ECE 385

Fall 2021

Experiment #6

X-wing Attack

Michael Grawe, Matthew Fang
ABE
Abigail Wezelis

Index

Introduction

For our final project, we created X-Wing Attack, a Star-Wars themed space shooter game similar to galaga and space invaders. The unique factor of the development of this game is that it was created with a majority of the code being SystemVerilog, a digital hardware description language not generally used for game creation. This was a fun experience to challenge our problem solving skills and think about SystemVerilog in a different way than we've been using it since now.

Written Description

The goal of this game is to survive four different waves of enemies and defeat the boss. The waves consist of three different types of enemies: normal enemies which bounce across the screen before flying down offscreen, enemy shooters, which in addition to similar movements as normal enemies, randomly shoot downwards, and enemy trackers, which track the player and constantly move towards the player ship. After the four waves, a boss appears which has different movement patterns and different attacks.

To play the game, use WASD to move around and space to shoot lasers. The player starts with 3 lives and loses once he/she reaches 0 lives. The player can win once the boss is defeated. Points are calculated based on how many laser shots he/she lands. At various times different power ups, including invincibility, speed ups, double points, and extra lives drop from the top of the screen and can be picked up by the player.

There are three different difficulty levels. The harder the difficulty, the more lives enemies have, and the faster the trackers fly. The difficulty also changes what types of enemies spawn while the boss is alive.

FSM Descriptions

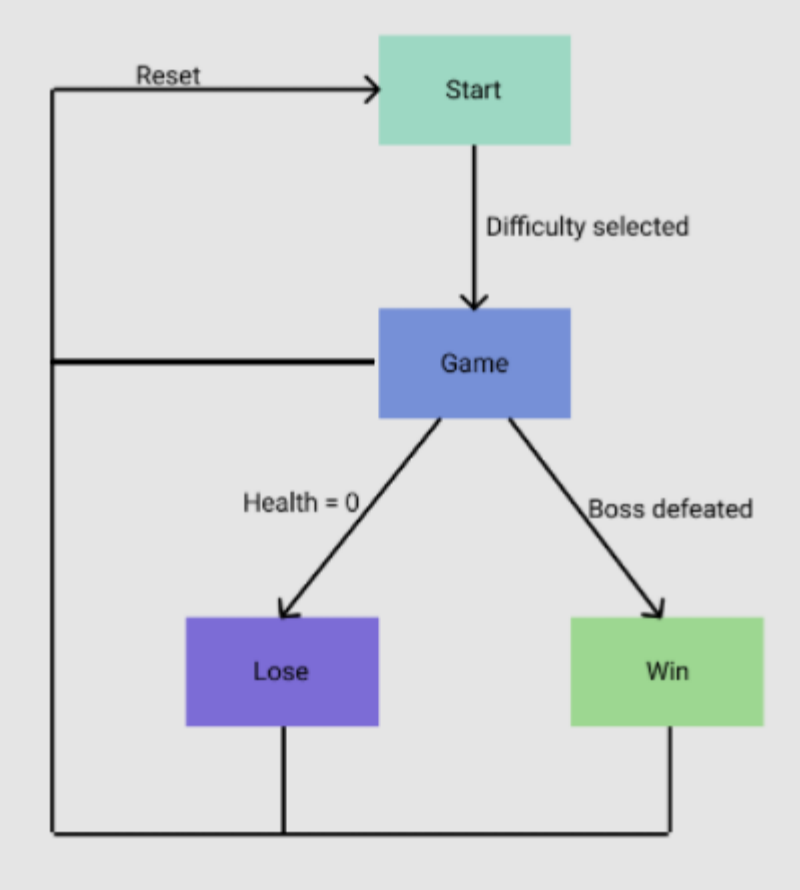


Figure 1: Game Status Controller

Game Status Controller: The game starts in a Start phase until a difficulty is selected by the player. The game will process to the Game state and the enemies will spawn. During the game state, if the signals representing health reached zero or the boss has been defeated, the game state proceeds to a Lose or Win game state. If at any time, the Reset button is pressed, the game returns to the Start Phase where the player is able to select another difficulty.

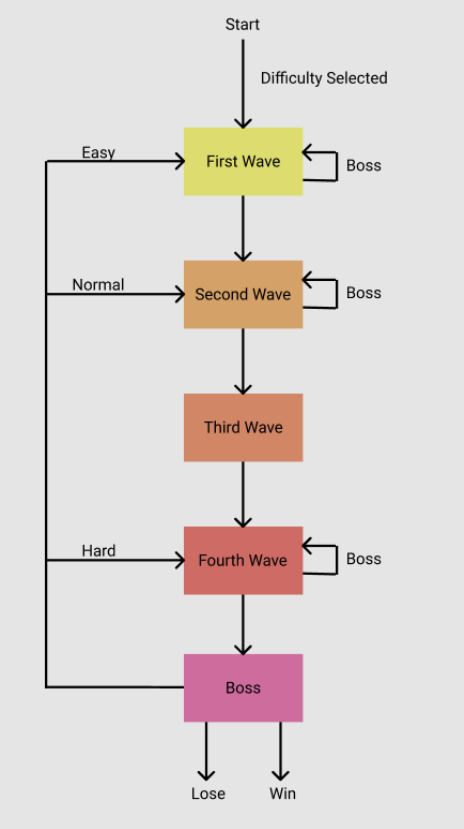


Figure 2: Enemy Wave Controller

Enemy Wave Controller: This FSM is located in the Game state of Figure 1. This controls the different waves that will spawn throughout the game and unconditionally progressing from the First Wave state to the Fourth Wave state There is a wave counter inside the Enemy Controller module which counts which wave the game is currently in. Once the Boss stage has been reached, the boss will be spawned before the FSM returns to one of the previous waves based on the difficulty selected at the beginning of the game. The Boss stage sets the wave counter to a special value signifying that when the game returns to a previous state, this is not the first time through and the game should not progress to the next wave and instead, should continually repeat the wave until the Game state is completed.

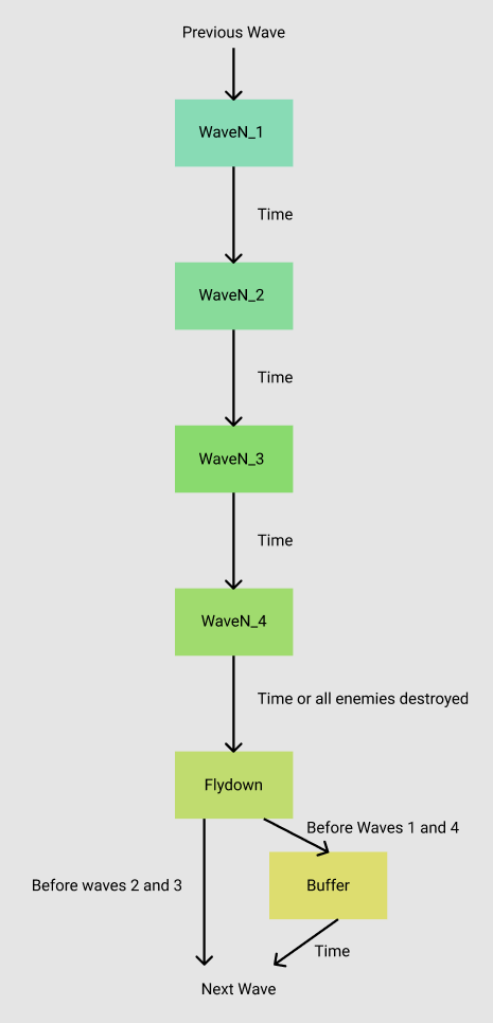


Figure 3: Spawn control

This FSM represents each of the waves in the Enemy Wave Controller of Figure 2. This controls the progression of each wave as well as the movement of normal and shooting enemies. There are four WaveN_# which are controlled by a timer. This is used to space out the enemy spawns in a wave so they don't all spawn on top of each other. After a certain time period, if enemies are still alive, the Flydown state occurs in which enemies fly downwards and disappear off screen when they reach the bottom. Waves 2 and 3 use unique enemies from the previous wave. However, wave 1 and 4 use enemies from the previous wave. The timing of flydown has the start of the next wave while the enemies of the current wave are in their flydown state so the Buffer state is necessary for waves 1 and 4 so that there is enough time for the enemies to disappear off screen before spawning back the next wave. In addition, flydown can be triggered when all enemies are destroyed. Even though nothing will fly down since all enemies are destroyed, this is useful if the player destroys all the enemies of a wave early so that the game can progress more quickly rather than the player having to wait the entire time for the next wave to spawn.

