# ECE 385

Spring 2019
Final Project

# Space Invaders Clone in Systemverilog

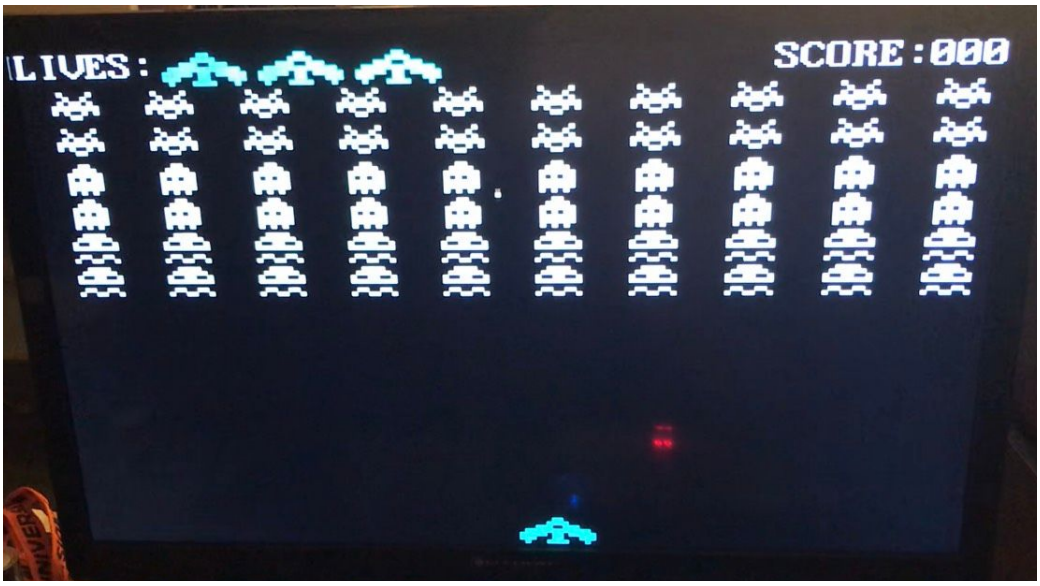
Brock Boehler & Yaning Lan
ABL Thursday 8:00am - 11:00am
TA: Vikram Anjur & Xinying Wang

## Introduction

For our final project, we set out to create a hardware clone of the popular arcade game Space Invaders. Our version is a bit different and a bit simpler, but still just as fun. Using a keyboard, a player controls a spaceship, located at the bottom of the screen, and fires missiles at an array of 60 alien enemies at the top of the screen. The enemies can also fire missiles back at the player, and when hit the player loses a “life.” The game can end on two conditions: whenever the player defeats all enemies (the player wins) or whenever the player runs out of lives (the player loses).

The game “machine” can be in one of four states. Upon boot, the machine enters a wait state the game will wait and prompt for the player to hit the space key. When the player does so, this enters the next state, which is where the player plays the game. The next two states are either when the player wins or loses.

The machine utilizes an EZ-OTG USB chip to communicate with the USB keyboard, a NIOS II processor to make an SOC, and an FPGA as glue logic, a game controller, and VGA controller.




*Space Invaders Gameplay Screenshot*

## Written Description of Circuit

The space invaders game machine is comprised of multiple modules working together to create the end result - there is no “primary” module to coordinate the behaviour of all on-screen components. Although one may expect the state machine we have included to be the overarching “boss,” it is really only responsible for deciding what needs to be printed to screen and whether or not the enemy array can attack the player. Every item, such as the spaceship, the enemy array, the missiles, etc, all are self-controlled entities but share information about themselves, such as location, with each other so they can react appropriately and semi-intelligently.

All these entities share the necessary information about themselves to the color mapping module, which is responsible for deciding how things should be printed to the television screen. All on-screen items have an associated sprite located on FPGA memory, from text to enemies. To help us put things onto the screen, we utilize modules that we named text-mappers and

pixel-mappers. Given an X and a Y coordinate, these modules will return the color of the pixel for that item at that point. More information is given below.

The machine's modules can all be grouped into the following categories: the state machine, keyboard related modules, enemy controllers, player controllers, text and pixel mappers, and video printing modules. For a graph showing how they all connect, see the block diagram later in this report. Each section of modules is discussed as follows.

State Machine

The state machine is responsible for controlling the machine, albeit very loosely. There are 4 states: The initial "press space" state, the gameplay state, and the two ending states ("you win" and "game over"). The state machine only affects two things, that being what is displayed on screen and whether or not the enemies are allowed to fire missiles. Even when the gameplay is not displayed, the enemy array, score, lives, etc all operate as regularly just without being displayed. This is done by feeding the state into the color mapping module, which only displays the game when in the correct state. To prevent the player from being unfairly killed before the game begins, the enemy array's missiles are disabled while in the initial "press space" state. Upon pressing space, the enemy gets a "free" shot at the enemies and the game begins.

After the game is over, either by the player defeating all enemies or the player runs out of lives, the state machine moves to the winning or losing states and prints the associated text to screen. Whenever the use presses reset on the machine, the game goes back to the initial screen and the game can begin anew.

*See the state machine diagram later in the report for a diagram of connections and more detailed state descriptions.*

Keyboard Input

In order for the user to be able to actually play the game, we need some sort of human interface device. To accomplish this task, we use a USB keyboard for playing the game. This is a relatively complex part of the system despite being only used for player input. We use a NIOS II SOC connected to an EZ-OTG USB chip that makes the process a little easier. The NIOS II then relays the key presses back to the FPGA via the hpi_io_intf module. Once we have the key press, we can then pass it to the game machine.

*See the following section for more detail in regards to software methods used, how the USB works, and an addendum on how VGA works.*

Enemy Control

In order for our game to be any fun, we have to have some kind of enemy to shoot at. And what makes a better target than an evil assembly of 8-bit aliens? The enemies are arranged in a 6-tall, 10-wide fashion, with the space between all enemies being the width of one enemy itself. Because the aliens all move as a unit, we need only to store the top left corner of the array. We can then calculate the position of each enemy based of this information. Because each enemy printed on screen is 32 x 32 pixels, the math for calculating which spots of the screen the enemies lie is trivial (this is important for collision detection): Take any pixel's normalized X and Y position (X position - top left X coordinate of the enemy array, Y position - top left Y coordinate of the enemy array), divide it by 32 (ie bit shift right 6 bits) and if the result is even for both X and Y, then that space is to be occupied by an enemy.

Now that we have a method for calculating where enemies belong on screen, we want them to move back and forth across the screen. This is trivial, and simply involves moving the top left X coordinate back and forth a small amount every tick of the clock (we use vga vertical sync for our “clock”). Using the previously established logic, this automatically updates the position of all the enemies in the array.

We have our enemy array and a way to move them around. Now we need a way to record which enemies are still alive. This is easy, we just use a modified two dimensional register-like system to record who’s still alive. Whenever a player missile hits an enemy, we simply set that bit in the array to 0 to specify that the enemy has been killed. Detecting collision is very easy. Using the logic formulated earlier, we can take the player’s missile’s X and Y coordinate, normalize them with respect to the top left corner of the enemy array, divide it by 32, and if this number is even (and the X and Y coordinates are inside the enemy array) we know that the missile is in a spot that is occupied by an enemy. If this is the case, we can then divide the coordinates again by 2, and by subtracting a small offset we can calculate exactly which enemy’s box the missile is in. Checking if this enemy is still alive using the enemy status register, we can then detect if we’ve hit a living alien, and then update it’s status. Whenever the enemy register is completely empty, this means that the player has won the game, and the state machine is updated.

At the same time, the enemies need a method of attacking the player. Because of this, enemies have the added ability to fire missiles back at the player. Deciding which enemy should fire the missile when was a challenge. We decided that every two seconds a missile should be fired by the enemy, and that the enemy closest to the player in the X dimension should be the one who fires. Calculating the closest enemy is relatively simple, using the same logic as before. Take the player’s X position minus the top left X position of the enemy array, set the lower 6 bits to 0 (this gives the X position of the left side of the nearest enemy), and then add a 16 bit offset (to get to the middle of the enemy). Then to determine the Y position, we look at the enemy status register to see who’s remaining in that column. We then choose the lowest enemy, and have him shoot the missile from his Y position. The missile begins at the bottom of an enemy and travels down the screen (i.e. it’s y position is increased)  until it either goes off screen or hits the player.

Player Control

If a human is to play a video game, he or she needs some way to manipulate the on-screen character. For our game, the player controls the spaceship at the bottom of the screen. Whenever the player pressed a or d on the keyboard, the player moves left or right, respectively. Whenever the player presses space, the spaceship fires a missile upwards towards the enemy array, and collision is detected as previously discussed.

Detecting whenever an enemy missile hits the player is also fairly straightforward. Whenever the enemy missile is within the hitbox of the player (ie below the top of the spaceship and in between the left and right bound) the player’s life count is decremented. Whenever this reaches 0, the game ends as the player has lost.

