ECE 385

Fall 2019

Final Project

Fighting Game in SystemVerilog

David Antonowicz, Christopher Hu
Section ABD/Tuesday 08:00 AM-10:50 AM
TAs: Mihir Iyer, Jiaxuan Liu

DISCLAIMER: *Much of our provided code was derived from the code we originally used for Lab 8. This includes the SoC code for the USB OTG, the software code in the SoC, and the VGA clock and controller. Other modules such as the color mapper and players also came from revised modules in Lab 8. Additionally, the module we used for the PS/2 keyboard was provided to us by the ECE 385 website.*

Purpose of Circuit

For the Final Project, we decided to develop a fighting game on the FPGA via SystemVerilog. The fighting game has the fundamental mechanics: two controllable players on a 2D-oriented screen, each player able to move left, move right, and punch. There's additionally a pop-up menu for the beginning and end of a match, and a resettable score system for each respective player on the FPGA.

Description of Fighting Game

The game is played by two players using two keyboards, a USB keyboard and a PS/2 keyboard. A player can perform one of two animations at a time. When a player uses the preset movement keys, a walking animation plays as they traverse the screen. Similarly, when a player presses the attack key, a punch animation plays. If one of the players' fists is within the hitbox of the other player, the other player loses a fraction of his/her health. The moment a player has their health fully depleted, the game ends. The game initializes with both players at opposite ends of the screen while displaying "BOX HIT: PRESS ENTER". Once a player presses the enter key on either keyboard to indicate they are ready to play, the game begins. Once a player is defeated, the game will state the winner, as well as the text "PRESS RESET".

Description of Keyboard Interfaces

- **PS/2 Keyboard**

The PS/2 keyboard is operated by Player 1 in our game design. Its driver is implemented solely within the FPGA hardware without the use of an external chip or addressing protocol. The PS/2 keyboard driver created by Sai Ma and Marie Liu on the ECE385 website was utilized to allow the PS/2 keyboard (player 1) to provide inputs to the game. The driver uses a counter to synchronize the PS/2 clock with the system clock and a series of conditional statements as an edge detector for the PS/2 clock. The PS/2 protocol consists of a series of make and break codes, where a make code corresponds to the initial press of a key, and a break code indicates the release of that key. The codes are passed serially through the data bus, while the clock line carries the keyboard's generated clock signal, indicating when a code is ready to be transmitted to the system. The driver uses two 11-bit shift registers to hold the make and break codes. Every time the PS/2 clock edge is detected, bits are serially shifted from the data line into the registers, after which the code is checked to establish a 'press' signal which indicates whether the code is a make or break code, i.e. the press or release of a key.

- **USB Keyboard**

The USB keyboard is operated by Player 2 and is addressed using the NIOS II to communicate with an external USB chip which temporarily stores data from the keyboard. The NIOS II and EZ-OTG USB chip interact with each other in an indirect manner. Rather than allowing direct access to its RAM,

the USB controller manages its access via HPI registers. There are four registers provided, and of these four, HPI Data and HPI Address are what's used to read and write from the OTG's RAM. The data in the HPI Address register specifies the memory address in the RAM, whereas the HPI Data register specifies the data currently stored in that memory address. Should the NIOS II wish to read from a specific address in the RAM, it would have to first set the address in the HPI Address register, then read from the HPI Data register. Writing to a memory address would be like reading, but instead, the NIOS II directly writes into the HPI Data register after setting the HPI Address register.

The NIOS II and EZ-OTG USB controller still need to have a physical connection to communicate, and hardware-wise they transmit data to each other via a data bus, the bus being driven by a tri-state buffer to avoid any data transmission glitches and errors. The functions used to implement the USB protocol are described below:

- **USBRead**: `USBRead` is the function used to read from an address-specified internal register in the USB controller. There are two main steps in this function. The first step involves utilizing the `IO_write` function to write into the HPI address register the address of the OTG's internal register that the program wants to read from. After setting up the HPI's address register, the second and final step comes into play. This is when `USBRead` calls the `IO_read` function to retrieve the data of the OTG's internal register- whose address is specified by the HPI's recently modified address register- by reading from the HPI's data register and returning the value.

- **USBWrite**: `USBWrite` is the function used to write to an address-specified internal register in the USB controller. There are two main steps in this function, both of which involve using `IO_write`. The first `IO_write` is used to write into the HPI address register the address of the OTG's internal register that the program wants to write to. After the HPI's address register is set to the internal address the function wants to write to, the second `IO_write` is called, which then takes the provided data and writes it into the HPI data register.

- **IO_read**: `IO_read` is one of the functions that provides the direct connection and communication between the NIOS II system and the USB controller, as it's specifically used to read from the USB controller. It takes an HPI address, then reads and returns the data stored in that address-specified HPI register. It's worth noting that before reading from the HPI registers, the function sets the chip select and read enable flags on the OTG to active-low, and after retrieving the data, sets the chip select and read enable flags back to their original states.

- **IO_write**: `IO_write` is one of the functions that provides the direct connection and communication between the NIOS II system and the USB controller, as it's specifically used to write to the USB controller. It takes an HPI address, then writes the provided data into that address-specified HPI register. It's worth noting that before writing to the HPI registers, the function sets the chip select and write enable flags on the OTG to active-low, and after writing the data, sets the chip select and write enable flags back to their original states.

Description of VGA Interface

The VGA itself is managed mostly through hardware and via two main modules: VGA_controller and Color_Mapper. The VGA_controller module uses counters to synchronize and generate horizontal and vertical sync signals. It also keeps track of the current pixel coordinate being counted. The Color_Mapper module defines what color each pixel should output at any given time based on a color index stored at the current address of the on-chip memory.

Description of Sprites and Graphics Implementation

The game graphics were implemented using sprites stored in the on-chip memory. Three partitions of the on-chip memory were instantiated to store the color indices for each character and all of their animation frames, one for player 1, one for player 2, and one for the start and end screens. We programmed a python script to read through each .png sprite image and convert the pixels to color indices, where each pixel represents a 4-bit index. Each pixel's representation is then individually stored on a line of a .mif file that is loaded into the on-chip memory, where each line represents an address of the OCM. The on-chip memory partitions were instantiated using the predefined 1-port ROM IP block in Quartus. We instantiated two 262,144-word partitions for the two player sprites and one 524,288-word partition for the menu sprites.