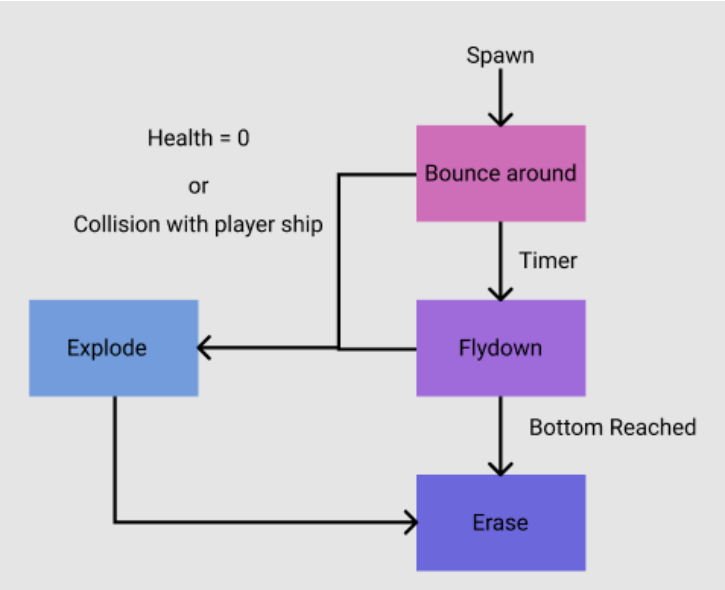


Figure 4: Normal Enemy FSM

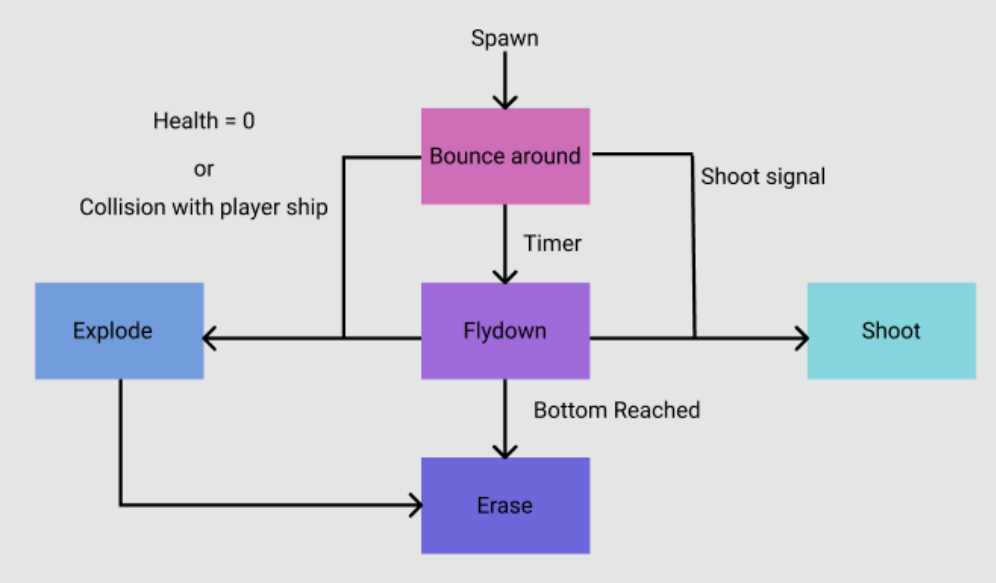


Figure 5: Enemy Shooter FSM

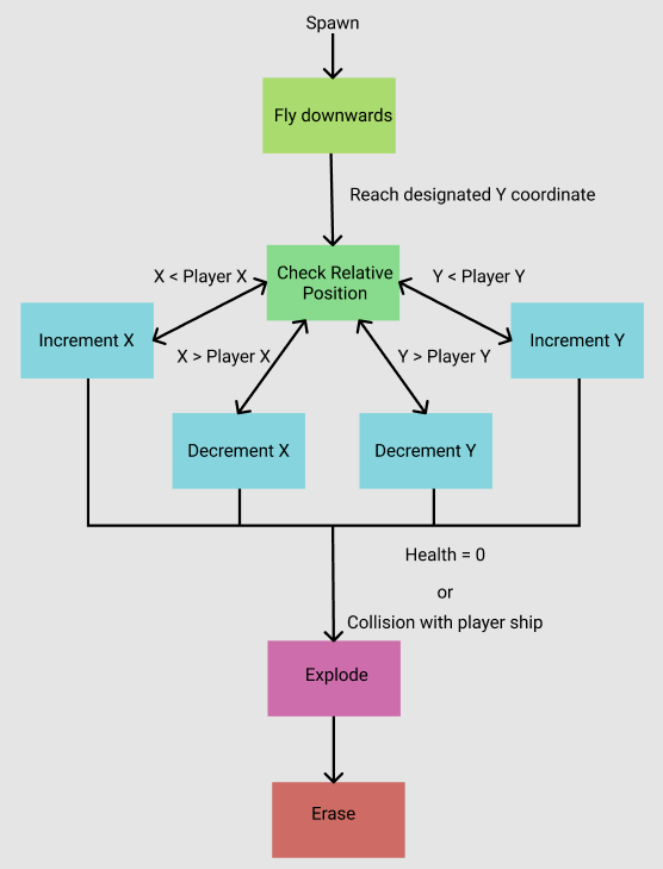


Figure 6: Enemy Flyer/Tracker FSM

Enemy Controller: Figures 4, 5, and 6 describe the various enemy actions. Each enemy has been assigned certain WaveN_# from the Spawn Control of Figure 3. Once spawned, the enemies follow their certain FSM controller before they are destroyed, or reach the bottom of the screen in their flydown state in the case of normal and shooter enemies.

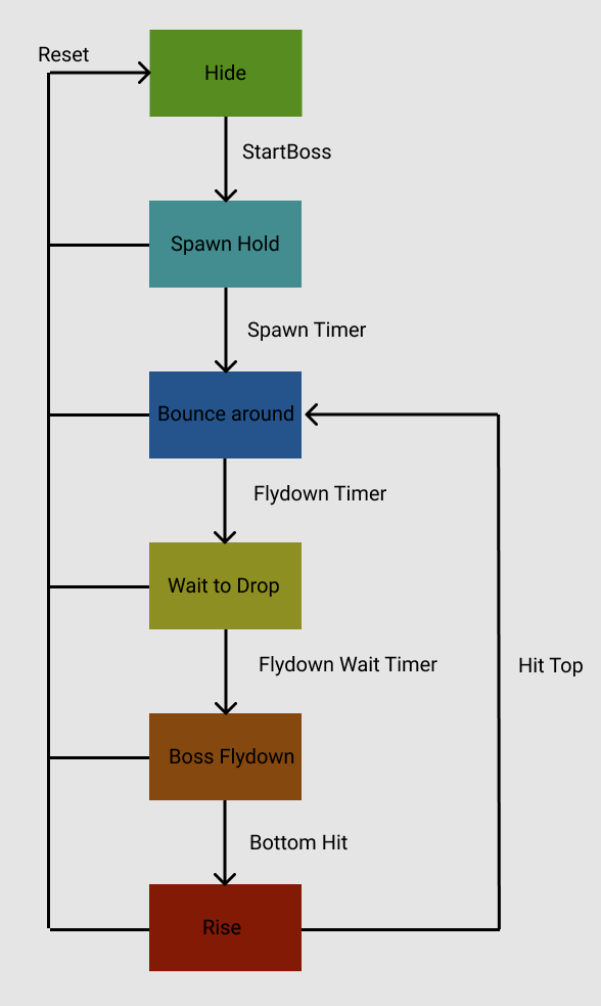


Figure 7: Boss Controller FSM

Boss Controller: The FSM used to control the Boss has 7 states. The Reset state only is reached by pressing the reset button and unconditionally transitions to the Hide state. In this state, the Boss does not exist and is offscreen. When the Start Boss signal goes high, then the Boss spawns and begins the Boss stage cycle (Spawn Hold state). It pauses until the spawn wait timer counts down to 0 and then the boss goes back and forth across the top of the screen in the Back/Forth state until the flydown timer reaches 0. Then, the Boss enters the Wait to Drop state and pauses briefly before entering the Flydo
wn state. The Flydown state ends when the boss hits the bottom of the screen, and then the boss transitions into the Rise state until it hits the top of the screen. Once the boss hits the top of the screen, it goes back to the Spawn Hold State and continues to repeat the cycle depicted below until the boss is beaten. If the reset button is pressed at any point, the boss will branch to the Reset state.

