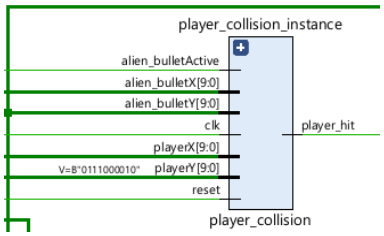
This module allows our player to move using the “A” and “D” key to move along the x axis between pixels 100 and 600. The y position stays constant because we wanted to keep the game accurate-ish. Using the current player coordinates, the color mapper gets a signal to draw the spaceship sprite.

Alien bullet fired from aliens that are alive:



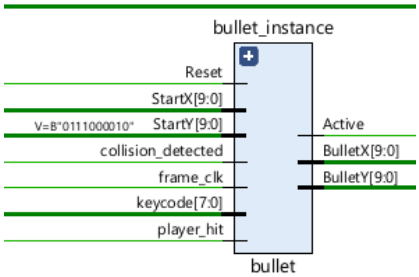
The alien bullet module just fires a bullet from a given alien (depending on the randomness) and then describes the bullet movement as straight down towards the “ground” of the game if they miss the player. The collision logic between the alien bullet and the player is done in another module.

Player collision (when alien bullet collides with spaceship/player):



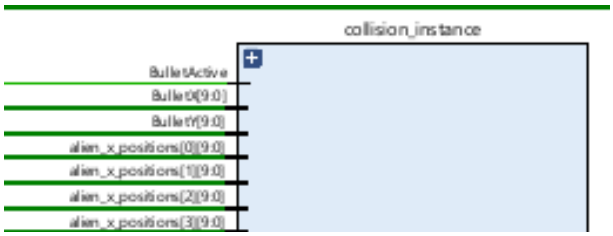
This module uses inputs of whether the alien bullet is active and the player and alien bullet coordinates to determine if a player has been hit by a bullet. If this is true, the player_hit flag is marked and the color mapper switches the sprite.

Bullet fired from spaceship:



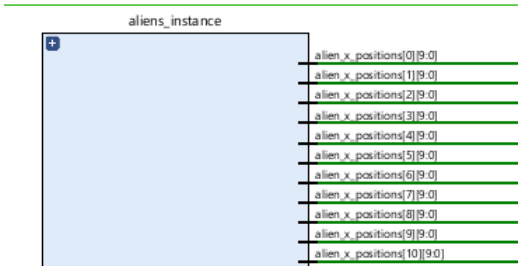
The bullet fired from the spaceship occurs every time a player pressed the spacebar on the keyboard. The bullet fires from the center of the player's coordinates and sends a signal to the color mapper to ensure that the proper sprite for the player bullet is drawn.

Collision of player and alien:



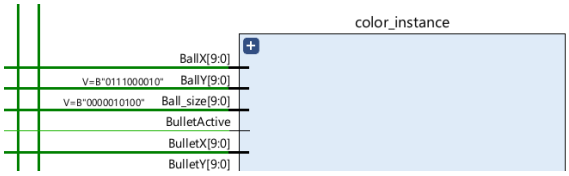
This module checks whether a player bullet has hit an alien and updates the alien states arrays to ensure that the color mapper gets the right message to color the alien as "dead". We only provided a partial screenshot as there are many inputs to this module.

Moving aliens for the space ship to shoot down:



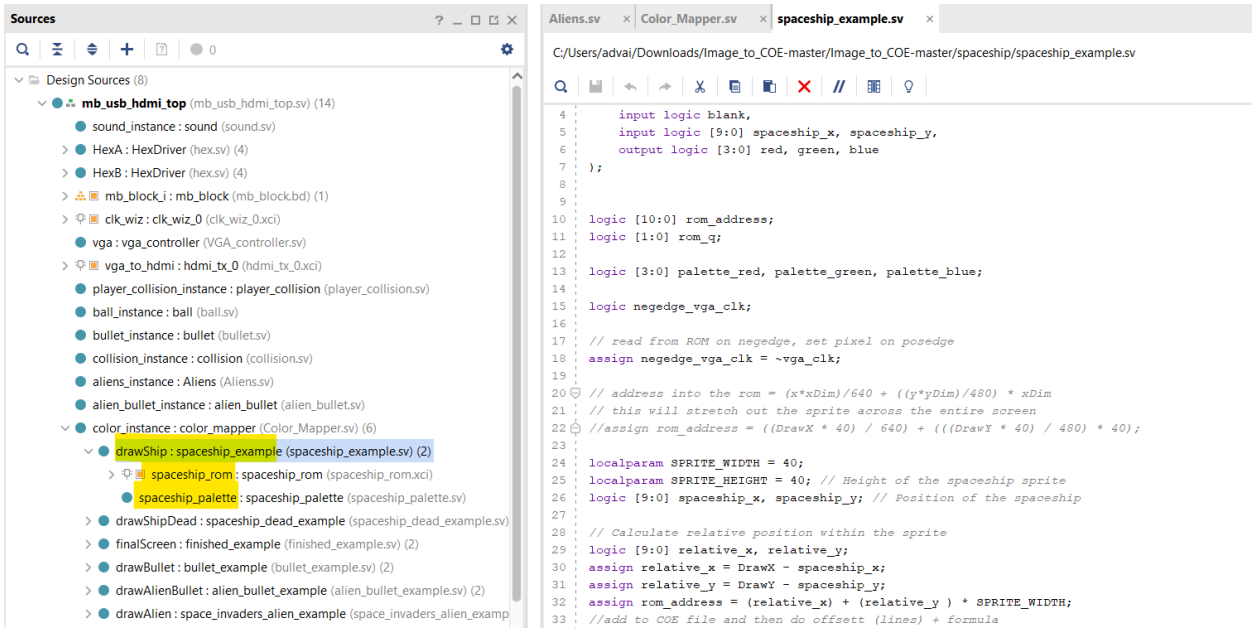
We only used a partial screenshot of the collision module because it's huge as it dynamically shifts the aliens to add a movement feature to them. The aliens' states (alive/dead) and their x and y coordinates are stored in arrays which allow the color mapper to accurately draw sprites of the moving aliens. We also used a linear feedback shift register to implement the randomness logic of the alien bullet firing because we thought that this would be a cool aspect to ensure that there isn't a pattern.

Color mapper for drawing everything in the game:



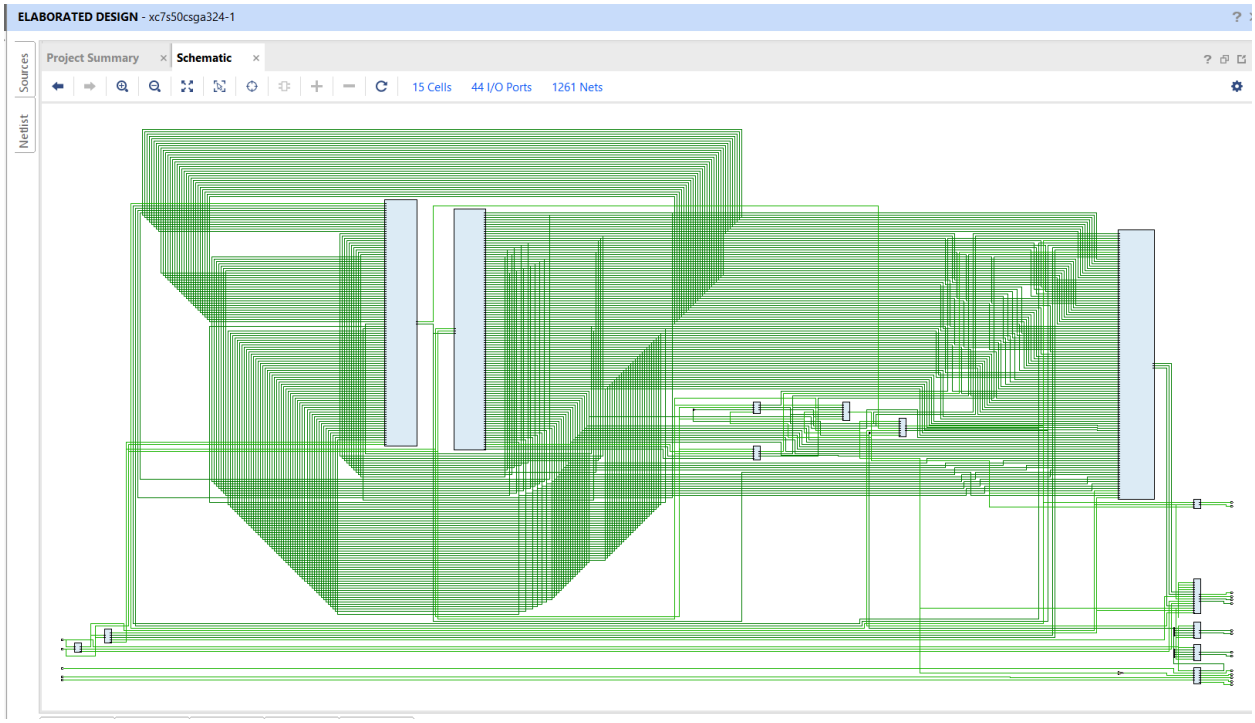
The color mapper draws all our sprites accurately and depending on the position they need to be in since our game is very dynamic. Again, we only provided a small screenshot as it has many different inputs in the top level to ensure that our game works. Our color mapper also contains instantiations of the sprites we need to draw using the BRAM (pictures below).

```
spaceship_example drawShip(
.vga_clk(clk_25MHz),
.DrawX(DrawX),
.DrawY(DrawY),
.blank(vde),
.spaceship_x(BallX - Ball_size),
.spaceship_y(BallY - Ball_size),
.red(spaceship_red),
.green(spaceship_green),
.blue(spaceship_blue)
);


spaceship_dead_example drawShipDead(
.vga_clk(clk_25MHz),
.DrawX(DrawX),
.DrawY(DrawY),
.blank(vde),
.spaceship_x(BallX - Ball_size),
.spaceship_y(BallY - Ball_size),
.red(spaceship_dead_red),
.green(spaceship_dead_green),
.blue(spaceship_dead_blue)
);


 finished_example finalScreen(
```