Once the position and state of the sprites/game is calculated in the player logic and menu modules, the OCM is set to the relevant address and one-by-one retrieves the color indices of the sprite being printed. The color mapper then takes the color index output by the OCM for each pixel and maps it to a preset color to display on the VGA monitor.

Description of Game Mechanics/Logic Implementation

The current state of the game, including player positions, animations, health status, and start/pause status, are handled by three modules: player1.sv, player2.sv, and menu.sv. The player1 and player2 modules process keyboard inputs (PS/2 for player 1 and USB for player 2) to alter the state of the game and calculate outputs to the other modules, including the OCM address, a flag to indicate whether we are printing a sprite or just the game background, a flag to indicate if the player has performed a punch action, the horizontal position of the player, and the status of the player's health. The 'D' and 'S' keys are used to move left and right, respectively, for either player, while the 'F' key is used to attack. When one of the keys is pressed, the values for all the intermediate logic variables are set for the next clock cycle, including the player's direction of motion, animation counters, and various flags to indicate the state of the player.

There were several design choices that had to be made in the development process for the game logic. Some of the most important issues that had to be addressed included player animations, key-spamming prevention, sprite borders overlapping, hitbox creation, a player health system, and the start and end screens.

The animations were implemented using a series of counters, one for the movement animation, one for the hurt animation, and one for the punching animation. A keypress or an action from the other player initiates the relevant counter, which returns to zero following the playback of the animation. We used the value of the counter at a given time to count the number of frames and specified which sprite to print based on the current frame.

Another design component essential to the game is the proper printing of the sprites. Before fixing the sprite logic, the background of the player 1 sprite was being printed over player 2 if the two came close enough together. To prevent this, we instantiated two individual OCM modules, each with its own addressing variable. The first module prints pixels exclusively for the player 1 sprite, while the second handles player 2. By this method, we were able to program the color mapper to recognize the background of either sprite, and choose to print the opposites sprite's pixel (stored in the other OCM) in the case of their borders overlapping.

The hitboxes and health system were implemented by traditional means of conditional statements that check the borders and positions of both players. If a player attacks while being within a certain number of pixels of the other player, the opposite player enters the hurt animation and has a fraction of their health deducted. The logic for printing the health bars of either player is handled in the color mapper module, where the health value of both players is used to calculate where the mapper should start printing red vs. green for the health bars. We also sought to prevent key spamming, particularly that which results from holding down the attack button near the opponent and trapping them. In order to combat this practice, we made it so that the 'F' key must be fully released before begin pressed again to carry out a punch. This was implemented using an attack flag that resets upon the release of the 'F' key. The flag prevents the attack animation from being initiated.

The final feature we designed was the start and end screen system. The menu sprites are all stored in the third partition of the on-chip memory, the menu OCM. Upon game start, the logic variable used to keep track of whether the game is in a paused state is initialized to active. Then, when either player presses the 'ENTER' key, the paused variable is set low and the game begins. The variable is also set when one of the players is finally defeated. Based on the value of this variable, the color mapper knows whether to use pixel values from the menu OCM, rather than the player 1 or player 2 OCM. We have also implemented logic to ensure that when in a paused state, neither player may respond to inputs from the keyboard.