Block Diagrams

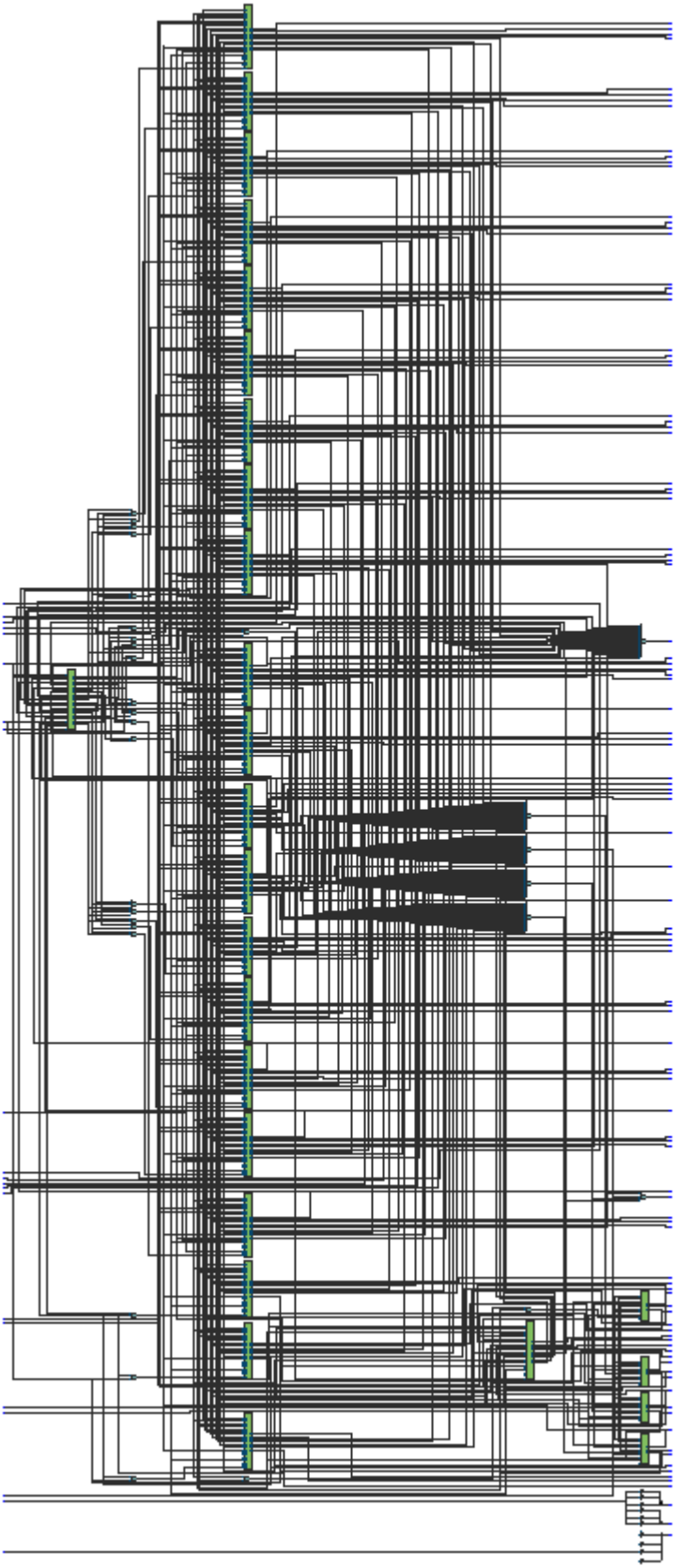


Figure 8: Enemy controller, Enemies, Cannons, Difficulty blocks

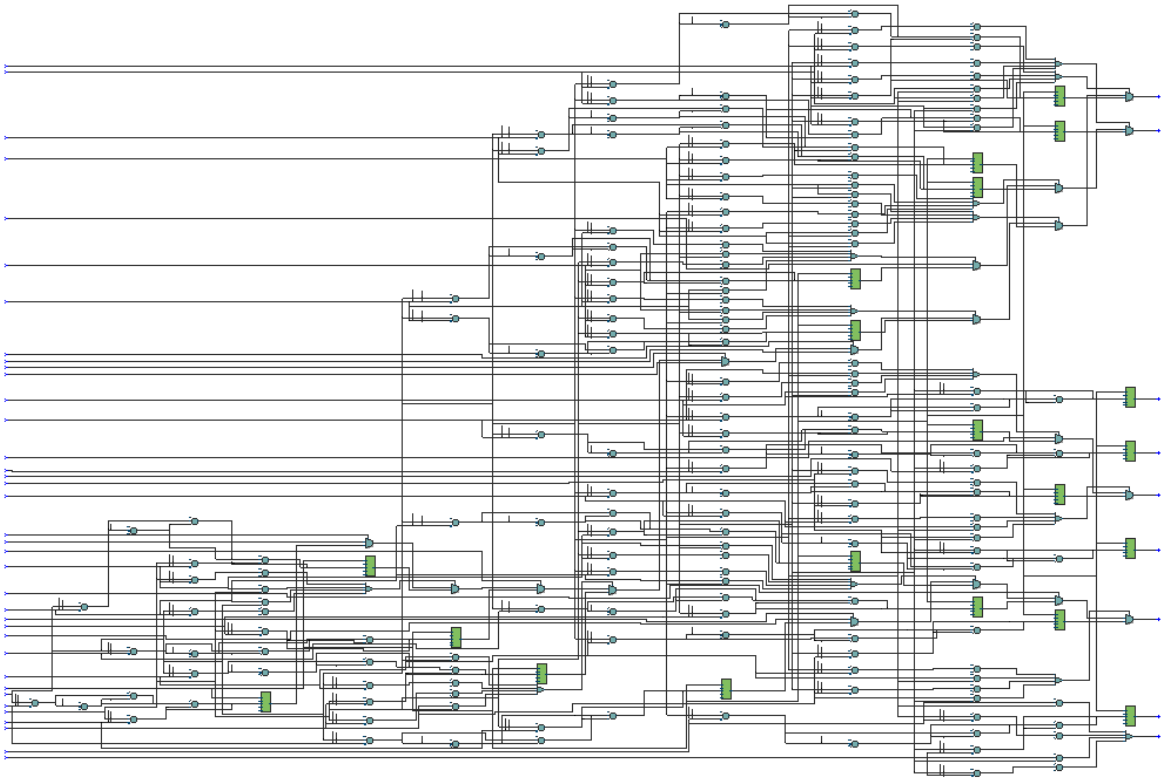


Figure 9: drawRam modules 1

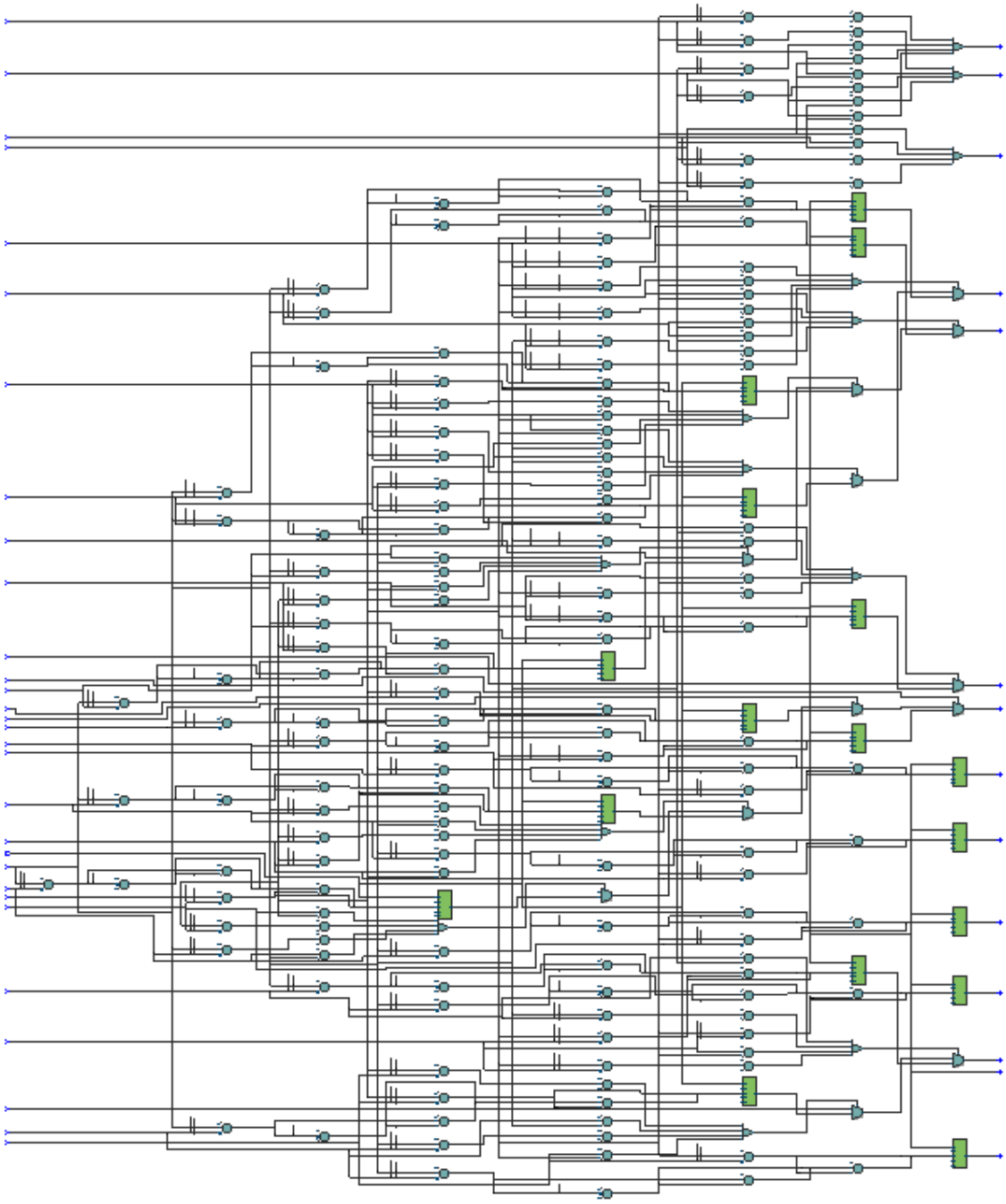


Figure 10: drawRam modules 2

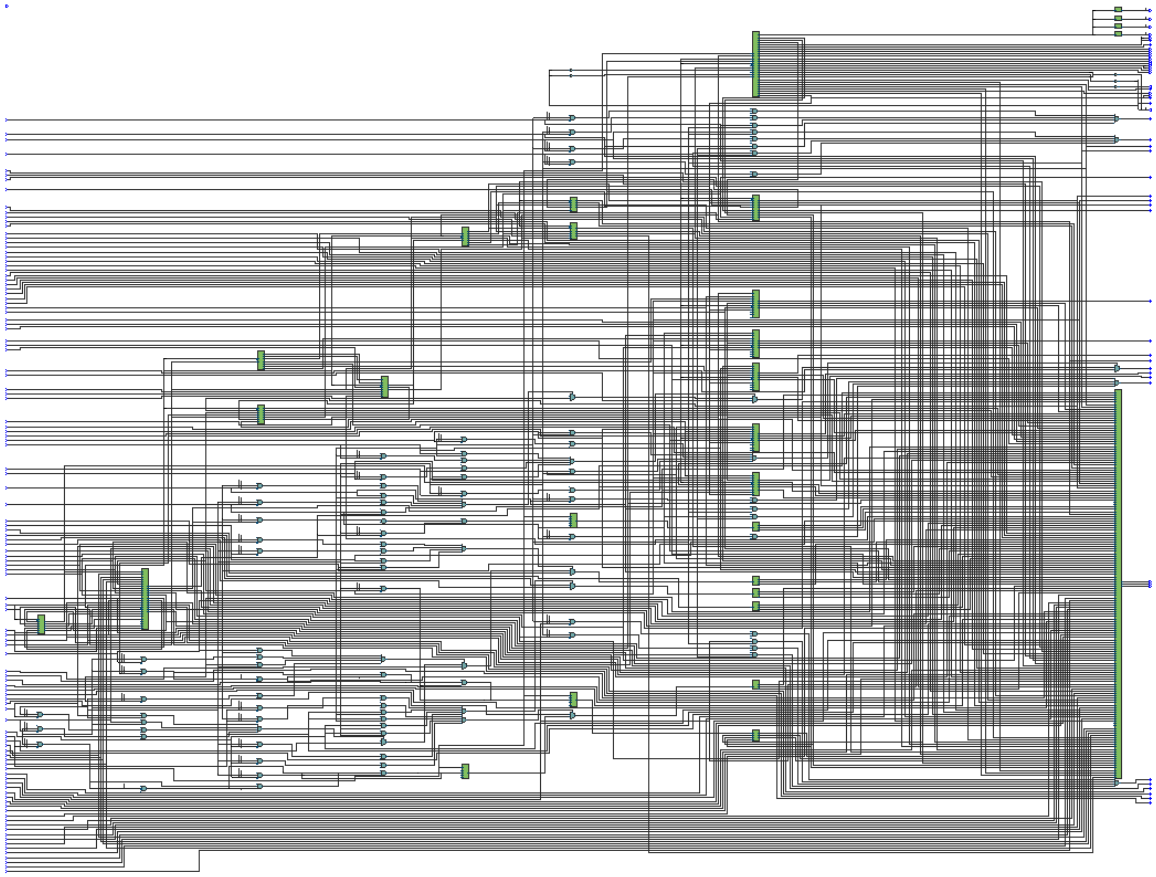


Figure 11: Spaceship, color mapper, boss, hex drivers, enemy cannons

Module Descriptions

To be concise, not all inputs and outputs are listed, as many modules contain quite many input and output logic elements. The inputs and outputs that are most important to understanding the function of the module will be listed.

Module: finalproj

Inputs:	MAX10_CLK1_50, [1:0] KEY, [9:0] SW,
Outputs:	[9:0] LEDR, [7:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, DRAM_CLK, DRAM_CKE, DRAM_LDQM, DRAM_UDQM, DRAM_CS_N, DRAM_WE_N, DRAM_CAS_N, DRAM_RAS_N, VGA_HS, VGA_VS [3:0] VGA_R, VGA_G, VGA_B, [12:0] DRAM_ADDR, [1:0] DRAM_BA
Inout:	[15:0] DRAM_DQ, ARDUINO_IO, ARDUINO_RESET_N

Description: This is the top-level module used. Within this module, finalproj_soc is instantiated, the Hex drivers are connected to the connect signals, and the Arduino signals are connected to the corresponding USB signal.

Purpose: The purpose of finalproj is to program the FPGA with the NIOS II processor and appropriate PIO's described in the Platform designer. The most important outputs to understand functionality are the VGA_R,G,B signals, as they output the 4 bit RGB color signals to the VGA monitor.

Module: finalproj_soc/synthesis/finalproj_soc
Description: These are the hardware components created in the platform designer.

Purpose: These files allow the interaction with the keyboard and monitor peripherals. They also connect the NIOS II CPU for our design. This module was used to connect PIO's created in the platform designer to signals coming from other modules in order to pass down data from our software to our hardware or vice versa.

Module: vga_text_avl_interface

Inputs:	CLK, RESET, AVL_READ, AVL_WRITE, AVL_CS [3:0] AVL_BYTE_EN [9:0] AVL_ADDR [31:0] AVL_WRITEDATA
Outputs:	[31:0] AVL_READDATA hs, vs, [3:0] red, green, blue [9:0] DrawX, DrawY

Description: vga_text_avl_interface implements the VRAM, the logic to write/read from the VRAM through the AVALON bus, as well as the algorithm to draw the characters from VRAM and font_rom. This is essentially the same module as the one from lab7.2, with an added output of DrawX and DrawY for the Color Mapper to draw at the correct position.

Purpose: The purpose of vga_text_avl_interface is to handle read/write operation into VRAM, whether it be on-chip or registers, and to translate the contents of memory into colors on screen.

Module: final_font_rom
Inputs: [10:0] addr,
Outputs: [7:0] data

Description: This module contains all the bytes for the sprites to be drawn by software through VRAM

Purpose: The purpose is to make drawing VRAM characters significantly easier.

Module: VGA_controller
Inputs: Clk, Reset
Outputs: hs, vs, pixel_clk, blank, sync
[9:0] DrawX, DrawY

Description: VGA_controller handles vertical and horizontal sync while outputting DrawX and DrawY signals to represent the part of the screen to be drawn, which is analyzed by other modules to determine what pixel should be drawn.