Text Mappers / Pixel Mappers

To aid in drawing all the on-screen elements, so-called text mappers and pixel mappers are used. Given an x and y position as input, these mappers output what color the pixel should be at that given location for that given element. For example, the spaceship has its own pixel mapper. When given position x=0 and y=0 (meaning the top left of the spaceship sprite), the pixel mapper returns 0, as the screen should be blank / black at that position. This is done with

the enemy sprites, the space ship, all text, etc. These are all utilized by the color mapper to draw on-screen elements, as discussed in the next section.

Drawing To Screen

The color mapper module is by far the largest, most complex module in the entire system. And although it is “complex” it is conceptually very simple, as it is practically just nested conditional statements. The first check performed by the color mapper is the state. If the state is not the gameplay state, the color mapper prints the associated text of that state in the middle of the screen with the help of a text mapper. These are relatively simple: check what state we’re in, and then use a text / pixel mapper to decide what colors to print when we’re in the middle of the screen.

Whenever in the gameplay state, the printing becomes more complex. Given a drawX and a drawY position, the color mapper does a series of checks to locate if we’re printing something in a specific “bounding box.” For example, using a series of conditional statements we can determine that we are printing something in a specific box in the top left of the screen. Because we know we’re there, we know to print the “LIVES:” text. So then we draw this text using the help of a text mapper at this location. This happens for all on-screen elements. We determine if we’re in the bounding box for a specific element, and then we use a mapper to print the element at that specific location. These bounding boxes can be static, for example when printing non-moving text such as score, “LIVES:”, “SCORE:”, etc. Other elements’ bounding box can be dynamic, such as the player and enemies, which move around.

The color mapper takes a wide variety of inputs, namely all data about the position of the enemies, which enemies are alive, how many player lives are left, what missiles are present and where they are, and where the player is. We use what enemies are left to know what enemies to print and what to not, we use position to develop bounding boxes and then print the elements, we use player lives to specify how many spaceships to display at the top of the screen, etc.

After the color mapper does it’s job and maps each pixel, it is then passed onto the VGA module which passes through it a D2A converter so it can be used with a television. More information on VGA can be found in the next section.


*“GAME OVER” printed to the screen upon losing the game, with the help of a text mapper.*

# USB Description, VGA Description, and Code Description

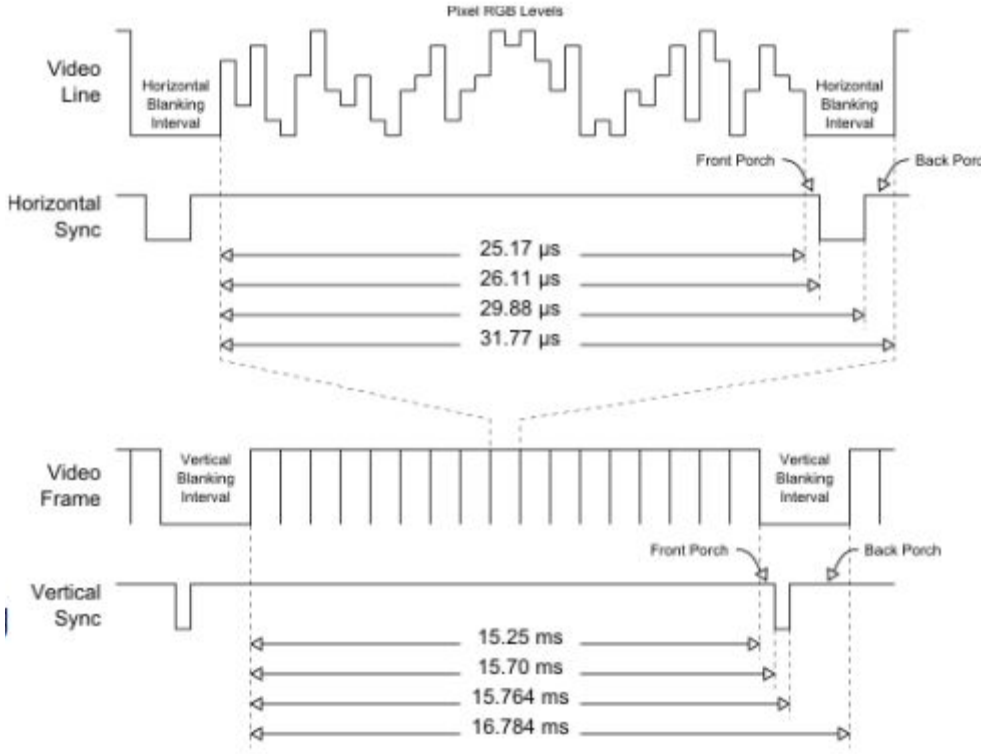
Communication with USB Keyboard

In order for our DE2 to communicate with a USB keyboard our board will act as a host device, which means that it is responsible for controlling all connected devices. Because our USB keyboard will not send data on it's own, our DE2 will be responsible for polling the keyboard to get the status of which keys are currently being pressed. We will send interrupt requests repeatedly, to which the keyboard will reply with report descriptors, which has the desired information.

Communication with VGA Hardware

VGA is a protocol for hardware to communicate with a video screen, and send video data to appropriate devices. In our case, we set the screen to 640 x 480 pixels. VGA communicates via analog voltages to control an electron beam (which can be real or simulated in devices such as LCD screens) which scans across each row and column to draw pictures and images. Our VGA controller uses vertical sync and horizontal sync to enforce synchronization with the "electron beam."

The VGA hardware operates on about a 25 MHz clock which is used for timing the writing of each individual pixel, and overall results in about a 60 Hz refresh rate for the entire screen. In between the pulses of horizontal and vertical sync, analog RGB levels are fed into the device at a specific frequency to represent each and every pixel. The operation is nicely explained by the following diagram:



DE2-115 USB Connection Method

Our FPGA and NIOS II process are not solely responsible for handling communication with the USB keyboard. To help us with all USB-related endeavors, the DE2 has a built in Cypress EZ-OTG USB Controller. This chip is, in itself, a small computer system with a RISC processor, RAM, ROM, and direct memory access. In order to communicate with the EZ-OTG, a selection of four specific registers are read and written to in order to send commands to the EZ-OTG, write data to the USB, read data from the USB, service USB interrupts, etc. Writing to the 2 address lines connected to the chip selects the desired register, and setting chip select, read, and write selects which action to take upon each register. The four registers are:

| | |
|---|---|
| HPI DATA | Used for moving data to and from the RAM of the EZ-OTG. Whenever the read or write lines are set low, this register is loaded with or written to the RAM address specified by the HPI ADDRESS register |
| HPI MAILBOX | Relays messages between the EZ-OTG hardware such as ACK, NACK, etc |
| HPI ADDRESS | Holds the address that read or write operations are to take place on in the on-board RAM of the EZ-OTG |
| HPI STATUS | Holds the current status of the EZ-OTG chip, such as if devices are connected |

These four registers are used in tandem to perform operations on the USB hardware. For example, suppose we want to write data to the EZ-OTG RAM at a specific location. To do this, we would write the address to the HPI ADDRESS register and then write the desired data to the HPI DATA register.

Our hardware communicates with these registers via 5 lines as follows:

| | |
|---|---|
| HPI nRD | Active low read line. When pulled low, the contents of the register selected by the address pins is output through the data pins |
| HPI nWR | Active low write line. When pulled low, the contents of the register selected by the address pins is loaded with the value of the data pins |
| HPI nCS | Active low chip select, when low this activates the EZ-OTG |
| HPI A0, A1 | Address pins, selects which register to read or write to |
| HPI D | Data pins, 15-bits wide. Allows for data to be piped in or piped out of the system |

Looking at a diagram of the EZ-OTG chip, we can get a better view of how it is put together and what exactly is connected to our system (our system, in the first diagram is the “external master”):

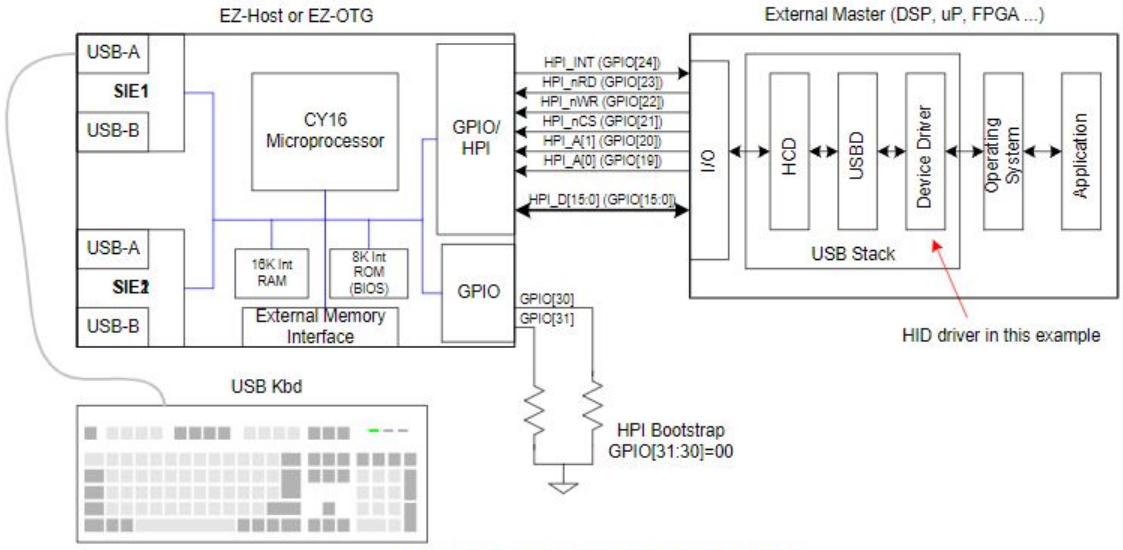



**Figure 1. Host Port Interface (HPI)**

The actual EZ-OTG is shown here in more detail, showing physical connections with all associated devices:

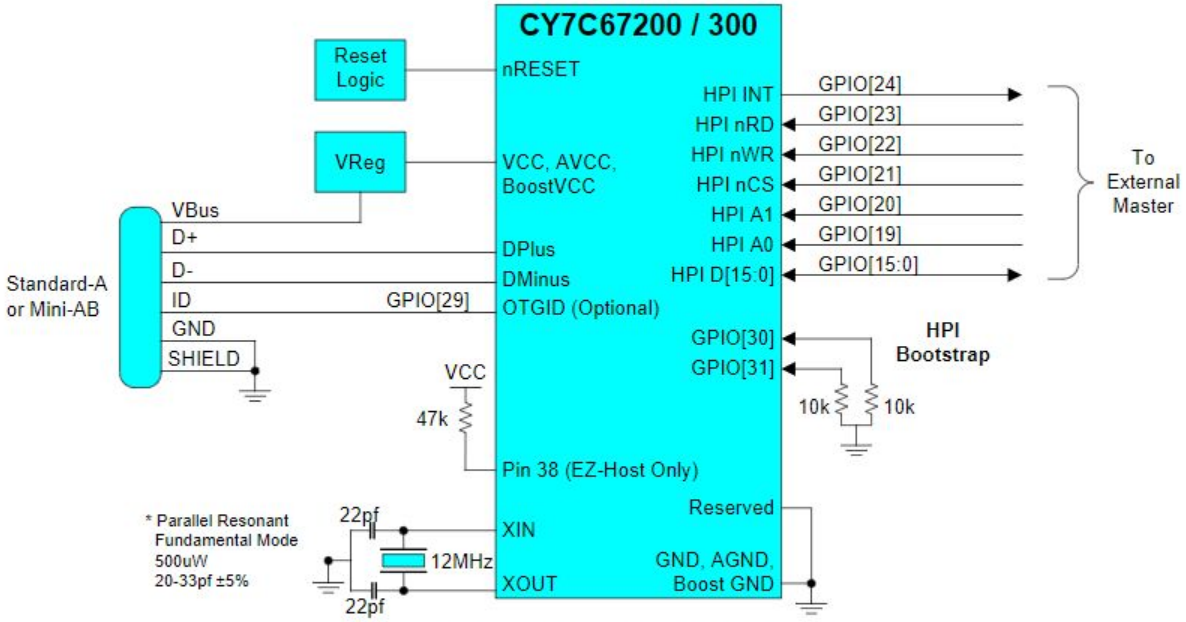


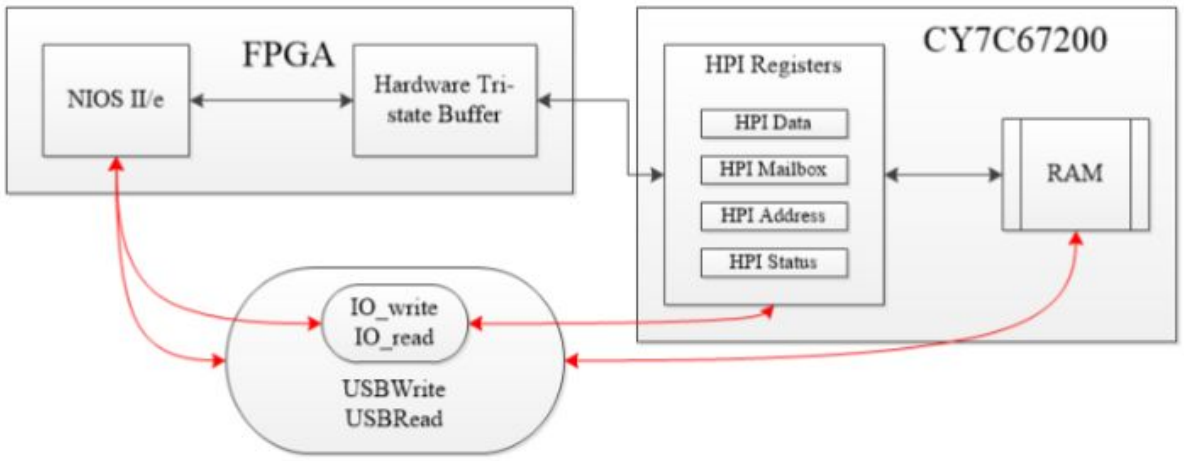

**Figure 2. Example OTG-Host HPI Schematic**

Software Methods used in Final Project

Now we can finally discuss implementations on how to move data about. To do this, the NIOS II processor uses the following methods to move data to and from the USB devices:

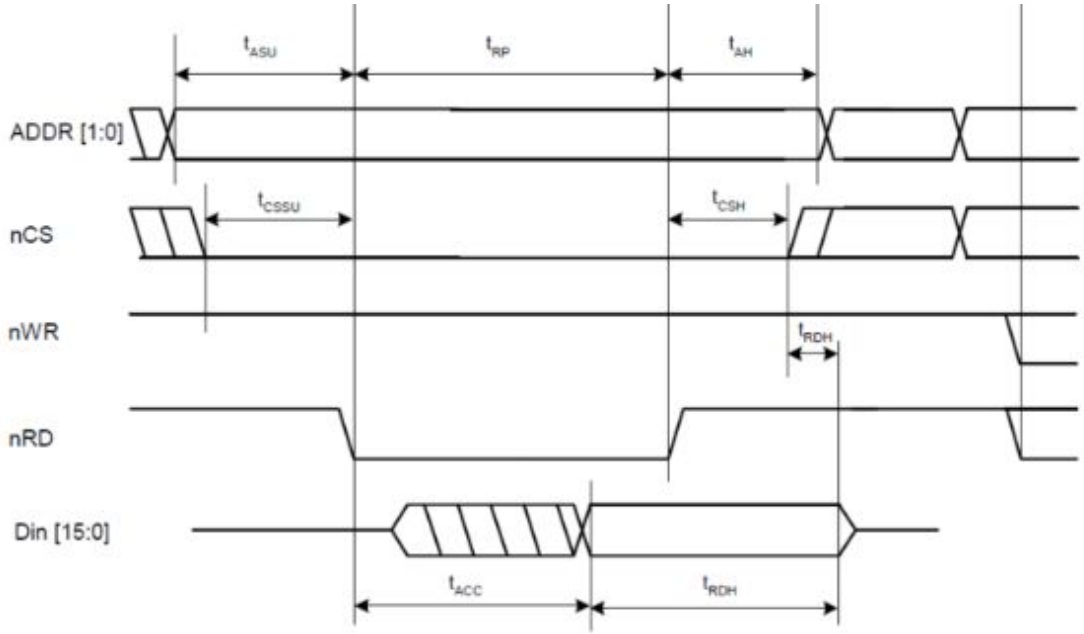
| | |
|---|---|
| IO_read(Address) | IO_read is responsible for reading the contents of one of the four registers. When called, it returns the value of the register specified by Address. Address here is the 2-bit argument that specifies the register, not a location in the EZ-OTG RAM. |
| IO_write(Address, Data) | IO_write is responsible for writing values to one of the four registers. When called, the register specified by Address is filled with whatever is contained in Data. Here Address means the 2-bit argument that specifies the register, not a RAM location. |
| USB_read(Address) | This is responsible for reading data from RAM in the EZ-OTG. Here Address means a location in RAM, and not a register. To do this, the Address argument is written to the HPI ADDRESS register, and then data is read from the HPI DATA register |
| USB_write(Address, Data) | This is responsible for writing data to RAM in the EZ-OTG. Here Address means a location in RAM, not a register. To do this, the Address argument is written to HPI ADDRESS register, and then data is written to the HPI DATA register. |

The aforementioned functions are nicely summarized by the following graph:

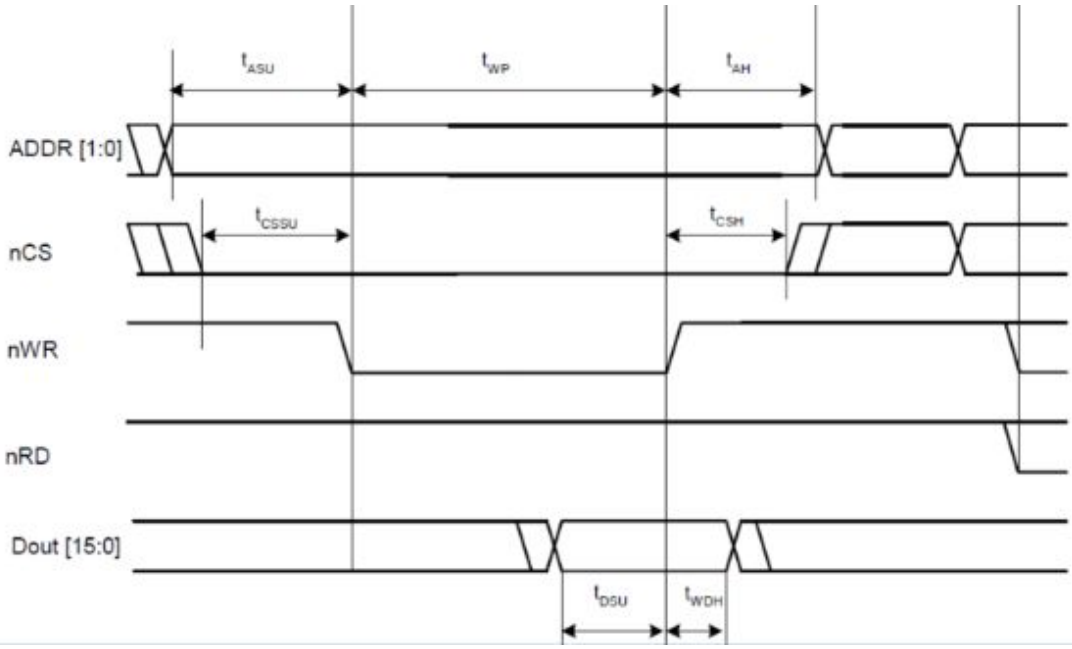


In relation to the previously mentioned registers of the EZ-OTG, the reading and writing functions manipulate the io pins on the chip in a very specific manner to get the desired result. The timing and desired values are shown in the following charts:

Reading data from the EZ-OTG




Writing data to the EZ-OTG

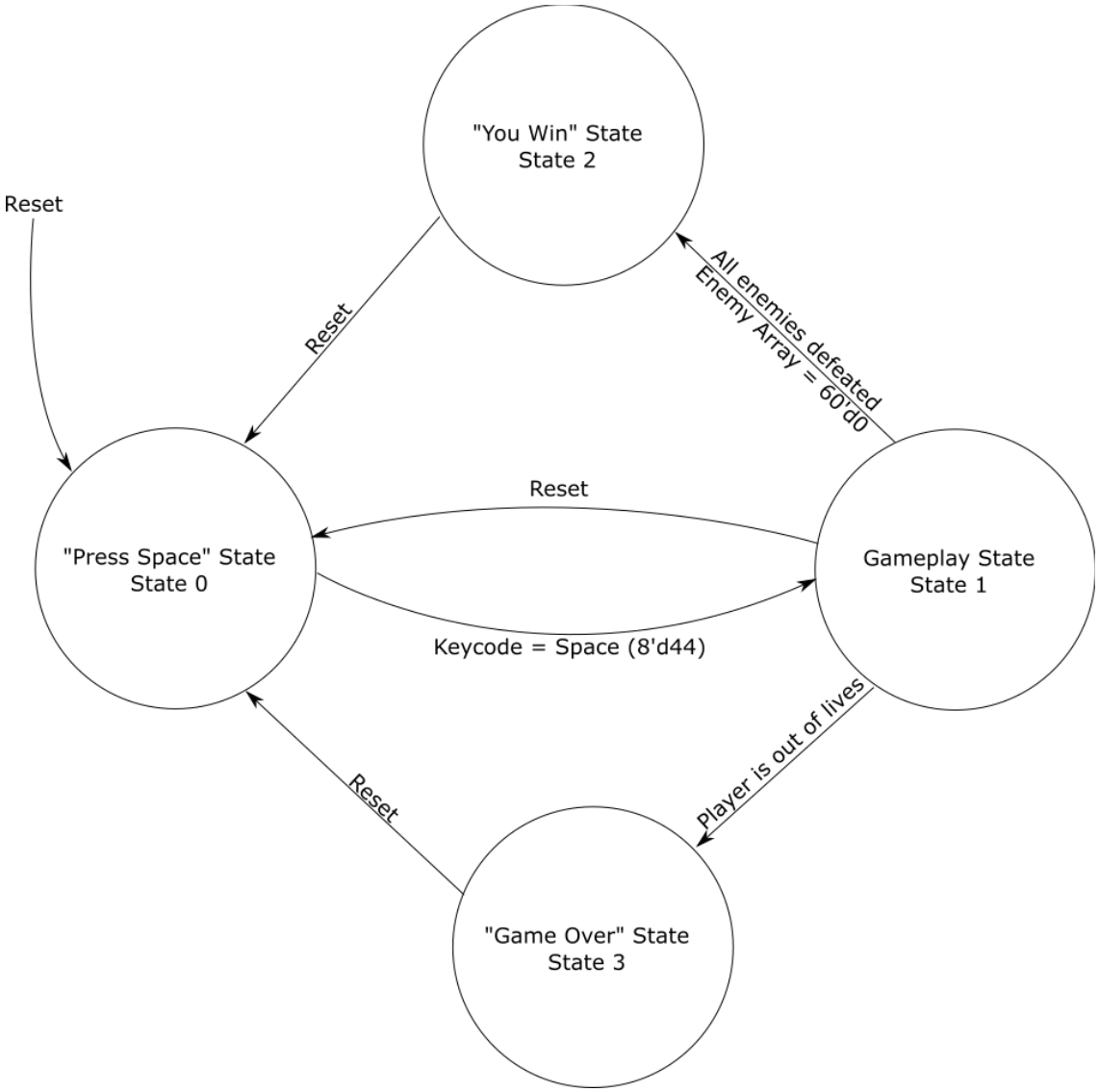



Now that we have these two methods to communicate with the EZ-OTG, we can now specify how reading and writing to the USB works.

Writing: Write the address to the address register, and then write the data to the data register

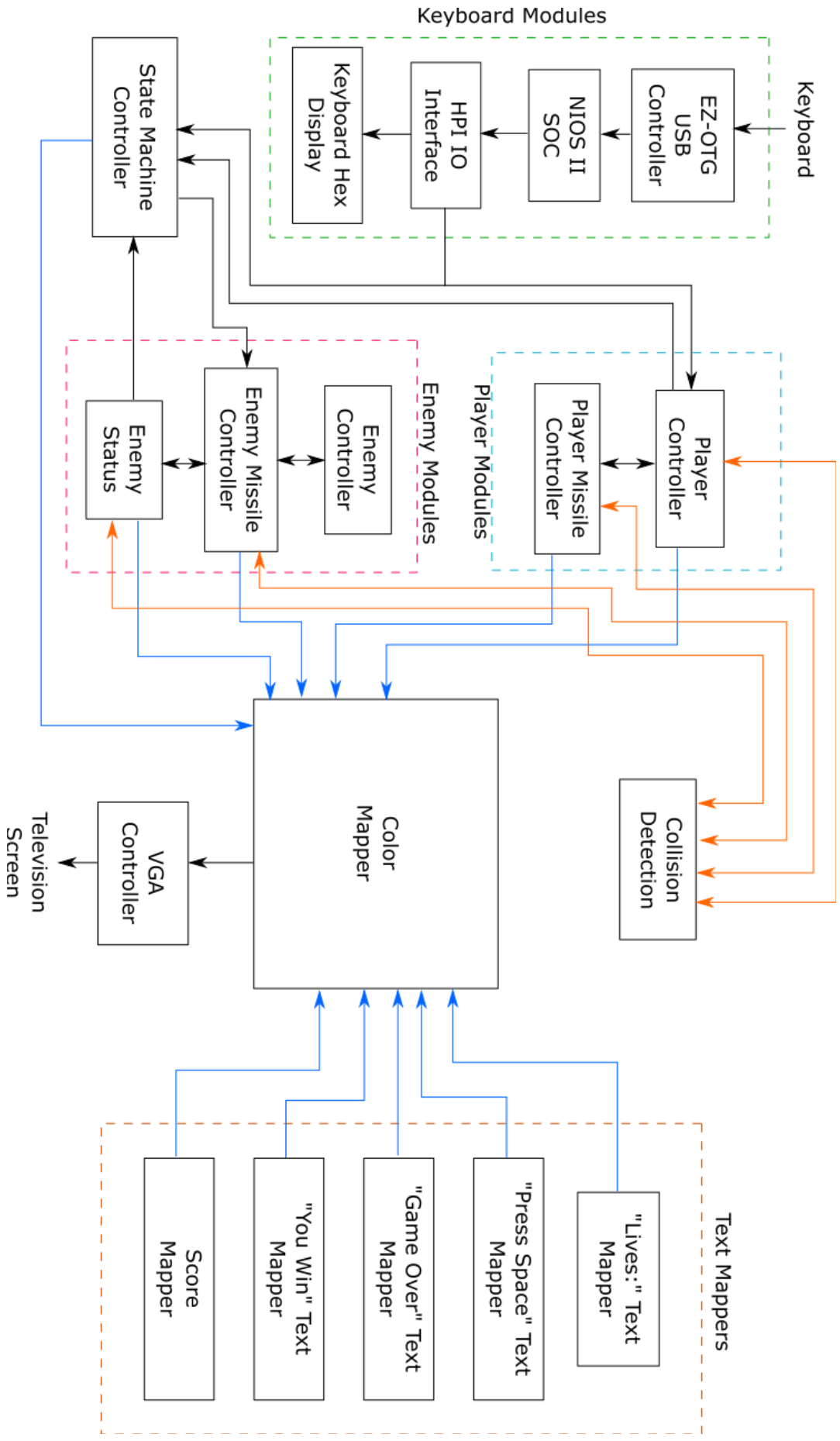
Reading: Write the address to the address register, and then read the data from the data register.