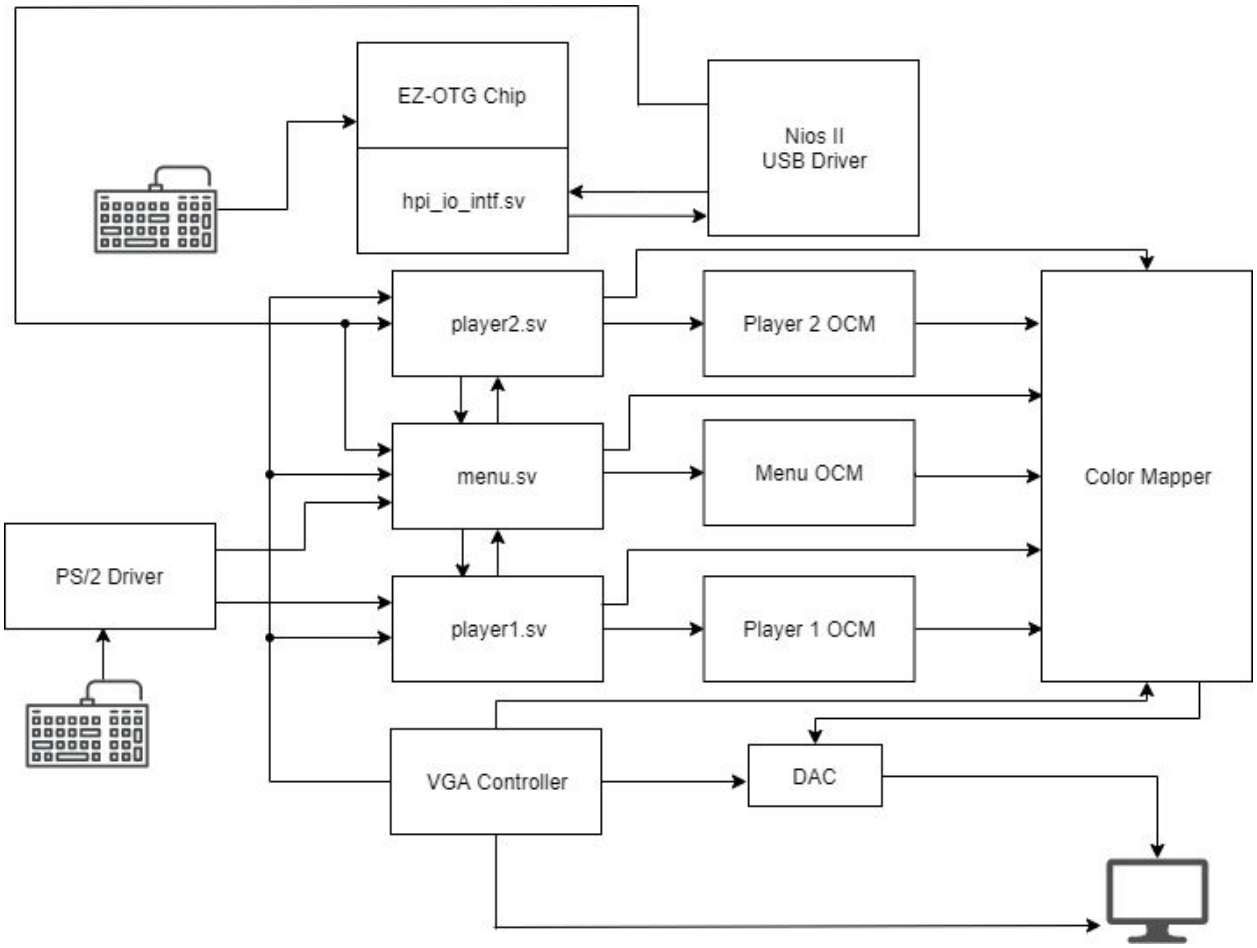

Fig. 1. High Level Block Diagram

Fighting Game Design Modules

- top_level (top_level.sv)
  - *Inputs*: `CLOCK_50,`
    `[3:0] KEY,`
    `OTG_INT,`
    `PS2_CLK, PS2_DAT,`
  - *Outputs*: `[6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, HEX6, HEX7,`
    `[7:0] VGA_R, VGA_G, VGA_B,`
    `VGA_CLK, VGA_SYNC_N, VGA_BLANK_N, VGA_VS, VGA_HS,`
    `[1:0] OTG_ADDR,`
    `OTG_CS_N, OTG_RD_N, OTG_WR_N, OTG_RST_N,`
    `[12:0] DRAM_ADDR,`
    `[1:0] DRAM_BA,`
    `[3:0] DRAM_DQM,`
    `DRAM_RAS_N, DRAM_CAS_N, DRAM_CKE, DRAM_WE_N, DRAM_CS_N, DRAM_CLK`
  - *Inouts*: `[15:0] OTG_DATA,`
    `[31:0] DRAM_DQ`
  - *Description*: This is the unit that takes the NIOS II processor, USB controller, PS/2 Driver, VGA monitor, and SDRAM and integrates them together to create our functioning SoC. `CLOCK_50` is used as the main clock signal for the system. `KEY[0]` is used as the reset button. `HEX0` through `HEX7` are used to display the point system for the game, with `HEX0` and `HEX1` displaying player 2's score and `HEX4` and `HEX5` displaying player 1's score. The other variables are outputs to the peripherals. `VGA_R`, `VGA_G`, `VGA_B`, `VGA_CLK`, `VGA_SYNC_N`, `VGA_BLANK_N`, `VGA_VS`, and `VGA_HS` are signals that are assigned to the connected VGA monitor. `OTG_INT`, `OTG_ADDR`, `OTG_CS_N`, `OTG_RD_N`, `OTG_WR_N`, and `OTG_RST_N` are used as signals to the EZ-OTG USB controller, `OTG_DATA` being used as a data bus for the OTG. `PS2_CLK` and `PS2_DAT` are the clock and data inputs, respectively, from the PS/2 port on the DE2 which feed into the PS/2 keyboard driver. Finally is the SDRAM, which uses `DRAM_ADDR`, `DRAM_BA`, `DRAM_DQM`, `DRAM_RAS_N`, `DRAM_CAS_N`, `DRAM_CKE`, `DRAM_WE_N`, `DRAM_CS_N`, and `DRAM_CLK` as signals, `DRAM_DQ` being used in particular as the data bus for the SDRAM.
  - *Function*: The top-level module for the SoC machine. This module takes the assigned pins that we need for our machine to work and maps each of them to their respective peripherals, namely the EZ-OTG USB controller, the VGA monitor, and the SDRAM. In the center of everything is the NIOS II processor, which is able to read from and write to the USB controller, provide data to the VGA controller modules, and store programs in the SDRAM.