Purpose: The purpose of VGA_controller is to move the virtual electron gun across the screen and handle the vertical and horizontal sync in order to draw pixels.

Module: Hex Driver
Inputs: [3:0] In0
Outputs: [6:0] Out0

Description: For the purposes of the Final Project, the Hex Drivers are meant to display the keycode value of up to two currently pressed keys.

Purpose: The purpose of hexdriver is to display the keycode of the most recent key pressed.

Module: final_sdc.sdc

Description: This module sets output delays for precise time signals coming from dram and false paths for the less time sensitive signals from the switches, keys, and leds.

Purpose: The purpose of this timing file is to ensure external signals are stable before measurement.

Module: ram.v
Inputs: [3:0] byteena_a
Clock, wren
[31:0] data
[10:0] rdaddress, wraddress
Outputs: [31:0] q

Description: This module comes from a Quartus Mega-function to implement on-chip memory. It implements a two-port, one read one write memory.

Purpose: The purpose of this module is to act as VRAM in On-chip memory for VRAM (writing character in our font ROM on screen).

Module: spaceship
Inputs: Reset, frame_clk, hit, speedup, powerup, invincible, doublescore,
[7:0] keycode, keycode2, keycode3,
Outputs: [9:0] SpaceshipX, SpaceshipY, SpaceshipS,
shoot, lose_life,
[7:0] spaceship_red, spaceship_green, spaceship_blue

Description: This module creates logic to create the player's ship. This is a modification of the ball module from lab6.2. The X and Y position of the player's ship are outputs from the module, and these positions are determined by adding a speed factor to the current ship position. This speed is variable depending on which powerup the ship has. The other output of the module is the spaceship RGB signals, which vary depending on which powerup is active. Finally, this module implements the keycode based movement and shooting of the ship by updating the ship motion by the corresponding key pressed and sending a shoot signal whenever the space bar is pressed.

Purpose: The purpose of this module is to update spaceship position, the shoot signal, and spaceship colors based on powerup information and keycode information.

Module: enemy
Inputs: Reset, frame_clk, spawn, flydown,
[2:0] difficulty, row
[9:0] laser_width, laser_height, PX1, PX2, PX3, PX4, PY1, PY2, PY3, PY4,
PL1, PL2, PL3, PL4,
rand_side,
[9:0] PSX, PSY,
Outputs: [9:0] SpaceshipX, SpaceshipY, SpaceshipS,
SpaceshipE, h1, h2, h3, h4, hs, explosion
Description: This module creates logic to create the base enemy. This is a modification of the ball module from lab6.2. The X and Y position of the enemy ship are outputs from the module, and these positions are determined by logic coming from the enemy_controller.

This module also has position and size inputs for all four of the player's ship lasers as well as for the player's ship itself in order to determine of the enemy ship has been hit by one of the lasers (h1-h4) or has hit the player's ship (hs). The explosion signal is outputted when a health counter determined by the difficulty chosen reaches zero.