# State Diagram & Description



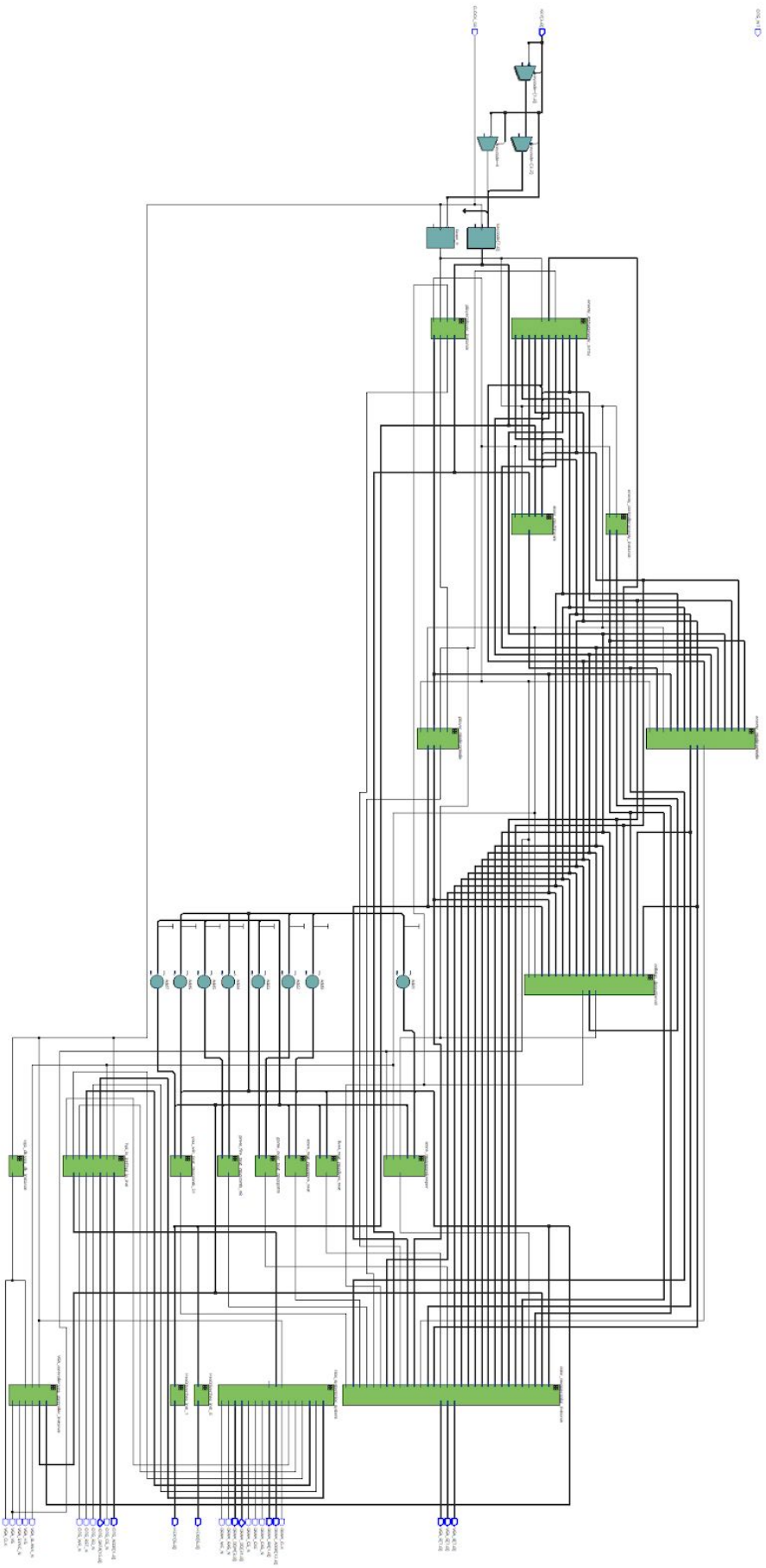

| State | Description |
|---|---|
| State 0: “Press Space” State | The initial state of the game. Upon boot, the screen is black save for centered text saying “Press space”. Upon doing so, the game begins. |
| State 1: Gameplay State | The state in which the game is actually played. The player has full control over the ship and continues until all enemies are defeated or the player dies |
| State 2: “You Win” State | Upon defeating all 60 enemies, the screen goes black and displays the text “You Win” |
| State 3: “Game Over” State | Whenever the player runs out of lives, the screen goes black and displays the text “Game Over” |