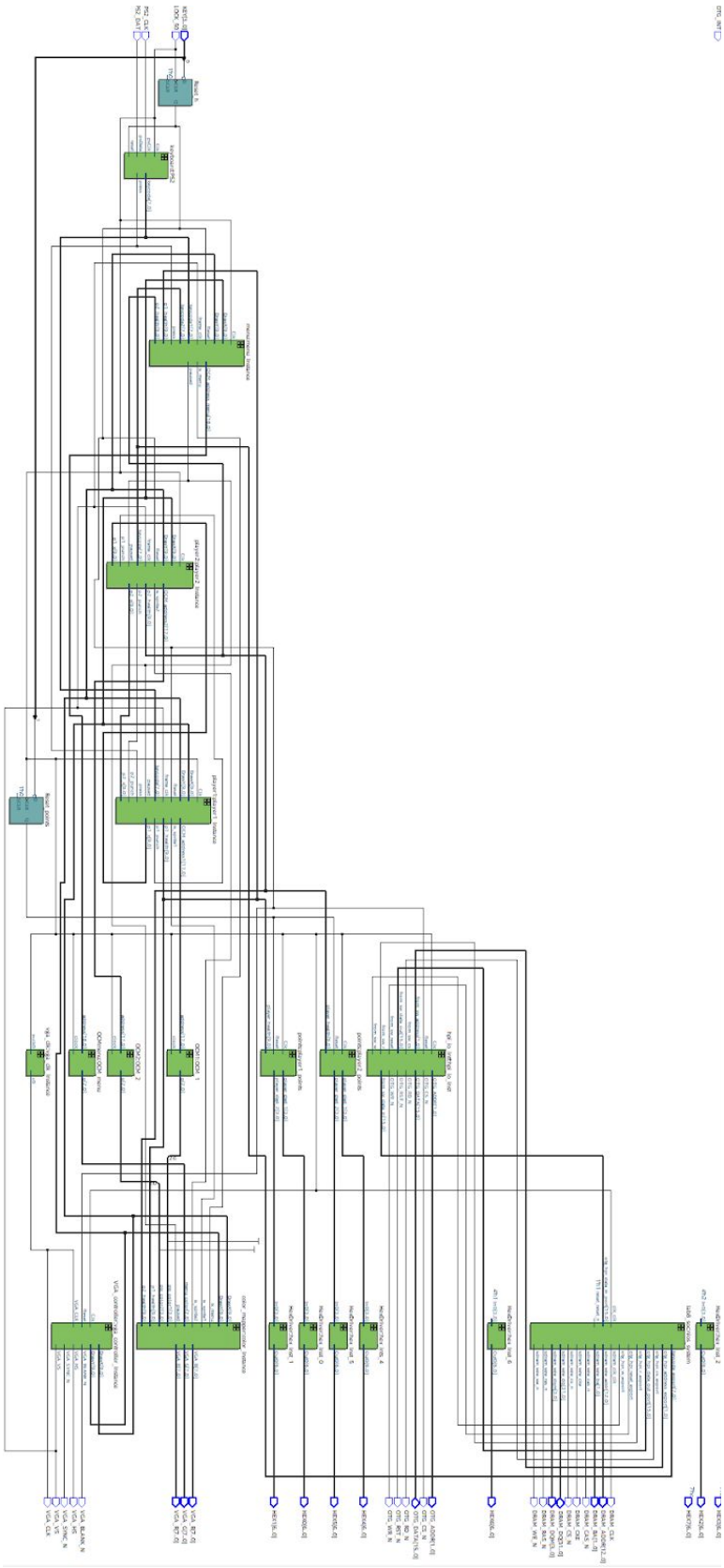

Fig. 2. Top Level Block Diagram

- lab8_soc (lab8_soc.v)
    - *Inputs*: `clk_clk,`
      `[15:0] otg_hpi_data_in_port,`
      `[1:0] push_buttons_export,`
      `reset_reset_n`
    - *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,`
      `[3:0] sdram_wire_dqm,`
      `sdram_wire_ras_n,`
      `sdram_wire_we_n`
    - *Inouts*: `[31:0] sdram_wire_dq`
    - *Description*: This is the unit that establishes the wiring of the NIOS II system's IP blocks and their relations with each other and the DE2 board. The module declares all inputs and outputs as wires and connects them to their respective IP blocks that were generated in the SoC platform designer. `clk_clk` is assigned to the Clock Source block. `reset_reset_n` is used for all blocks that require a reset input. `keycode_export` is assigned to the keycode PIO block. `otg_hpi_address_export` is assigned to the otg_hpi_address PIO block. `otg_hpi_data_in_port` and `otg_hpi_data_out_port` are assigned to the otg_hpi_data PIO block. `otg_hpi_r_export` is assigned to the otg_hpi_r PIO block. `otg_hpi_w_export` is assigned to the otg_hpi_w PIO block. `otg_hpi_cs_export` is assigned to the otg_hpi_cs PIO block. `otg_hpi_reset_export` is assigned to the otg_hpi_reset PIO block. The other variables, `sdram_clk_clk`, `sdram_wire_addr`, `sdram_wire_ba`, `sdram_wire_cas_n`, `sdram_wire_cke`, `sdram_wire_cs_n`, `sdram_wire_dqm`, `sdram_wire_ras_n`, `sdram_wire_we_n`, and `sdram_wire_dq` are all used as a means of wired data transfer from the NIOS II system's established SDRAM PLL and controller blocks to the board's external SDRAM.
    - *Function*: The unit that connects the NIOS II system with its respective components and establishes the interface the SoC will run on. This module turns its received inputs and designated outputs into wires that transfer data between the DE2 board as well as the IP blocks, which were declared and generated in the SoC platform designer.