Purpose: The purpose of this module is to update enemy ship position, health, and existence based on collision logic with lasers and the enemyship.

Module: enemy, enemy_flyer

Inputs:	Reset, frame_clk, spawn, flydown, [2:0] difficulty, row [9:0] laser_width, laser_height, PX1, PX2, PX3, PX4, PY1, PY2, PY3, PY4, PL1, PL2, PL3, PL4, rand_side, [9:0] PSX, PSY,
Outputs:	[9:0] SpaceshipX, SpaceshipY, SpaceshipS, SpaceshipE, h1, h2, h3, h4, hs, explosion

Description: This module creates logic to create the enemy ship. This is a modification of the ball module from lab6.2. The X and Y position of the enemy ship are outputs from the module, and these positions are determined by logic coming from the enemy_controller. This module also has position and size inputs for all four of the player's ship lasers as well as for the player's ship itself in order to determine of the enemy ship has been hit by one of the lasers (h1-h4) or has hit the player's ship (hs). The explosion signal is outputted when a health counter determined by the difficulty chosen reaches zero. This description is same for both modules, except that the movement logic is different for the enemy_flyer which tracks the player ship at a set speed instead of following a set flight path.

Purpose: The purpose of this module is to update enemy ship position, health, and existence based on collision logic with lasers and the player ship.

Module: enemy_cannon

Inputs:	Reset, frame_clk, [5:0] LFSR, [9:0] PSX, PSY, [9:0] EX, EY, prev_laser_exists, [9:0] laser_width, laser_height, enemy_alive,
Outputs:	[9:0] laserX, laserY, laserexists, Phit

Description: This module creates logic to create enemy lasers that are shot from the center of an enemy ship. A shot is fired every time the LFSR signal hits 0 and the previous laser fired from the ship has left the screen. Logic using lasers from other ships is used to determine whether the next laser can be shot, since only 4 lasers can be on the screen at any given time. The laser positions are outputs and updated based movement logic. Collision detection with the bottom of the screen and the player ship determine whether the laser should disappear and if the Phit signal should go high.

Purpose: The purpose of this module is to generate enemy lasers and update enemy laser position and existence based on collision logic with the player ship.

Module: enemy_cannon
Inputs: Reset, frame_clk, laser, laser_hit, prevlaser_exists, shootfaster, powerup,
[9:0] ShipX, ShipY, ShipS,
Outputs: [9:0] laserX, laserY,
laser_exists
Description: This module creates logic to create lasers that are shot from the center of the player ship. A shot is fired every time spacebar is pressed and there are less than 4 lasers currently on the screen. The laser positions are outputs and updated based movement logic. Collision detection with the top of the screen and the enemy ships determine whether the laser should disappear.

Purpose: The purpose of this module is to generate player lasers and update player laser position and existence based on collision logic with the enemy ship.

Module: enemy_controller
Inputs: Reset, frame_clk, easy_selected, normal_selected, hard_selected,
e1, e2, e3, e4, e5, e6, e7, e8, e9, e10, e11, e12, start_game,
es1, es2, es3, es4,
Outputs: start_Boss, flydown, enemyshoot,
[2:0] difficulty,
wave1_1sig, wave1_2sig, wave1_3sig, wave1_4sig,
wave2_1sig, wave2_2sig, wave2_3sig, wave2_4sig,
wave3_1sig, wave3_2sig, wave3_3sig, wave3_4sig,
wave4_1sig, wave4_2sig, wave4_3sig, wave4_4sig

Description: This module is a state machine to determine the behavior of the enemy ships. Spawning enemies is determined by the wave signals, all of which have different compositions of shooters, flyers, and regular enemies. Timers are used to determine when enemies will stop oscillating back and forth and will try to fly down at the player ship. When the last wave is reached, control is passed to the boss controller and the boss stage begins.

Purpose: The purpose of this module is used to control the behavior of the enemy ships based on timers and a state machine.

Module: boss
Inputs: Reset, frame_clk, spawn, flydown, rise, hold, back_and_forth
[2:0] difficulty, row,
[9:0] laser_width, laser_height, PX1, PX2, PX3, PX4, PY1, PY2, PY3, PY4,
PL1, PL2, PL3, PL4, Boss_exists,
[9:0] PSX, PSY,
Outputs: [9:0] BossX, BossY, Bosswidth, Bossheight,
h1, h2, h3, h4, hs, beat_Boss,
hit_top, hit_bottom

Description: This module creates logic to create the boss. This is a modification of the ball module from lab6.2. The X and Y position of the boss are outputs from the module, and these positions are determined by logic coming from the boss_controller. This module also has position and size inputs for all four of the player's ship lasers as well as for the player's ship itself to determine if the boss ship has been hit by one of the lasers (h1-h4) or has hit the player's ship (hs).

Purpose: The purpose of this module is to update boss position, health, and existence based on collision logic with lasers and the player ship.

Module: enemy_beam
Inputs: Reset, frame_clk,
[1:0] LFSR_position,
[2:0] difficulty,
[9:0] PSX, PSY,
[9:0] bsX, bsY,
boss_exists,
Outputs: [9:0] beamX, beamY,
[4:0] beam_size,
beam_exists, Phit

Description: This module creates logic to create the boss weapon, a beam that extends in width over time. When the beam first spawns, it is white and cannot hurt the player ship. However, it then turns red and begins to expand. In this stage, the player ship can be harmed. Collision logic is used to determine whether the beam has collided with the ship and Phit should be set high. The position of the beam on the boss is determined by the LFSR position. There are four possible beam positions on the boss.

Purpose: The purpose of this module is to create an expanding beam as the boss weapon that spawns at different positions on the boss.

Module: boss_controller
Inputs: Reset, frame_clk, start_Boss, beat_Boss, hit_bottom, hit_top,
[2:0] difficulty,
Outputs: boss_flydown, boss_rise, boss_hold, boss_back_and_forth,
boss_shoot1, boss_shoot2, boss_shoot3, boss_shoot4, Boss_exists

Description: This module is a state machine to determine the behavior of the boss ships. Spawning the boss is determined by the start_Boss signal. Timers are used to determine the transitions between states. The boss can oscillate back and forth as well as 'thwomp' down to try and smash the player ship.

Purpose: The purpose of this module is used to control the behavior of the boss based on timers and a state machine.

Module: power_up
Inputs: Reset, frame_clk, generate_powerup,
[9:0] ShipX, ShipY, ShipS, powerup_startpos,
Outputs: got_powerup, powerup_exists,

[9:0] PowerupX, PowerupY, Powerupwidth, Powerupheight
Description: This module implements the movement and collision logic for the powerups. The powerups fall straight down after they are generated, which is done by updating position. If the powerup collides with the playership, then it disappears and the got_powerup signal is set high. If it collides with the bottom of the screen, the powerup also disappears.

Purpose: The purpose of this module is to make powerups fall from the top of the screen and either disappear or be absorbed by the playership.

Module: powerup_select
Inputs: Reset, frame_clk, generate_powerup,
[1:0] LFSR_powerup_type,
Outputs: logic speedup, extralife, invincibility, doublescore,
[3:0] powerup_red, powerup_green, powerup_blue

Description: This module selects which powerup is to be chosen upon generation. This is done by the LFSR powerup type signal, which chooses from 1 of 4 different powerups. The corresponding color for that powerup is sent to the color mapper, and the corresponding powerup signal is also sent as an output to determine how to modify game behavior.

Purpose: The purpose of this module is to select which type of powerup to drop.

Module: start_game
Inputs: frame_clk, Reset, easy_selected, normal_selected, hard_selected, powerup_exists,
[8:0] LFSR_powerup_startpos, LFSR_powerup_drop_timer,
Outputs: [9:0] powerup_startpos,
generate_powerup, start_game

Description: This module implements the start game logic. This includes determining powerup drop times and positions, which come from the corresponding LFSR signals.

Purpose: The purpose of this module is to determine which powerup to drop, as well as where to drop it, and pass that logic to the powerup_selector.

Module: LFSR
Inputs: Reset, frame_clk,
Outputs: [5:0] enemy_shoot1, enemy_shoot2, enemy_shoot3, enemy_shoot4,
[8:0] LFSR_powerup_pos, LFSR_powerup_timer,
[1:0] LFSR_powerup_type

Description: This module implements an LFSR, or linear feedback shift register. This is done by XORing the bottom three bits of various initial 8 bit seeds and placing the result in the 8^{th} bit while down-shifting the remaining bits. To take the desired output, a corresponding bit slice is selected and outputted.

Purpose: The purpose of this module is to implement a pseudo random number generator to randomize the type of powerup dropped, the time between powerup drops, the position of the powerup drop, and the time between enemy lasers.

Module: draw_score
Inputs: Reset, frame_clk, ship_hit, doublescore, powerup, start_game,
Outputs: [9:0] ScoreX, ScoreY,
[9:0] score_counter,
win_game

Description: This module keeps track of score for the game. Whenever an enemy ship is hit, score is added to by 1 or 2 depending if doublescore powerup is activated.

Purpose: The purpose of this module is to keep track of the game score and output that value to the software to display on the screen.