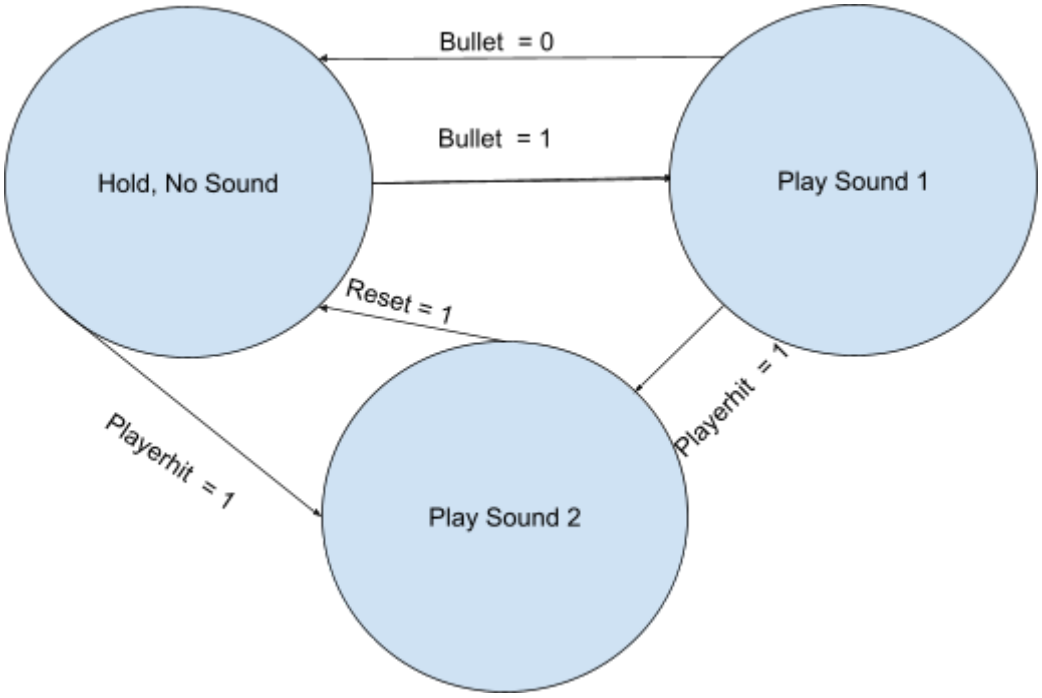


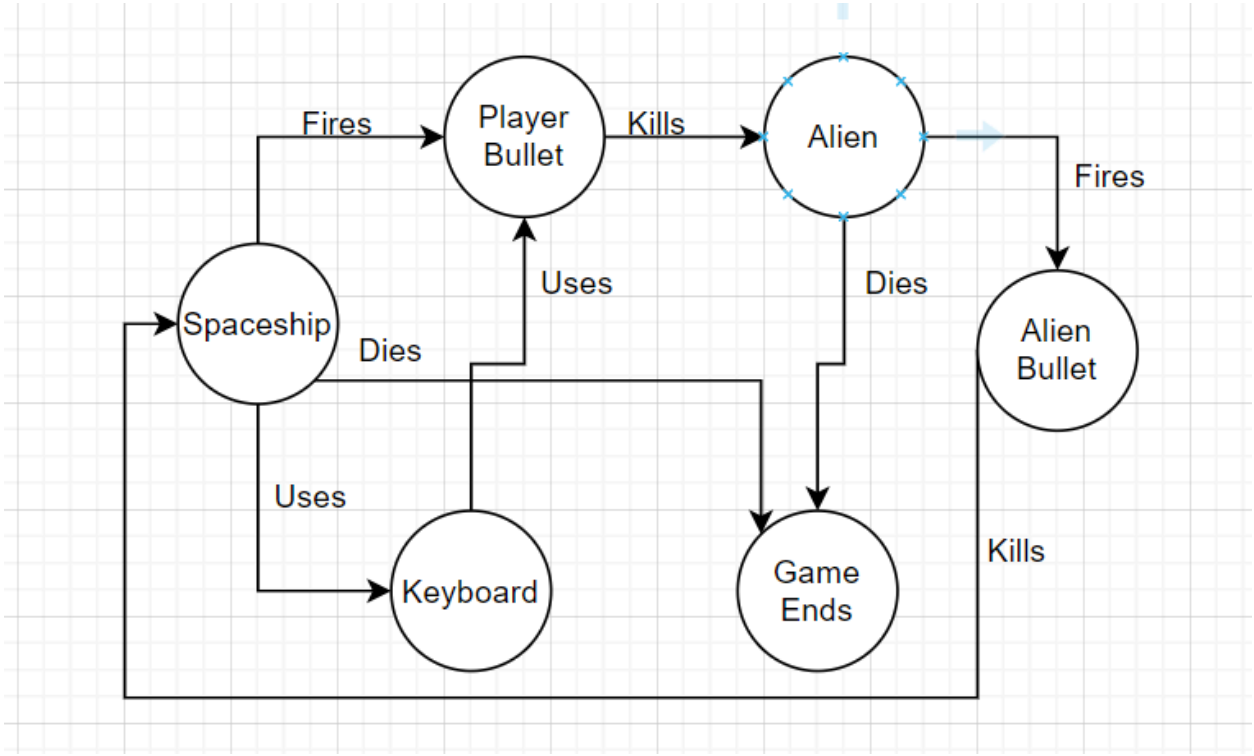
This is a further explanation of code and structure of the ROM we were using to implement the sprite images.

Overall Elaborated Block Design:



Finite State Machine Diagrams:





Design Statistics:

Tcl Console | Messages | Log | Reports | **Design Runs**

Name	Constraints	Status	WNS	TNS	WHS	THS	WBSS	TPWS	Total Power	Failed Routes	Methodology	RQA Score	QoR Sugges
synth_1 (active)	**constrs_1**	**Synthesis Out-of-date**											
impl_1	constrs_1	write_bitstream Complete!	-1.082	-32.53	0.026	0.000		0.000	0.513	0	261 CW, 84 Wa		
Out-of-Context Module Runs													
hdmi_tx_0_synth_1	hdmi_tx_0	synth_design Complete!											
clk_wiz_0_synth_1	clk_wiz_0	synth_design Complete!											
mb_block		Submodule Runs Complete											
spaceship_rom_synth_1	spaceship_rom	synth_design Complete!											
finished_rom_synth_1	finished_rom	synth_design Complete!											
bullet_rom_synth_1	bullet_rom	synth_design Complete!											