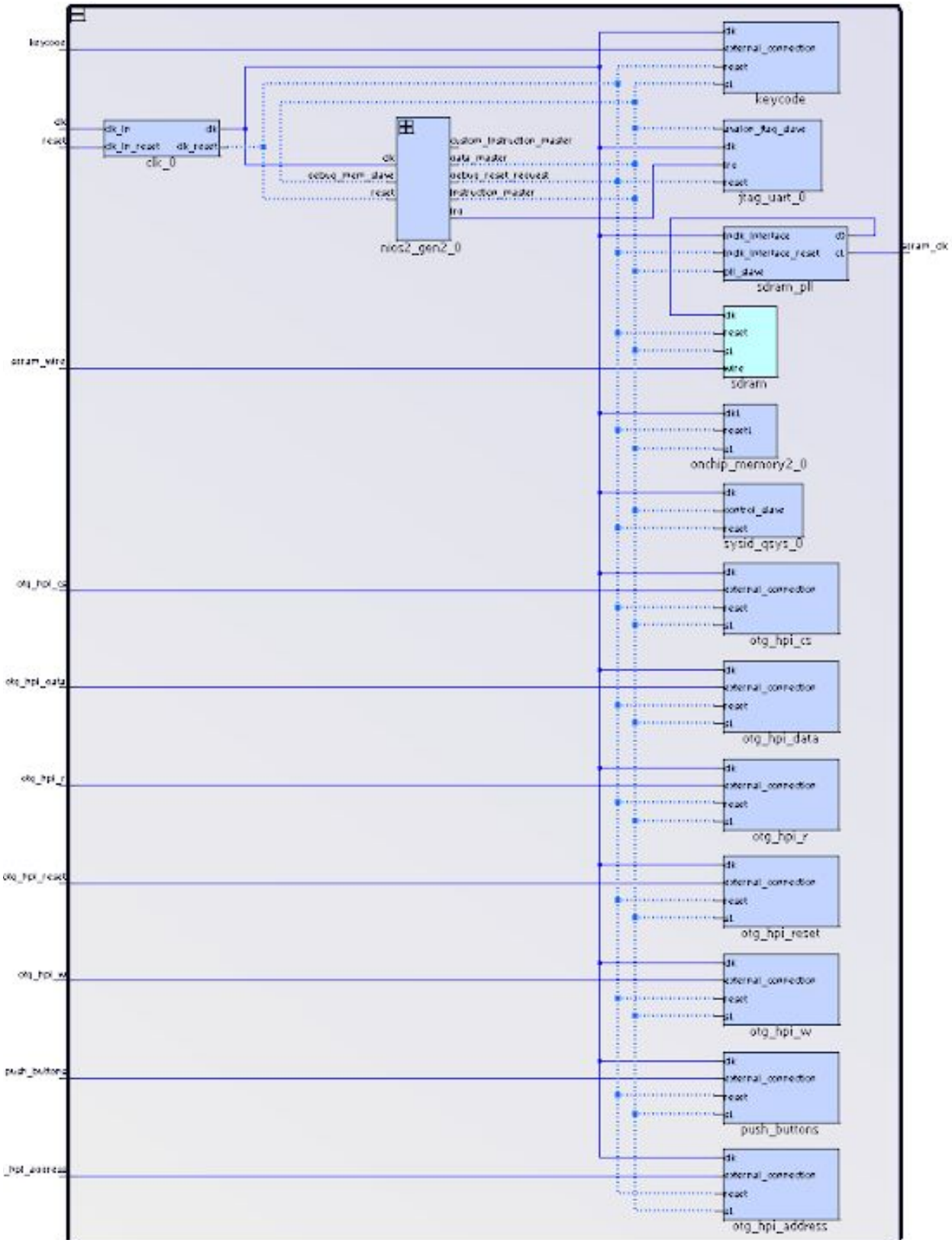

Fig. 3. lab8_soc Block Diagram

- hpi_io_intf (hpi_io_intf.sv)
    - *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`
    - *Outputs*: `[15:0] from_sw_data_in,`
      `[1:0] OTG_ADDR,`
      `OTG_RD_N, OTG_WR_N, OTG_CS_N, OTG_RST_N`
    - *Inouts*: `[15:0] OTG_DATA`
    - *Description*: This is the sequential interface between the NIOS II and USB controller, managing how the NIOS II controls the behaviour of the USB controller as well as transmitting data between the two components. `Clk` is used as the clock signal and causes the USB controller to update itself whenever a positive edge occurs. When `Reset` is active, the USB controller is set back to its default state. The other inputs correspond to the NIOS II wires, and most of the outputs correspond to the USB controller pins, with an exception. `from_sw_address` is assigned to `OTG_ADDR`, `from_sw_r` is assigned to `OTG_RD_N`, `from_sw_w` is assigned to `OTG_WR_N`, `from_sw_cs` is assigned to `OTG_CS_N`, and `from_sw_reset` is assigned to `OTG_RST_N`. The special case is the means of data transmission, which uses `from_sw_data_out`, `from_sw_data_in`, and `OTG_DATA`. `OTG_DATA` is used as the data bus between the NIOS II and OTG, and is controlled by a tristate buffer within the module. Depending on the operation, the `OTG_DATA` bus will either output its data into `from_sw_data_in`, or take `from_sw_data_out` as an input.
    - *Function*: The module that houses the physical interface between the NIOS II and EZ-OTG chip. It determines how the OTG behaves through the NIOS II, namely which NIOS II wires are assigned to which OTG pins, and also decides how the OTG should be reset. Additionally, the module has a tristate buffer in order to prevent data leaks and properly manage data transmissions between the USB controller and NIOS II system.