Block Diagram

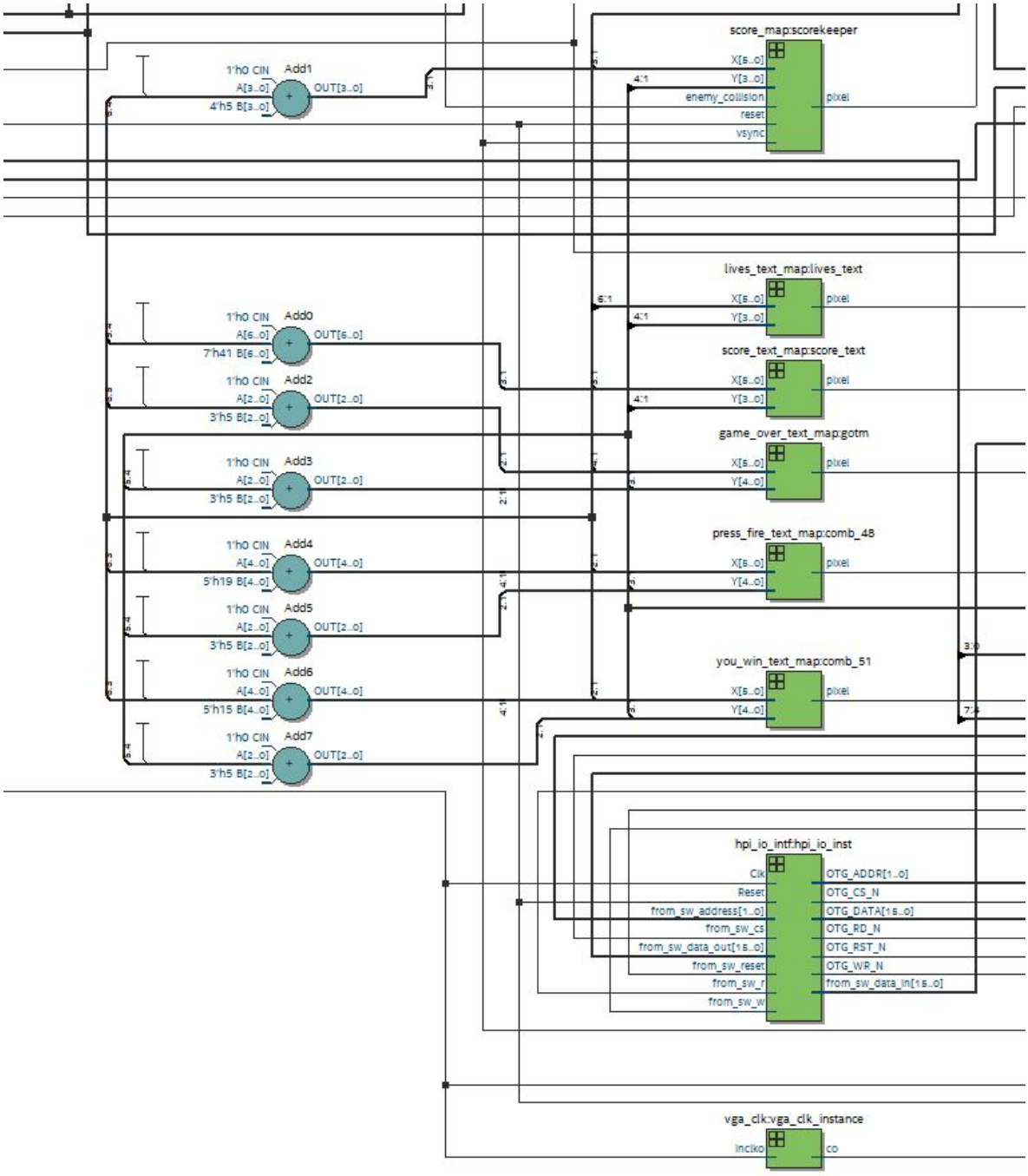


# Top-level RTL Block Diagram

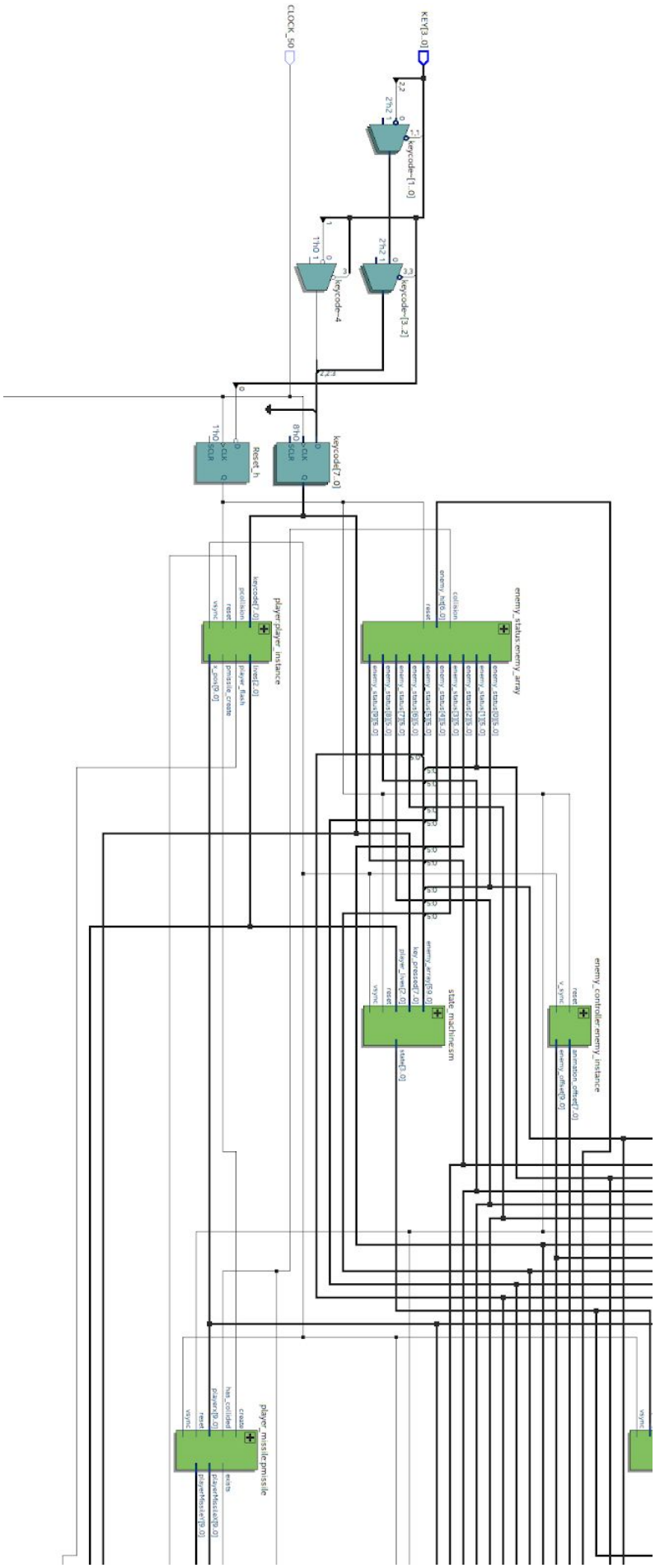
Whole System



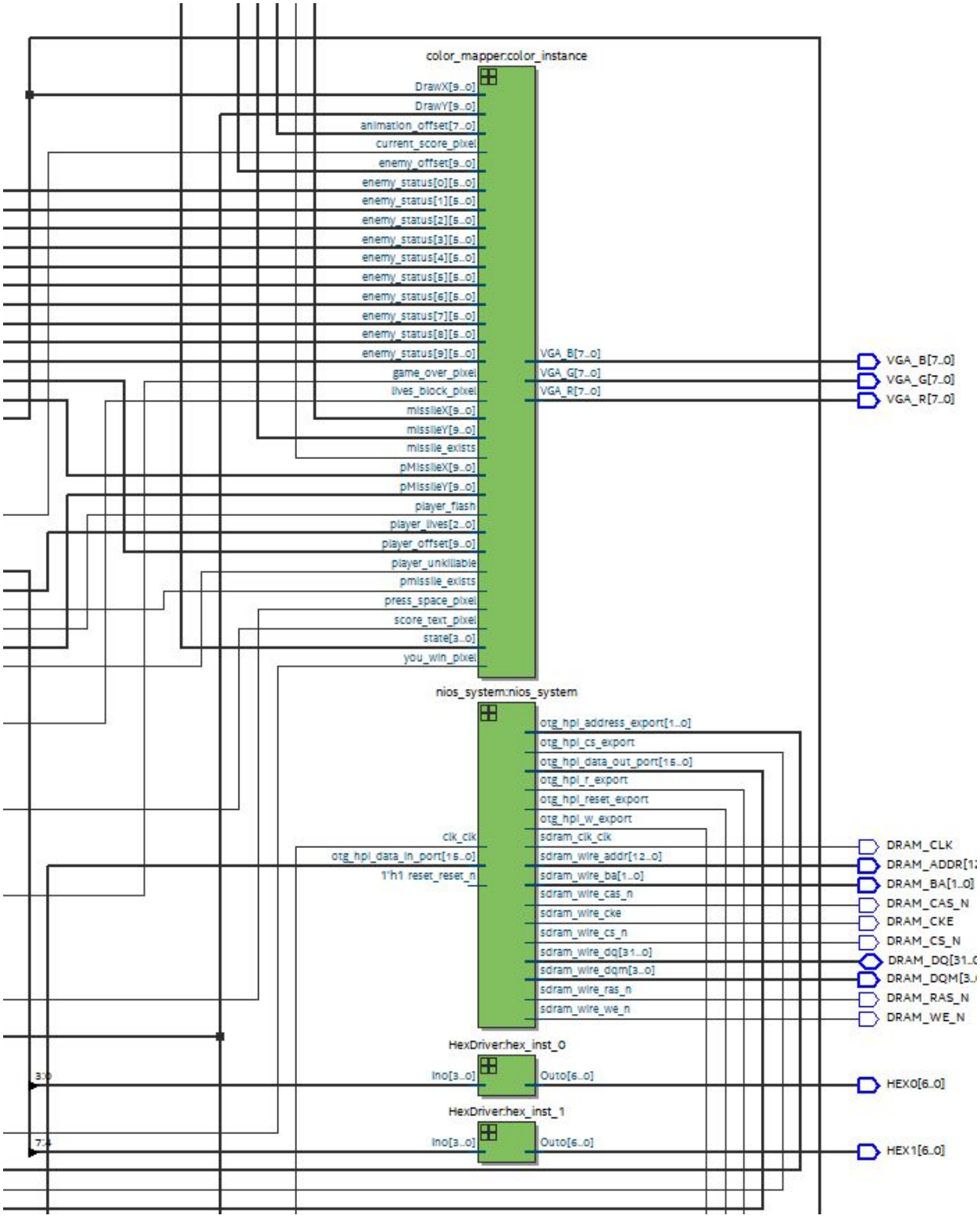
Bottom Left: Zoom in



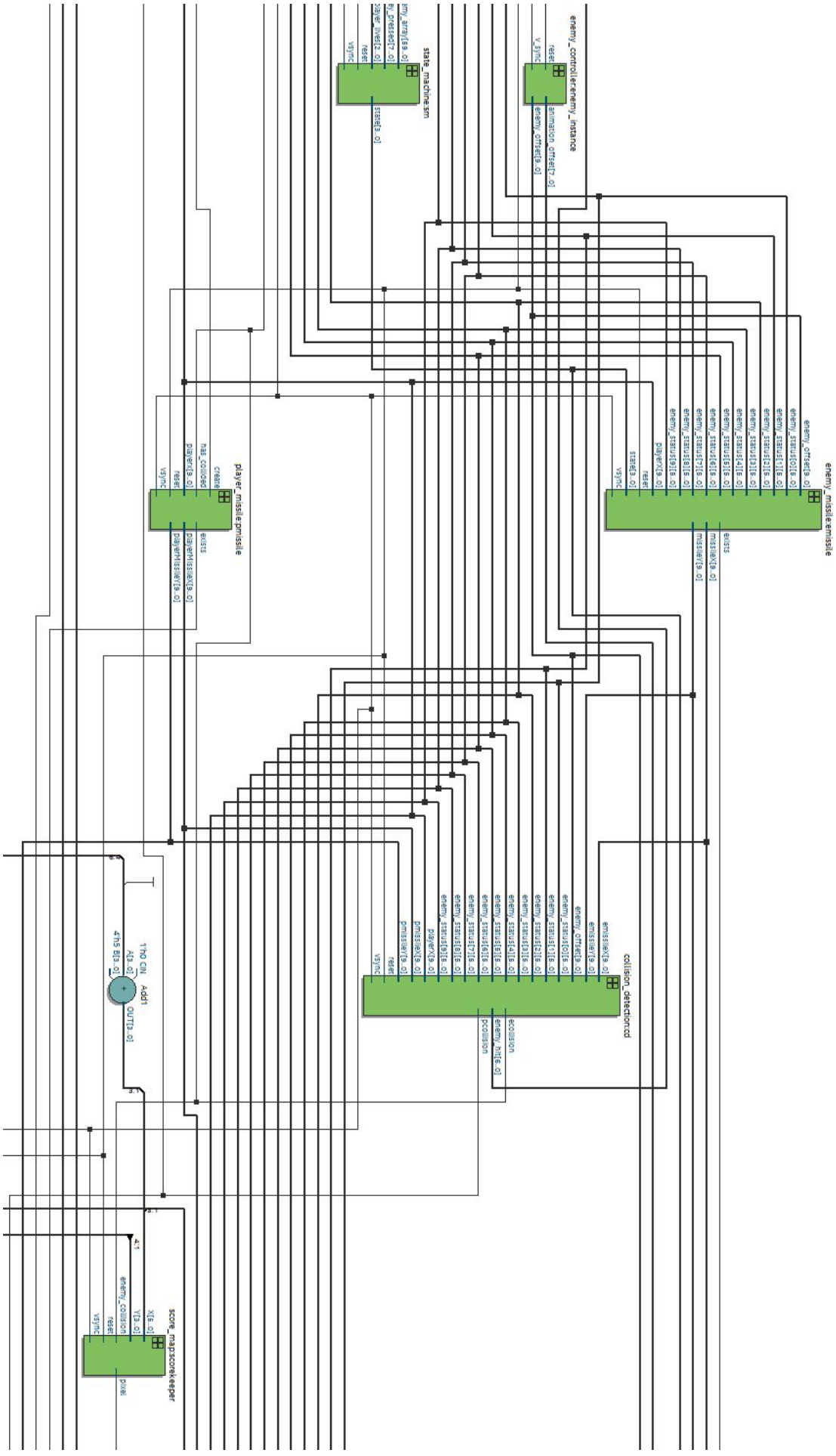
Left Side Zoom In



Right Side Zoom In



Top Middle: Zoom In



# SV Module Descriptions

| Module Name | Description |
|---|---|
| space_invaders | **Inputs:** CLOCK_50, [3:0] KEY, [15:0] OTG_DATA, OTG_INT, [31:0] DRAM_DQ<br>**Outputs:** [6:0] HEX0, [6:0] HEX1, [7:0] VGA_R, [7:0] VGA_G, [7:0] VGA_B, VGA_CLK, VGA_SYNC_N, VGA_BLANK_N, VGA_VS, VGA_HS, [15:0] OTG_DATA, [1:0] OTG_ADDR, OTG_CS_N, OTG_RD_N, OTG_WR_N, OTG_RST_N, [12:0] DRAM_ADDR, [31:0] DRAM_DQ, [1:0] DRAM_BA, [3:0] DRAM_DQM, DRAM_RAS_N, DRAM_CAS_N, DRAM_CKE, DRAM_CKE, DRAM_WE_N, DRAM_CS_N, DRAM_CLK<br>**Description:** This is the top-level module that connects all components of the space invaders game, and performs no other actions. See the block diagram towards the beginning of the report for how all modules are connected.<br>**Purpose:** Again, this is the top-level module that connects all components. |
| color_mapper | **Inputs:** [3:0] state, game_over_pixel, you_win_pixel, press_space_pixel, player_unkillable, [9:0] DrawX, [9:0] DrawY, [7:0] animation_offset, [9:0] enemy_offset, player_flash, missile_exists, [9:0] missileX, [9:0] missileY, pmissile_exists, [9:0] pmissileX, [9:0] pmissileY, [2:0] player_lives, [9:0][5:0] enemy_status, current_score_pixel, score_text_pixel, lives_block_pixel<br>**Outputs:** [7:0] VGA_R, [7:0] VGA_G, [7:0] VGA_B<br>**Description:** Given the state, the current X and Y coordinate being drawn, and the pixel of the enemies, player, text, etc at that point, color_mapper will feed the values of red, green, and blue that need to be drawn to print the appropriate item on the screen. For example, when drawing elements in the top left corner of the screen, color_mapper will know pull data from the “lives_text_mapper” module and display the data given to it.<br>**Purpose:** This module draws all appropriate elements on the screen. It feeds the values or red, green, and blue that need to be printed at every X and Y value on screen to the VGA module. |
| collision_detection | **Inputs:** reset, vsync, [9:0] pmissileX, [9:0] pmisisleY, [9:0]emissileX, [9:0] emissileY, [9:0] playerX, [6:0] enemy_hit, [9:0] enemy_offset, [9:0][5:0] enemy_status<br>**Outputs:** pcollision, ecollision<br>**Description:** This module takes the position of the player missile and enemy missile positions, and then determines if a collision has occurred based off the position of the player, the enemies, and what enemies are still remaining. Upon collision between a missile and the player or enemy occurs, pcollision or ecollision is set high for one clock cycle.<br>**Purpose:** Collisions are detected with this module, which allows for enemies to be killed and the player to lose a life. |