Module: draw_lives
Inputs: Reset, frame_clk, lose_life, extralife, powerup,
Outputs: [9:0] LivesX, LivesY,
[1:0] lives_counter,
lose_game

Description: This module keeps track of lives for the game. Whenever the player ship is hit, lives counter decrements by one. If extralife is high, lives counter increases by one. LivesX and LivesY are outputs that correspond to the top left corner of a hearts sprite box on the screen (display for lives counter).

Purpose: The purpose of this module is to keep track of the lives

Modules: explosion/enemyship/spaceship/boss_drawRAM
Inputs: [3:0] data_In,
[18:0] write_address, read_address,
we, Clk,
Outputs: [3:0] data_Out

Description: All four of these modules take in a 19-bit address and reads the data from the corresponding line in the correct .txt file. This data corresponds to a color index, and is outputted to be processed by the correct color pallete.

Purpose: The purpose of this module is to obtain the correct color index for a sprite based on the draw x and y values.

Modules: boss/enemy/enemy_flyer/enemy_shooter/explosion_color_pallete
Inputs: [3:0] data_in,
Outputs: [7:0] red, green, blue

Description: All these modules take a 4-bit color index and map it to the corresponding RGB values for a particular sprite.

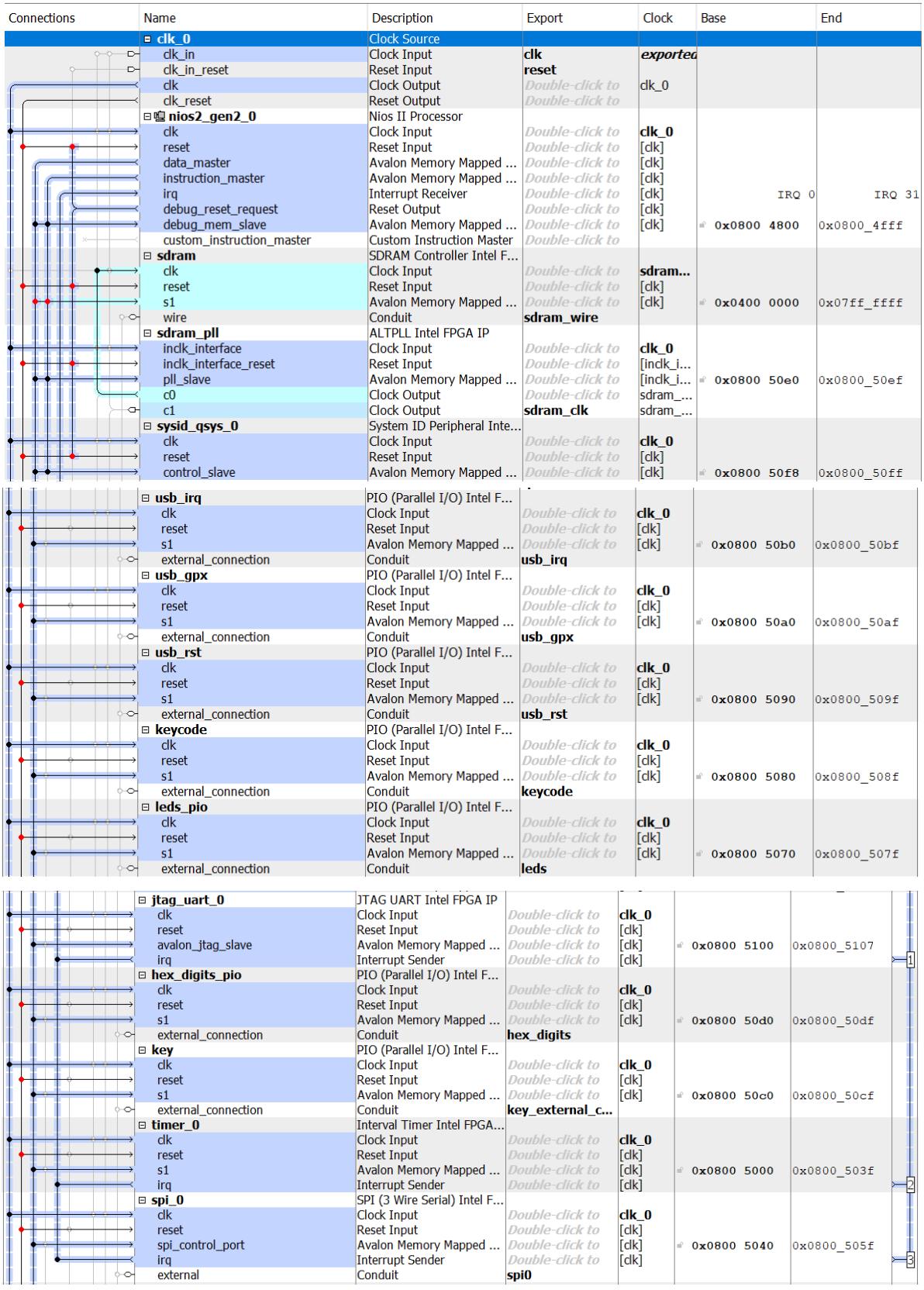
Purpose: The purpose of this module is to map each color index for a sprite to a given RGB value.

Modules: Explosion/Slave 1/X-Wing/Tie-Fighter/Explosion.txt
These are just text files containing a list of color indexes going in raster order that correspond to each individual sprite.

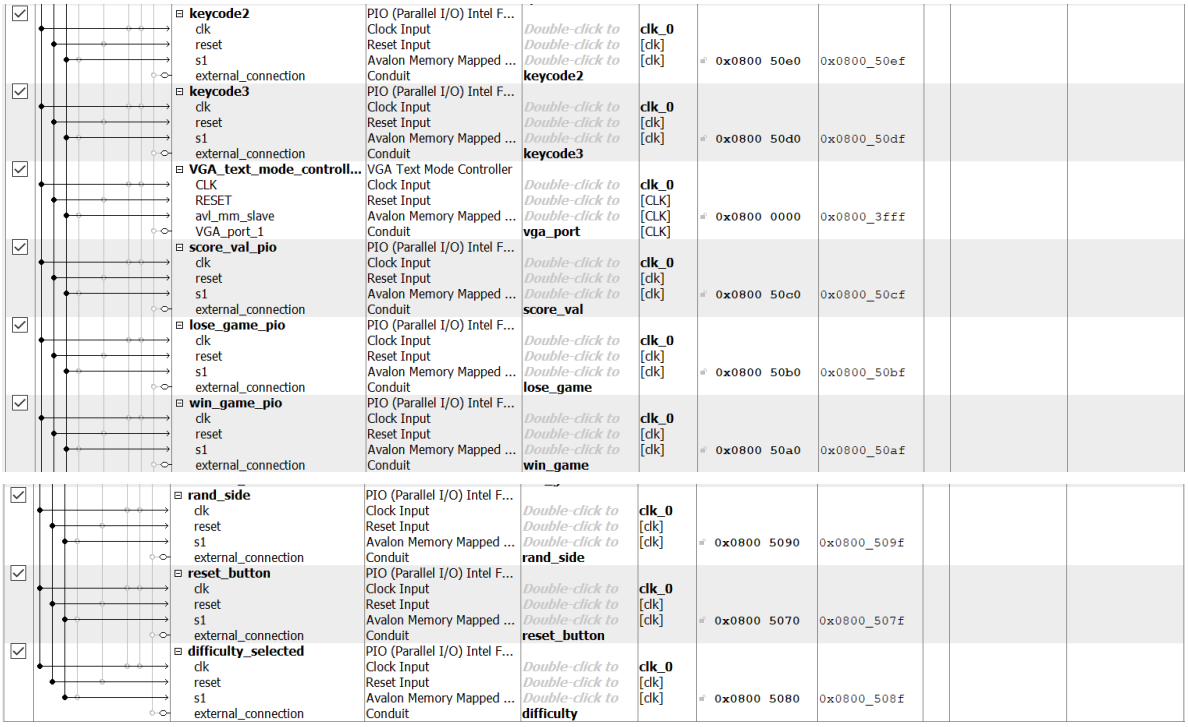
Module: Color Mapper (no inputs and outputs listed for conciseness)
Description: This module takes the X and Y values of all the instantiations of all the different things to be drawn on screen and outputs the correct RGB values at each pixel.

Purpose: The purpose of this module is to draw all the modules on the screen.