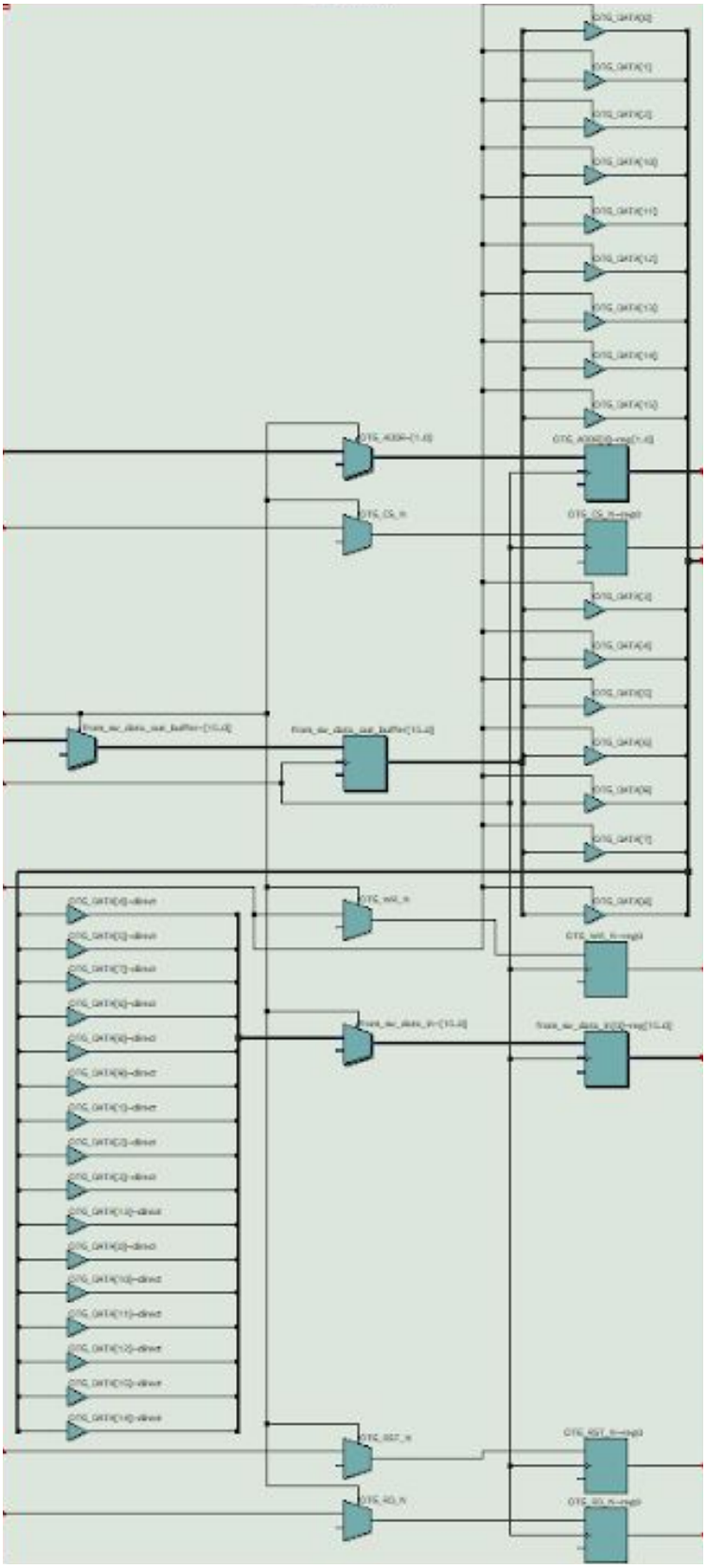

Fig. 4. hpi_io_intf Block Diagram

- vga_clk (vga_clk.v)
    - *Inputs*: `inclk0`
    - *Outputs*: `c0`
    - *Description*: This is the synchronizer for the VGA components. It takes in a clock signal `inclk0` and modifies it, outputting the new clock signal through `c0`, which will be used for the VGA controller.
    - *Function*: The clock synchronizer for the VGA monitor. For the VGA monitor to horizontally and vertically synchronize all of its pixels, it needs to have a timing signal for both its horizontal and vertical coordinates. Along with the fact that it is not part of the FPGA and therefore will have a clock skew, it can't use the default clock signal, and thus uses this module to synchronize itself with the rest of the system.

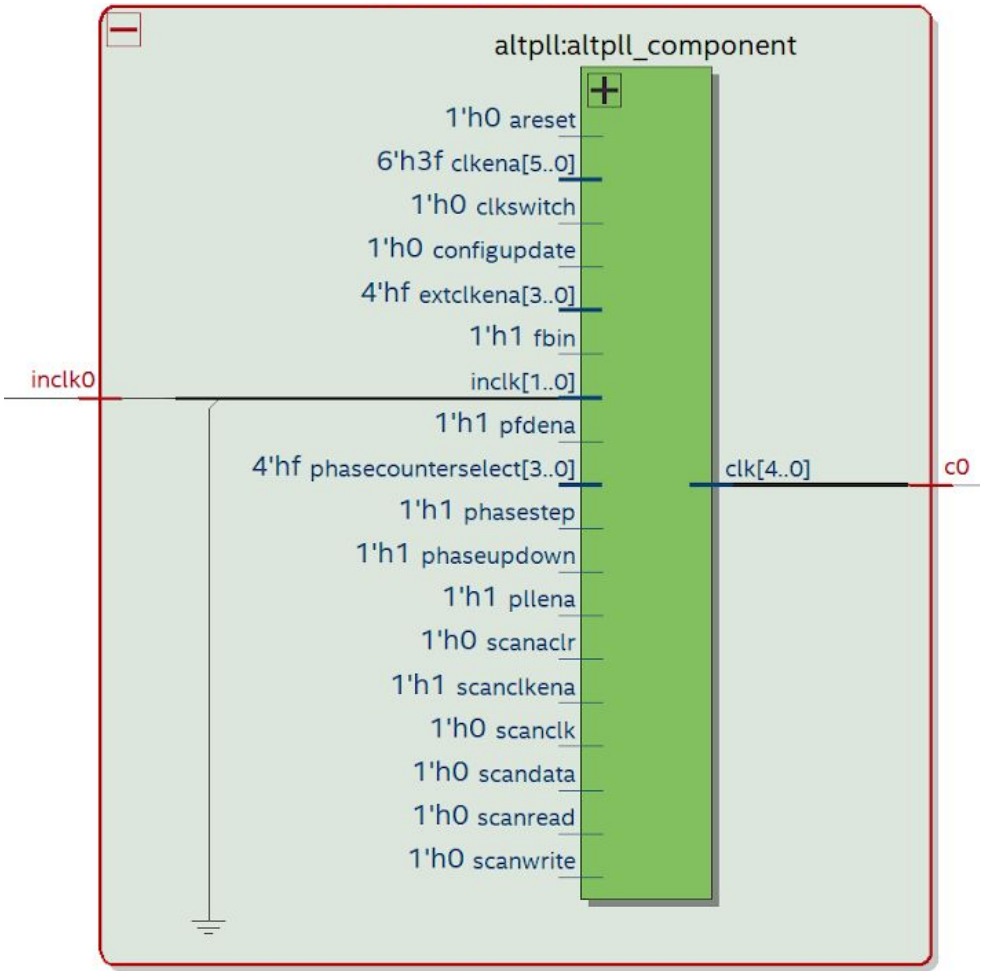

Fig. 5. vga_clk Block Diagram

- VGA_controller (VGA_controller.sv)
    - *Inputs*: `Clk, Reset, VGA_CLK`
    - *Outputs*: `VGA_HS, VGA_VS, VGA_BLANK_N, VGA_SYNC_N, [9:0] DrawX, DrawY`
    - *Description*: This is the sequential module that manages the outputted pixels on the VGA monitor. It takes in the clock signal `VGA_CLK`, which causes the module to update every time a positive edge occurs. If `Reset` is active, it sets all components of the VGA back to their default values. `VGA_HS` and `VGA_VS` are respectively the horizontal and vertical sync pulse variables used to properly and consistently iterate through the pixels horizontally and vertically. `VGA_BLANK_N` is the blanking interval signal that tells the VGA monitor whether or not display the current pixels. `VGA_SYNC_N` is the overall sync enable signal that's always active low in order for the VGA components to be able to synchronize. `DrawX` and `DrawY` correspond to the coordinates of the current pixel being modified.
    - *Function*: The controller for the VGA monitor. The VGA monitor needs to be able to properly manage and update all of its available pixels in a manner that's synchronous to the overall system, which is managed by this module.