| player_missile | **Inputs:** reset, [9:0] playerx, vsync, create, has_collided<br>**Outputs:** exists, [9:0] playerMissileX, [9:0] playerMissileY<br>**Description:** Whenever create is set high, a player missile is created that travels from the player’s ship upwards towards the enemy array until it collides with an enemy or goes off screen. |

| | |
|---|---|
| | **Purpose:** Because we must have some way to fight the enemy array, this module allows the player to file missiles towards the aliens. |
| enemy_controller | **Inputs:** v_sync, reset<br>**Outputs:** [7:0] animation_offset, [9:0] enemy_offset<br>**Description:** The enemy array floats back and forth about the screen, and also changes animation. This module is responsible for determining the X offset of the array and which animation needs to be showing.<br>**Purpose:** Again, this module is responsible for moving the enemy array back and forth and selecting which enemy animation needs to be shown at any moment. |
| sprite_rom | **Inputs:** [7:0] addr<br>**Outputs:** [7:0] data<br>**Description:** sprite_rom is a sprite table containing data about the enemies and their associated animation. Depending on the addr argument, a slice of an enemy image along the X dimension is returned via data.<br>**Purpose:** sprite_rom is used to store the data of all the enemy sprites. |
| ship_rom | *Same as sprite_rom except for an image of the player's spaceship, see sprite_rom for description* |
| player | **Inputs:** reset, vsync, [7:0] keycode, pcollision<br>**Outputs:** pmissile_create, player_flash, [9:0] x_pos, [2:0] lives, player_unkillable<br>**Description:** The player module is responsible for allowing a human player to control the on-screen spaceship. It controls the movement of the spaceship, firing missiles, and counting remaining lives.<br>**Purpose:** This module allows for a human to control the on-screen spaceship. |
| enemy_missile | **Inputs:** reset, [9:0] playerX, [9:0][5:0] enemy_status, vsync, [9:0] enemy_offset, [3:0] state<br>**Outputs:** exists, [9:0] missileX, [9:0] missileY<br>**Description:** Because the enemies need a way to fire missiles back at the player, the enemy_missile module is responsible for determining when and where to fire at the player. The x and y position are then updated until it collides with the player or goes off screen.<br>**Purpose:** Again, this module handles when and where to fire an enemy's missile, and then updates its position and path. |
| enemy_status | **Inputs:** reset, [6:0] enemy_hit, collision<br>**Outputs:** [9:0][5:0] enemy_status<br>**Description:** enemy_status is a modified two-dimensional register that represents the enemy array. Whenever a collision occurs the bit in the register associated with the hit enemy is set to 0.<br>**Purpose:** This module allows other modules to know what enemies are still alive and allows the player to kill enemies upon a missile collision. |
| number_rom | **Inputs:** [7:0] address<br>**Outputs:** [7:0] data<br>**Description:** This is a sprite table containing images of the numbers 0 through 9. Given an address, it returns a slice of an image of a number.<br>**Purpose:** This table is used to display the current score of the player on screen. |