Platform Designer



Connections	Name	Description	Export	Clock	Base	End
	clk_0	Clock Source				
	clk_in	Clock Input	**clk**	*exported*		
	clk_in_reset	Reset Input	**reset**			
	clk	Clock Output	*Double-click to*	clk_0		
	clk_reset	Reset Output	*Double-click to*			
	nios2_gen2_0	Nios II Processor				
	clk	Clock Input	*Double-click to*	**clk_0**		
	reset	Reset Input	*Double-click to*	[clk]		
	data_master	Avalon Memory Mapped ...	*Double-click to*	[clk]		
	instruction_master	Avalon Memory Mapped ...	*Double-click to*	[clk]		
	irq	Interrupt Receiver	*Double-click to*	[clk]	IRQ 0	IRQ 31
	debug_reset_request	Reset Output	*Double-click to*	[clk]		
	debug_mem_slave	Avalon Memory Mapped ...	*Double-click to*	[clk]	0x0800 4800	0x0800_4fff
	custom_instruction_master	Custom Instruction Master	*Double-click to*			
	sdram	SDRAM Controller Intel F...				
	clk	Clock Input	*Double-click to*	**sdram...**		
	reset	Reset Input	*Double-click to*	[clk]		
	s1	Avalon Memory Mapped ...	*Double-click to*	[clk]	0x0400 0000	0x07ff_ffff
	wire	Conduit	**sdram_wire**			
	sdram_pll	ALTPLL Intel FPGA IP				
	inclk_interface	Clock Input	*Double-click to*	**clk_0**		
	inclk_interface_reset	Reset Input	*Double-click to*	[inclk_i...		
	pll_slave	Avalon Memory Mapped ...	*Double-click to*	[inclk_i...	0x0800 50e0	0x0800_50ef
	c0	Clock Output	*Double-click to*	sdram_...		
	c1	Clock Output	**sdram_clk**	sdram_...		
	sysid_qsys_0	System ID Peripheral Inte...				
	clk	Clock Input	*Double-click to*	**clk_0**		
	reset	Reset Input	*Double-click to*	[clk]		
	control_slave	Avalon Memory Mapped ...	*Double-click to*	[clk]	0x0800 50f8	0x0800_50ff
	usb_irq	PIO (Parallel I/O) Intel F...				
	clk	Clock Input	*Double-click to*	**clk_0**		
	reset	Reset Input	*Double-click to*	[clk]		
	s1	Avalon Memory Mapped ...	*Double-click to*	[clk]	0x0800 50b0	0x0800_50bf
	external_connection	Conduit	**usb_irq**			
	usb_gpx	PIO (Parallel I/O) Intel F...				
	clk	Clock Input	*Double-click to*	**clk_0**		
	reset	Reset Input	*Double-click to*	[clk]		
	s1	Avalon Memory Mapped ...	*Double-click to*	[clk]	0x0800 50a0	0x0800_50af
	external_connection	Conduit	**usb_gpx**			
	usb_rst	PIO (Parallel I/O) Intel F...				
	clk	Clock Input	*Double-click to*	**clk_0**		
	reset	Reset Input	*Double-click to*	[clk]		
	s1	Avalon Memory Mapped ...	*Double-click to*	[clk]	0x0800 5090	0x0800_509f
	external_connection	Conduit	**usb_rst**			
	keycode	PIO (Parallel I/O) Intel F...				
	clk	Clock Input	*Double-click to*	**clk_0**		
	reset	Reset Input	*Double-click to*	[clk]		
	s1	Avalon Memory Mapped ...	*Double-click to*	[clk]	0x0800 5080	0x0800_508f
	external_connection	Conduit	**keycode**			
	leds_pio	PIO (Parallel I/O) Intel F...				
	clk	Clock Input	*Double-click to*	**clk_0**		
	reset	Reset Input	*Double-click to*	[clk]		
	s1	Avalon Memory Mapped ...	*Double-click to*	[clk]	0x0800 5070	0x0800_507f
	external_connection	Conduit	**leds**			
	jtag_uart_0	JTAG UART Intel FPGA IP				
	clk	Clock Input	*Double-click to*	**clk_0**		
	reset	Reset Input	*Double-click to*	[clk]		
	avalon_jtag_slave	Avalon Memory Mapped ...	*Double-click to*	[clk]	0x0800 5100	0x0800_5107
	irq	Interrupt Sender	*Double-click to*	[clk]		
	hex_digits_pio	PIO (Parallel I/O) Intel F...				
	clk	Clock Input	*Double-click to*	**clk_0**		
	reset	Reset Input	*Double-click to*	[clk]		
	s1	Avalon Memory Mapped ...	*Double-click to*	[clk]	0x0800 50d0	0x0800_50df
	external_connection	Conduit	**hex_digits**			
	key	PIO (Parallel I/O) Intel F...				
	clk	Clock Input	*Double-click to*	**clk_0**		
	reset	Reset Input	*Double-click to*	[clk]		
	s1	Avalon Memory Mapped ...	*Double-click to*	[clk]	0x0800 50c0	0x0800_50cf
	external_connection	Conduit	**key_external_c...**			
	timer_0	Interval Timer Intel FPGA...				
	clk	Clock Input	*Double-click to*	**clk_0**		
	reset	Reset Input	*Double-click to*	[clk]		
	s1	Avalon Memory Mapped ...	*Double-click to*	[clk]	0x0800 5000	0x0800_503f
	irq	Interrupt Sender	*Double-click to*	[clk]		
	spi_0	SPI (3 Wire Serial) Intel F...				
	clk	Clock Input	*Double-click to*	**clk_0**		
	reset	Reset Input	*Double-click to*	[clk]		
	spi_control_port	Avalon Memory Mapped ...	*Double-click to*	[clk]	0x0800 5040	0x0800_505f
	irq	Interrupt Sender	*Double-click to*	[clk]		
	external	Conduit	**spi0**			



Description of Platform Designer Modules

Not all of the modules pictured above are used for Lab 7. The ones that are used are described below.

Clk_0 is a module that functions as the 50 MHz clock from the FPGA. This module outputs clk and clk_reset which are used as clock and reset inputs for most of the other modules

Nios2_gen2_0 is the 32 bit processor that is the centerpiece of the design. In this case, the version of NIOS II that is used is the NIOS II/e which is the resource optimized version of the processor. The processor sends a data output and instruction output to other modules.

SDRAM is the off-chip memory that is used in both labs. It has a size of 512 MBits and takes input from the data and instruction masters outputted by the NIOS II. An interesting thing to note about the sdram is that the input clock comes from the sdram_pll modules. This clock is phase shifted to allow the SDRAM to perform a data read or write in the correct window with respect to the master.

SDRAM_PLL creates the clock that is outputted to the sdram. This clock is delayed by 1ns to compensate for the difference between the master and slave clock phase.

Sys_id_qsys_0 checks for incompatibilities between the hardware and the software that are being connected. This is used as a preventative measure so that the FPGA is not overloaded and possibly ruined by the software that it is connected to.

Timer_0 is a 1-bit module that sends a signal every millisecond so that the NIOS II is able to keep track of how much time has passed.

Spi_0 is a module that allows for data transfer between master and slave and vice versa through a MISO and MOSI channel. This can be done simultaneously depending on the mode of the SPI. The SPI also has a slave select that is active low. This is the part where our ‘four function’ controlling reading and writing bits was implemented.

VGA_text_mode_contoller_0 is a user created IP that implements the AVALON bus. This module is also responsible for sending the RGB signals to the VGA output via VGA_port.

Keycode, Keycode2 and Keycode3 are 8-bit PIO outputs to pass up to three different keycode values from software to hardware.

Lose_game_pio is a 1-bit input PIO to tell the software when all the lives have been lost so that the lose game screen can be displayed.

Win_game_pio is a 1-bit input PIO to tell the software when the boss is defeated so that the win game screen can be displayed.

Score_val_pio is a 10-bit input PIO to tell the software what the current game score is so that it can be written to the screen.

Rand_side is a 1-bit output PIO to pass a random 0 or 1 to the hardware to spawn enemy waves on a truly random side.

Reset_button is a 1-bit input PIO to pass the value of the reset button to the software so that the game can be properly reset with only pushing the reset button.

Difficulty_selected is a 1-bit PIO input to tell the software whether the game has been started. This is done so that the start screen will be displayed until difficulty is selected.

Software Description

We wanted to complete the vast majority of our project using System Verilog. However, we decided to implement our I/O USB keyboard control from lab 6, as well as the combination of C code with the vga_text_avl_interface from lab 7 to support our game. In addition, we included our high level game FSM in C.

When signals are passed to System Verilog, it means the IOWR_ALTERA_AVALON_PIO_DATA function was used to write information to the corresponding SOC addresses. When signals are read from System Verilog, IORD_ALTERA_AVALON_PIO_DATA was used to access the information at an SOC address.

Vram_stuff.c

textVGAColorclr : Writes 00, a blank space in final_font_ROM, to memory locations to erase the entire screen and cover the screen in black besides sprites.

textVGADrawColorText : Takes a string, x and y coordinate, and background and foreground color and writes it into memory for VGA_text_avl_interface to draw on screen

setColorPalette : *S*elects all the colors that we can write the foreground or background of text.

drawGameScreen : Initializes the color palette with setColorPalette and writes “Score: “ and “Lives: “.

Update_score : Takes in the new score and calls textVgaDrawColorText to write the new value on screen.

Reset_score : Called when the game resets and erases the score and writes “---” in place of a score.

Drawstars : Creates a randomized star pattern to act as a background for the game.

DrawStartText : Calls textVgaDrawColorText to write the title of the game, directions, and the authors of the game. Also calls Drawstars to create a background for the start screen.

DrawLoseText : Calls textVgaDrawColorText to write the lose game screen text.

drawWinText : Calls textVgaDrawColorText to write the win gam screen text.

MAX3421E.c

MAXreg_wr and *MAXbytes_wr* are used to write values to registers while *MAXreg_rd* and *MAXbytes_rd* read registers or bytes. These are necessary for communication with the USB keyboard. For more information refer to the SOC and NIOS II in System Verilog lab.