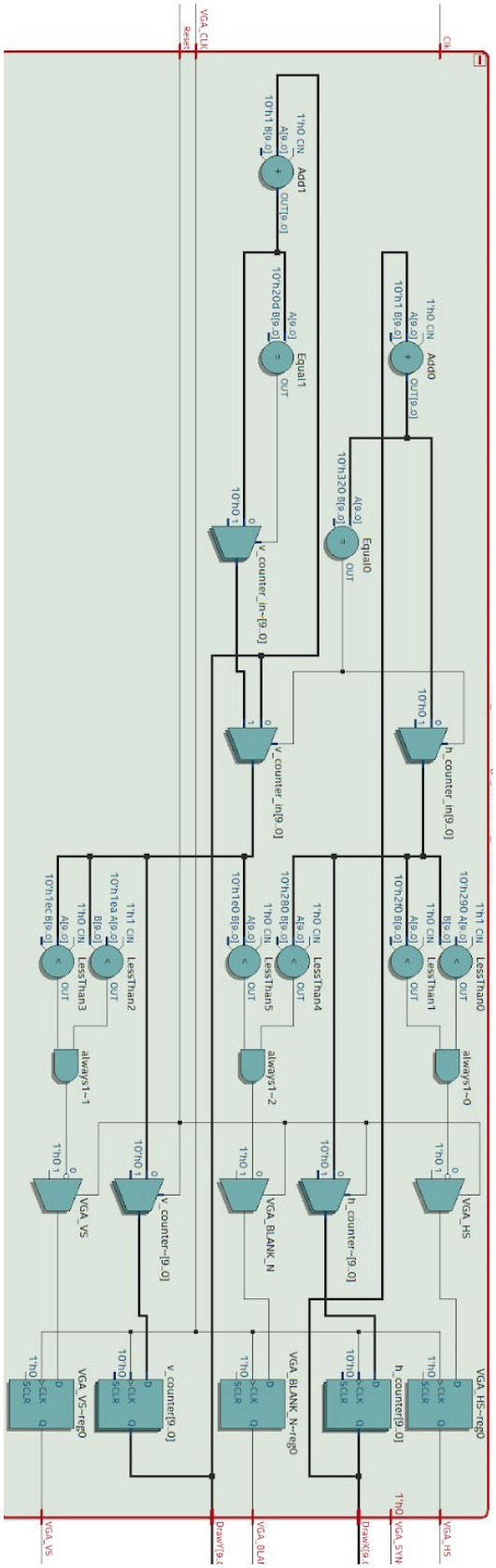

Fig. 6. VGA_controller Block Diagram

- player1 (player1.sv)
    - *Inputs*: `Clk, Reset, frame_clk, press, p2_punch, paused,`
      `[7:0] keycode,`
      `[9:0] p2_x,`
      `[9:0] DrawX, DrawY`
    - *Outputs*: `is_sprite1, p1_punch,`
      `[9:0] p1_x, p1_health,`
      `[17:0] OCM_address1`
    - *Description*: This is the sequential and combinational module that controls the behaviour of player 1 on the VGA monitor. `Clk` is the main signal that synchronizes the registers associated with player1's position, motion, health status, and animations upon a positive edge. If `Reset` is active during a positive edge trigger, the module sets all registers back in their default state. `frame_clk` is the clock signal that's used to determine when to update each of the parameters corresponding to player 1. `keycode` and `press` are used to decide when and how the player's current motion/animation should change based on a keypress or release. `p2_punch` is a flag that indicates whether the opposing player has activated an attack, `paused` indicates whether the game is in a paused state, and `p2_x` holds the x-coordinate of the opposing player. `DrawX` and `DrawY` specify the coordinates of the pixel that the VGA monitor is currently updating. If the coordinates fall within the borders of the current position of the sprite, `is_sprite1` is set to active to notify the VGA components that the color for that pixel should originate from OCM1. `p1_punch` is a flag that feeds to the player 2 module to inform player 2's logic that player 1 has attacked. `p1_x` is the current x-coordinate of player 1's position and `p1_health` is the health value of player 1. `OCM_address1` is the address that notifies OCM1 of the location of the color index for the pixel that must be printed.
    - *Function*: The module that manages the game state for player 1 and all of their actions and parameters. It uses keycodes from the PS/2 keyboard to alter the behavior of player 1, and utilizes a few parameters from player 2 to check hitboxes and sprite borders. It also exports several signals to player 2's module.

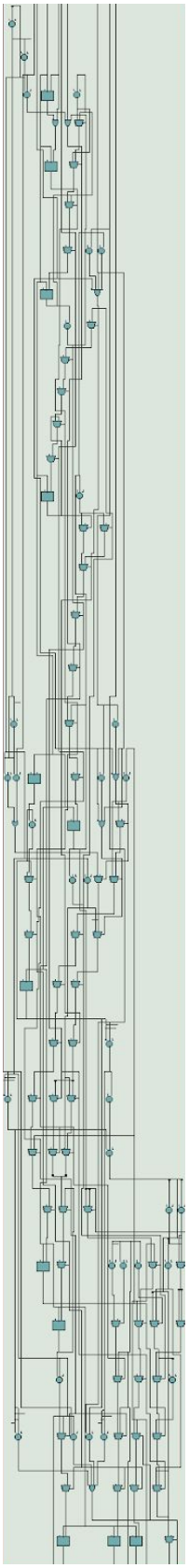

Fig. 7. player1 Block Diagram

- player2 (player2.sv)
    - *Inputs*: `Clk, Reset, frame_clk, p1_punch, paused,`
      `[7:0] keycode,`
      `[9:0] p1_x,`
      `[9:0] DrawX, DrawY`
    - *Outputs*: `is_sprite2, p2_punch,`
      `[9:0] p2_x, p2_health,`
      `[17:0] OCM_address2`
    - *Description*: This is the sequential and combinational module that controls the behaviour of player 2 on the VGA monitor. `Clk` is the main signal that synchronizes the registers associated with player 2's position, motion, health status, and animations upon a positive edge. If `Reset` is active during a positive edge trigger, the module sets all registers back in their default state. `frame_clk` is the clock signal that's used to determine when to update each of the parameters corresponding to player 2. `keycode` is used to decide when and how the player's current motion/animation should change based on a keypress or release. `p1_punch` is a flag that indicates whether the opposing player has activated an attack, `paused` indicates whether the game is in a paused state, and `p1_x` holds the x-coordinate of the opposing player. `DrawX` and `DrawY` specify the coordinates of the pixel that the VGA monitor is currently updating. If the coordinates fall within the borders of the current position of the sprite, `is_sprite2` is set to active to notify the VGA components that the color for that pixel should originate from OCM2. `p2_punch` is a flag that feeds to the player 1 module to inform player 1's logic that player 2 has attacked. `p2_x` is the current x-coordinate of player 2's position and `p2_health` is the health value of player 2. `OCM_address2` is the address that notifies OCM2 of the location of the color index for the pixel that must be printed.
    - *Function*: The module that manages the game state for player 2 and all of their actions and parameters. It uses keycodes from the USB keyboard to alter the behavior of player 2, and utilizes a few parameters from player 1 to check hitboxes and sprite borders. It also exports several signals to player 1's module.