| | |
|---|---|
| score_map | **Inputs:** vsync, reset, [5:0] X, [3:0] Y, enemy_collision<br>**Outputs:** pixel<br>**Description:** This module keeps track of the current score of the player, and then creates a pixel (given an X and a Y position, a “pixel mapper” specifies what value should be placed at that pixel) mapping using the number_rom sprite table that can be used to print the current score on the screen.<br>**Purpose:** This module is used by color_mapper to print the current score on the screen. |
| lives_text_map | **Inputs:** [5:0] X, [3:0] Y<br>**Outputs:** pixel<br>**Description:** This is a pixel mapper (see definition in score_map) that maps the text “LIVES:” to the screen, to be put before the pictures of the remaining player lives.<br>**Purpose:** This module is used by color_mapper to print the text “LIVES:” to the screen. |
| lives_text_rom | **Inputs:** [7:0] address<br>**Outputs:** [7:0] data<br>**Description:** This is a sprite table containing the characters needed for “LIVES:”. When given an address, this module returns a pixel slice from one of the characters along the X dimension<br>**Purpose:** This is used by lives_text_map to write “LIVES:” to the screen. |
| score_text_rom | *Same as lives_text_rom, except for the text “SCORE:”. See lives_text_rom for information* |
| score_text_map | *Same as lives_text_map, except for the text “SCORE:”. See lives_text_map for information* |
| game_over_text_map | **Inputs:** [5:0] X, [4:0] Y<br>**Outputs:** pixel<br>**Description:** This is another pixel mapper (see definition in score_map) that is responsible for writing the words “GAME OVER” to the screen whenever the player runs out of lives.<br>**Purpose:** This is used by color_mapper to print the text “GAME OVER” to the screen whenever the player runs out of lives. |
| game_over_rom | **Inputs:** [7:0] address<br>**Outputs:** [7:0] data<br>**Description:** This is a sprite table containing text needed to print the words “GAME OVER”. Whenever given an address, it returns a slice along the X dimension of the associated letter.<br>**Purpose:** This is used by game_over_text_map to write the words “GAME OVER” to the screen. |
| press_fire_text_map | *Same as game_over_text_map, except for the text “PRESS SPACE”. See game_over_text_map for more information* |
| press_fire_rom | *Same as game_over_rom, except for the text “PRESS SPACE”. See game_over_text_rom for more information* |
| you_win_text_map | *Same as game_over_text_map, except for the text “YOU WIN”. See game_over_text_map for more information* |

| | |
|---|---|
| you_win_rom | *Same as game_over_rom, except for the text “YOU WIN”. See game_over_text_rom for more information* |
| state_machine | **Inputs:** vsync, [59:0] enemy_array, [2:0] player_lives, [7:0] key_pressed, reset<br>**Outputs:** state<br>**Description:** The state machine moves about 4 states: the initial, beginning state which prints only “PRESS SPACE” to the screen. Whenever the player presses space, we move to the state where the player actually plays the game. Whenever the player loses all his or her lives, we move to the “GAME OVER” state, and whenever the player defeats all the enemies, we move to the “YOU WIN” state.<br>**Purpose:** This is used to control the “flow” of the system. It controls the start of the game and the end of the game. |
| HexDriver | **Inputs:** [3:0] In0<br>**Outputs:** [6:0] Out0<br>**Description:** This hex driver converts the 4 bit input to a signal that can be displayed on an seven segment display. I.e. it converts a 4-bit wide input into an 8-bit wide output that is connected to a seven segment display that shows the value of the input in hex<br>**Purpose:** These are used to show the hex value of the previously typed key. |
| hpi_io_intf | **Inputs:** Clk, Reset, [1:0] from_sw_address, [15:0] from_sw_data_out, from_sw_r, from_sw_w, from_sw_cs, from_sw_reset, [15:0] OTG_DATA<br>**Outputs:** [15:0] from_sw_data_in, [15:0] OTG_DATA, [1:0] OTG_ADDR, OTG_RD_N, OTG_WR_N, OTG_CS_N, OTG_RST_N<br>**Description:** This module connects the NIOS II to the EZ-OTG chip through the FPGA chip.<br>**Purpose:** Again, this module is responsible for routing data through the FPGA to and from the NIOS II and EZ-OTG chip so that they can communicate with each other. |
| VGA_controller | **Inputs:** Clk, Reset, VGA_CLK<br>**Outputs:** VGA_HS, VGA_VS, VGA_BLANK_N, VGA_SYNC_N, [9:0] DrawX, [9:0] DrawY<br>**Description:** This module handles transmitting data and the associated timing of the VGA hardware. It outputs the current pixel being handled, signals such as horizontal and vertical sync. This module requires a very specific clock because of the way in which VGA works.<br>**Purpose:** This module is responsible for handling control of the VGA hardware and sending data to the screen |
| nios_system | **Inputs:** clk_clk, [3:0] keys_wire_export, [15:0] otg_hpi_data_in_port, reset_reset_n, sdram_wire_dq<br>**Outputs:** [7:0] keycode_export, [1:0] otg_hpi_address_export, otg_hpi_cs_export, [15:0] otg_hpi_data_out_port, otg_hpi_r_export, otg_hpi_reset_export, otg_hpi_w_export, sdram_clk_clk, [12:0] sdram_wire_addr, [1:0] sdram_wire_ba, sdram_wire_cas_n, sdram_wire_cke, sdram_wire_cs_n, [31:0] sdram_wire_dq, [3:0] sdram_wire_dqm, sdram_wire_ras_n, sdram_wire_we_n<br>**Description:** This is the module that builds and represents the SOC. It uses the NIOS II as the main processor of the system and is connected to the EZ-OTG USB chip via PIO blocks |

| | |
|---|---|
| | **Purpose:** This SOC is responsible for reading and transmitting data to and from the USB port via a connection to the EZ-OTG chip. |
| vga_clk | **Inputs:** inclk0<br>**Outputs:** c0<br>**Description:** This module takes the 50MHz clock as input and outputs a 25 MHz clock signal which is then used by the VGA system<br>**Purpose:** The VGA system requires a 25 MHz clock to operate, but the FPGA is using a 50 MHz clock. So this module is responsible for stepping down the speed of the clock so it can be used with VGA. |
| otg_hpi_address PIO | **Inputs:** [1:0] address, chipselect, clk, reset_n, write_n, [31:0] writedata<br>**Outputs:** [1:0] out_port, [31:0] readdata<br>**Description:** This PIO block is connected to the EZ-OTG USB chip and selects the desired register that is to be read or written to.<br>**Purpose:** Again, this module is responsible for communicating with the EZ-OTG chip so we can select which register we want to communicate with (here address means the address of one of the four registers). |
| otg_hpi_data (PIO) | **Outputs:** [15:0] otg_hpi_data<br>**Description:** This PIO block is connected to the EZ-OTG USB chip and is responsible for reading and writing data from the USB registers and piping it into the NIOS II processor.<br>**Purpose:** Again, this is responsible for reading data from and writing data to the 4 registers of the USB chip. It is 32 bits wide. |
| otg_hpi_r (PIO) | **Outputs:** otg_hpi_r<br>**Description:** Connects to the read pin of the EZ-OTG USB chip, and controls read operations<br>**Purpose:** This PIO block is responsible for controlling read operations. When it is pulled low, in conjunction with the CS chip, this allows our NIOS II processor to read one of the four registers. |
| otg_hpi_w (PIO) | **Outputs:** otg_hpi_w<br>**Description:** Connects to the write pin of the EZ-OTG USB chip, and controls write operations.<br>**Purpose:** This PIO block is responsible for controlling write operations. When pulled low, in conjunction with the CS chip, this allows our NIOS II processor to write to one of the four registers. |
| otg_hpi_cs (PIO) | **Outputs:** otg_hpi_cs<br>**Description:** connects to the chip select pin of the EZ-OTG USB chip and enables reads/writes to the chip. Allows operations to be performed on the chip.<br>**Purpose:** Whenever a read or write is desired to one of the four registers on the EZ-OTG USB chip, this chip select pin must be pulled low to allow this operation, and then the read or write pins must also be pulled low to select which operation is to be performed. |
| otg_hpi_reset (PIO) | **Outputs:** otg_hpi_reset<br>**Description:** connects to the reset pin of the EZ-OTG USB chip and resets the chip whenever pulled low.<br>**Purpose:** Whenever we wish to reset the USB EZ-OTG chip we can pull this line low and the associated action will be performed. The EZ-OTG USB chip is a relatively complex piece of hardware, and whenever things go haywire a simple fix is to reset the entire system. |

| | |
|---|---|
| Jtag_uart_0 (JTAG UART) | **Outputs:** none<br>**Description:** This connects the DE2 board to the host computer and allows for communication via a serial wire.<br>**Purpose:** In this lab, we use this for debugging and viewing the status of the keyboard and associated hardware. |
| Keys (PIO) | **Outputs:** [3:0] keys_wire<br>**Description:** This connects the 4 buttons on the DE2 board to the NIOS II processor and allows their status to be read<br>**Purpose:** We use one of the keys as a reset button, so whenever the NIOS II detects that this button is pressed, a reset occurs. |

## Design Resources and Statistics Table

| | |
|---|---|
| LUT | 3,664 |
| DSP | None - 0 |
| Memory (BRAM) | 55,296 |
| Flip-Flop | 2,397 |
| Frequency | 137.62 MHz |
| Static Power | 105.19 mW |
| Dynamic Power | 0.93 mW |
| Total Power | 170.11 mW |

## Conclusion

Overall, this final project was both a fun and challenging endeavor. We met all our base features, but sadly due to other obligations with other classes, we had to skip some of the proposed extra features. One major lesson was learned in this project: planning ahead pays off in the future. Ensuring that all sprites and locations about the screen aligned in multiples of two payed off immensely. Due to the necessity of constant multiplication and division when calculating collision, enemy position, etc all we have to do is add zeros to a register or chop of zeroes to perform multiplication and division. We don't have to use the expensive or costly multiplication units on the chip. That lesson aside, this project was a great way to end our trials and traversals in ECE 385 and fully showcase what we have learned in this class.