Main.c

setKeycode : Takes in USB keyboard key inputs and passes to System Verilog.

setScoreVal : Reads the score from System Verilog.

setLoseGame : Reads the lose game signal from SystemVerilog.

setWinGame : Reads the win game signal from SystemVerilog.

setResetButton : Reads whether the reset button has been pressed from SystemVerilog.

setStartGame : Reads whether a difficulty has been selected from SystemVerilog.

Main : Initialized the MAX3421E and USB, then begins the game FSM from figure 1.

Run_game : calls drawStartText to draw the start screen. Then enters an infinite loop which updates the score value when necessary, as well as whether the game has been lost, won, or reset, in which case the corresponding screen will be drawn. At the same time, the program continually polls the keyboard for inputs and passes the key presses to SystemVerilog. When a difficulty has been selected, pass a random value to SystemVerilog to determine which side enemies should spawn from.

Design Process

We built this project off of lab 6, which consisted of a keyboard I/O to control a ball which would bounce off the edge of a VGA monitor screen. We started with player mechanics. The I/O was adjusted to account for multiple key presses at once so players could do things like move diagonally, or move and shoot at the same time. Multiple player lasers were also created which would spawn on the player and shoot upwards. One enemy was also created to laterally traverse and bounce off the screen, however, no interaction between the enemy and player and its lasers existed.

In the second week, 2d collision was implemented. This could have either existed in the cannon module or the enemy module but, we ultimately decided to place it in the enemy module because enemy ship collision would have to exist in the enemy module and it would be better to have all the collision logic in the same module. While the collision detection wasn't difficult, the various signals became more complex. All four laser coordinates needed to be passed to the enemy ship for collision detection, and if detected, four different hit signals were needed to distinguish which laser had collided to stop drawing it to the screen. So not only did we need the X and Y coordinates, but also an 'exists' signal which determined whether or not we needed to draw something to screen.

After getting one enemy working, we attempted the same logic with a second enemy. Once we were able to iron out some logic bugs and split other signals the second enemy was working properly. At that point, we were confident the written code to support multiple enemies and the rest of the normal enemies just became simple copy and paste in the top level module with small signal changes where necessary. The reason we started this design process with one is to first verify that the code does what we want it to and debugging one enemy is easier than debugging multiple. The step to two enemies allows us to make sure the code is compatible with the lowest number of multiple enemies. By doing this, this allows us to debug the simplest version of each case without having to deal with the complexity of having many enemies on screen at once.

We had envisioned that after a while of lateral bouncing, the enemies would all synchronously fly downwards off screen. The problem with placing this logic in the enemy module would be that all enemies would follow different clocks since they spawned at different times. The enemies would all flydown once they reached a certain point on screen rather than a certain time. To solve this problem, we created a FSM to control the enemy spawns and flydown timer. We created four waves, each of which had four separate spawn signals at different times to help space out the enemies. Each wave would then go to a flydown state which consisted of a timer, which would allow time for the spaceships to bounce around on screen before having to all synchronously fly downwards offscreen and the next wave would start spawning.

After the enemy controller was created, we started implementing other types of enemies. First on the list were enemies that would shoot back. We originally started writing a new module before we realized that similar to the player ship and player lasers, the laser module got the shoot signal rather than the player ship. So, the enemy shooters could

simply use the same module as the regular enemies and we could create new enemy laser modules which would receive shoot signals. We had another choice here to place the collision logic in the player ship or in the enemy laser module. We decided we wanted to keep the playership module as clean as possible and keep collision logic centralized in enemy modules so we placed this logic in the enemy lasers module. Another thing we needed to look at was how we could randomize the shooting. We couldn't use a single synchronous signal from the enemy controller FSM because we thought that would be a bit boring and creating various signals from the enemy controller, similar to having four different spawns in a wave, would also end up being very periodic and predictable. The solution was to use a random number generator. The choices were between true randomness, using C code and the srand and rand functions to pass down a shoot signals but we decided that would increase latency, take up more space on our SOC, and true randomness wasn't really necessary for this. Instead we used multiple linear feedback shift registers, a pseudo random number generator designed for hardware, which was a perfect fit for what we needed. The shooting was random enough to be unpredictable without using excessive resources towards adding SOC components. Again, similar to normal enemies, we tested a single enemy shooter. Unlike last time however, since we already figured out all the needed additional signals for multiple types of enemies, we didn't need to test two and we're able to just replicate the modules multiple times in the top level with minimal testing.

After the enemy shooters were created, we began looking at our third and final type of enemy, a tracking enemy which followed the player ship. Since the movement of this enemy is different from the other two, we had to create a new module. With the experience from making the other enemies, this enemy was rather straightforward and not many complications arose.

Once all the enemies were created we began looking at the user interface. We began with looking at difficulty. To do this, we had to include a state in the enemy controller FSM to not start spawning until after a difficulty was selected. We also had to include health logic on each of the enemies. We also included a lives and score interface as well. We connected the player ship collision signals to a health tracker which when health reaches zero, a signal would be sent out. At the time, we did not connect the signal to anything as we planned to have the signal control the lose game screen in the future. We also connected the player laser hit signals to a score module to keep track of how many hits a player got to calculate a score. Both of these signals are then exported to the SOC and NIOS II processor for future use.

To create the start game, win game, and lose game screens, we utilized the vga_text_avl_interface we created in lab 7. We use C to set color palettes and write text into memory for the vga_text_avl_interface to read from and display the score, directions, and any text we wanted to display. Using C we also wrote our high level game FSM. We used the difficulty selected from earlier to start the game which could then progress to a lose or win game screen which would display different text. The reset button was also set as an export from the DE10 board to send the game back to the start game screen at any time.

Since these screens required different text on screen, we decided not to implement this FSM in SystemVerilog since the SystemVerilog modules couldn't write text to memory easily.

After creating different screens, we created various power ups. We used the same collision logic and placed it within the powerup module again to keep with our pattern of keeping collision outside of the playership module. We utilized the LFSRs from earlier to create a pseudorandom pattern of different power ups including invincibility, speed ups, double score, and extra lives, as well as pseudo random drop locations. Again similar to enemy shooting, we didn't need to access true randomness here as pseudo randomness was efficient.

Unlike enemy shooting and the power ups, we decided that there should be some variation in enemy spawns which shouldn't be conducted with pseudo randomness. Unlike the other two which happen frequently enough that it would be hard to tell the difference between the two different randomness, we decided the side enemies spawn from each game should be true random. We included a random number generator based off of when the player selected a difficulty to determine the side.

With power ups and enemies created, we decided to include a special boss at the end of the game. Based on similar logic from the enemies, we created the outline of a boss. However, we wanted the boss to have more complex attacks, which we conducted through the creation of a boss FSM which controlled when the boss would appear, fly around, shoot, and flydown. We created a new type of beam projectile based off of LFSRs to determine where to shoot from, while the difficulty controlled how frequently the beam shot. However, after completing the boss, we wanted the boss stage to have more difficulty and we altered the enemy controller FSM so based on the difficulty, a certain wave of enemies would continually repeat until the game was won or lost.

The final step in our game design process was implementing sprites for the player ship, enemy ships, boss ship, and for an explosion animation. Up to this point, all the logic of our game had been tested using rectangles and circles. Creating the sprites was greatly helped by the provided python code found on the ECE 385 wiki which converted pngs to a series of numbers which would represent pixel colors. We first chose our desired sprite and then ran the image through the PNG to Palette Resizer program to resize our sprite to the correct size. This program outputted a .txt file which contained a list of color indexes for all the colors present in the sprite in raster order. These colors were loaded into a color palette, specific to each sprite, and the correct color is chosen to be drawn by the location of the sprite and the location of the DrawX and DrawY signals. Because this program made adding sprites quite simple, we were able to create a Star Wars theme for our game.

Design Resources and Statistics

LUT	32205
DSP	0
Memory (BRAM)	49664 bits
Flip-Flop	4707
Frequency	76.23 MHz
Static-Power	97.25 mW
Dynamic Power	178.07 mW
Total Power	334.72 mW

Conclusion

By the end of this project, we had created a fully functioning space shooter and themed it to a classic movie series. We implemented many commonly found features found in arcade games such as power ups, a boss stage, score counter, and lives displays. This project was an extremely rewarding experience that taught us a lot about randomization in hardware and interfacing in an efficient way between software and hardware. It also forced us to use previously learned skills in System Verilog in creative ways so that our game behaved in the desired way.

In its current state, X-Wing Attack is an extremely functional and playable game. However, there are some possible improvements that could have been made to improve the game experience, given more time. Adding sound effects and music would be beneficial, as almost all modern video games have some sound component. The game could also improve upon its difficulty balancing by making enemies move in a more random manner and adding more waves of enemies. Finally, more enemy and player lasers could be instantiated so that there is not a limit to how many lasers are on the screen. The laser issue is not one that is particularly noticeable but fixing it would add more opportunities to power up the player cannon.

Overall, this project was an invaluable part of the ECE 385 course. It forced us to think in both creative and practical ways while we balanced our lofty ideas for our game with what was realizable in the given timetable.