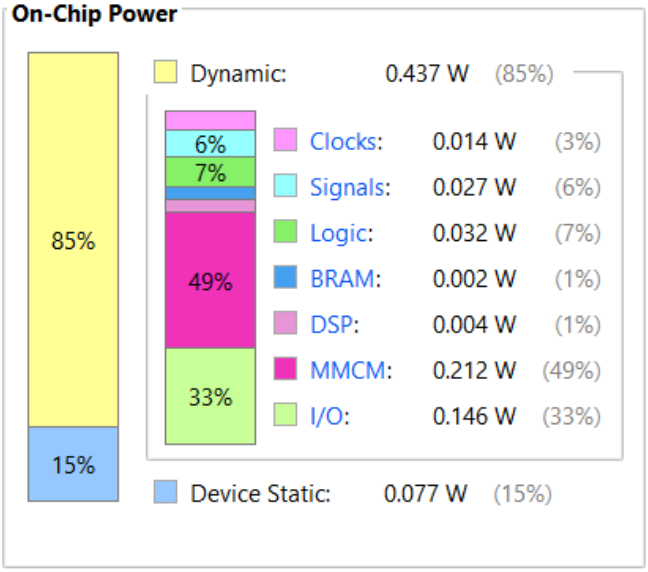
Methodology	RQA Score	QoR Suggestions	LUT	FF	BRAM	URAM	DSP	Start	Elapsed	Run Strategy	Report Strategy
			4142	**374**	**0**	**0**	**10**	**12/8/23, 1:34 PM**	**00:01:38**	**Vivado Synthesis Defaults (Vivado Synthesis 2022)**	**Vivado Synthesis De**
261 CW, 84 Wa			6794	3062	30	0	13	12/8/23, 1:36 PM	00:04:23	Vivado Implementation Defaults (Vivado Implementation 2022)	Vivado Implementati
			271	155	0	0	0	10/17/23, 5:29 PM	00:00:55	Vivado Synthesis Defaults (Vivado Synthesis 2022)	Vivado Synthesis Def
			0	0	0	0	0	10/17/23, 5:34 PM	00:00:46	Vivado Synthesis Defaults (Vivado Synthesis 2022)	Vivado Synthesis Def
								10/17/23, 5:36 PM	00:06:57		
			0	0	0.5	0	0	12/7/23, 4:58 PM	00:01:52	Vivado Synthesis Defaults (Vivado Synthesis 2022)	Vivado Synthesis Def
			16	10	47.5	0	0	12/7/23, 6:35 PM	00:01:55	Vivado Synthesis Defaults (Vivado Synthesis 2022)	Vivado Synthesis Def
			0	0	0.5	0	0	12/7/23, 7:02 PM	00:01:25	Vivado Synthesis Defaults (Vivado Synthesis 2022)	Vivado Synthesis Def

Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.513 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**27.5°C**
Thermal Margin:	57.5°C (11.6 W)
Effective ϑJA:	4.9°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity



Maximum Frequency = 1000/(10ns - WNS)
= 1000/(10-1.082)
= 112.13

LUT	**6794**
DSP	**13**
MEMORY (BRAM)	**30**
Flip-Flop	**3062**
Latches*	**0**
Frequency	**121.966 MHz**
Static Power	**0.077 W**
Dynamic Power	**0.437 W**
Total Power	**0.513 W**

Conclusion:

Overall, this project was a success, with the exception of some of the alien sprites. The player could move, shoot, lose, win as intended. The gameplay was at least comparable to the quality of the original game. The game included a game over screen, collision detection, sound, and movement of over 50 entities programmed nearly entirely in system verilog. The only thing that was done in C was the same code from lab 6.2 to convert keycodes from out keyboard. The enemy sprite fell short because it was incredibly difficult for all 50 enemy blocks to line up with their own alien sprite image. The sprite for the enemies kept showing up as a background rather than on top of each block. This is likely because we instantiate the enemies using a loop because we knew they were going to be in a grid formation the whole game. When one enemy is shot by the player it is made transparent so the projectiles can pass through it. Other than the one issue with a large number of enemy sprites, the project came out as intended. With more time, sound would be made more complex to add background noise and controller support would be added.