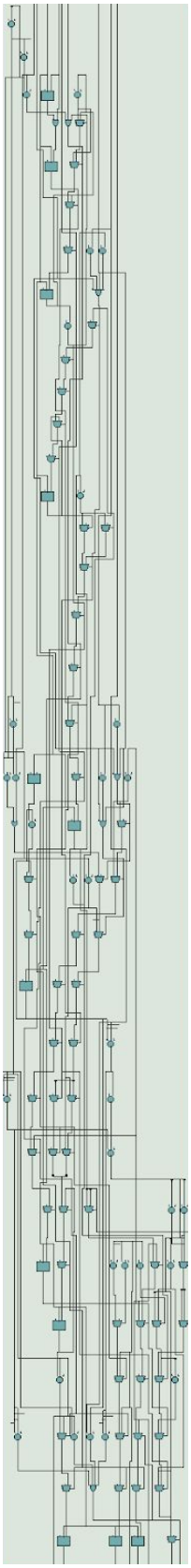

Fig. 8. player2 Block Diagram

- menu (menu.sv)
    - *Inputs*: `Clk, Reset, frame_clk, press,`
      `[7:0] keycode1, keycode2`
      `[9:0] p1_health, p2_health,`
      `[9:0] DrawX, DrawY`
    - *Outputs*: `paused, is_menu,`
      `[18:0] OCM_address_menu`
    - *Description*: This is the sequential and combinational module that controls the logic for when to print the start and end screens, as well as establishes the paused state for the game. `Clk` is the main signal that synchronizes the updating of the start menu and `paused` signals. If `Reset` is active during a positive edge trigger, the module returns the game to the start menu and places it in a paused state. `frame_clk` is the clock signal that's used to determine when to recalculate and update the `paused` and menu start signals. `keycode1` and `keycode2` are keycodes from the PS/2 and USB keyboards, respectively, that determine when to exit the start menu or restart the game. `p1_health` and `p2_health` are the health values from player 1 and 2 that trigger the menu module to print the end screen when either value falls to zero. `DrawX` and `DrawY` specify the coordinates of the pixel that the VGA monitor is currently updating. If the coordinates fall within the predefined borders of the menu, `is_menu` is set active to notify the VGA components that the color for that pixel should originate from the menu OCM. `paused` indicates whether the game has entered a paused state, freezing the characters and displaying a menu. `OCM_address_menu` is the address that notifies the menu OCM of the location of the color index for the pixel that must be printed.
    - *Function*: The module that manages the paused state and start/end screens for the game. It uses keycodes from the PS/2 and USB keyboards to exit the start menu and initiate gameplay. It pauses the game and displays the end screen if either player's health reaches zero.

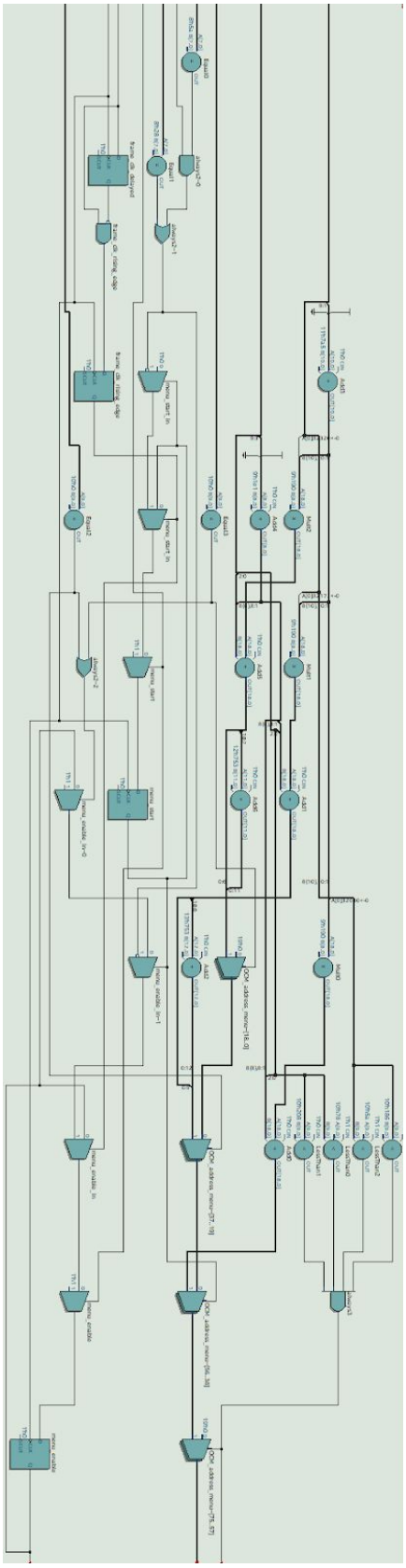

Fig. 9. menu Block Diagram

- points (points.sv)
    - *Inputs*: `Clk, Reset, [9:0] player_health`
    - *Outputs*: `[3:0] player_digit_1, player_digit_2`
    - *Description*: This is the sequential and combinational module that implements the points system for the game. It is instantiated twice in the top level, once for player 1 and once for player 2. `Clk` is the clock input that allows each individual digit of both players' scores to be updated synchronously. The `Reset` input is fed by Key 2 on the FPGA and resets the score exclusively from the game state. `player_health` is the health of the enemy player, opposite of the player that the points module controls. `player_digit_1` and `player_digit_2` are the least significant and most significant digits, respectively, of the player's current score.
    - *Function*: The module that manages the score for a player. If the health bar of the opposite player reaches 0, the score for the main player is incremented. The module also ensures that the score is incremented only once per match.

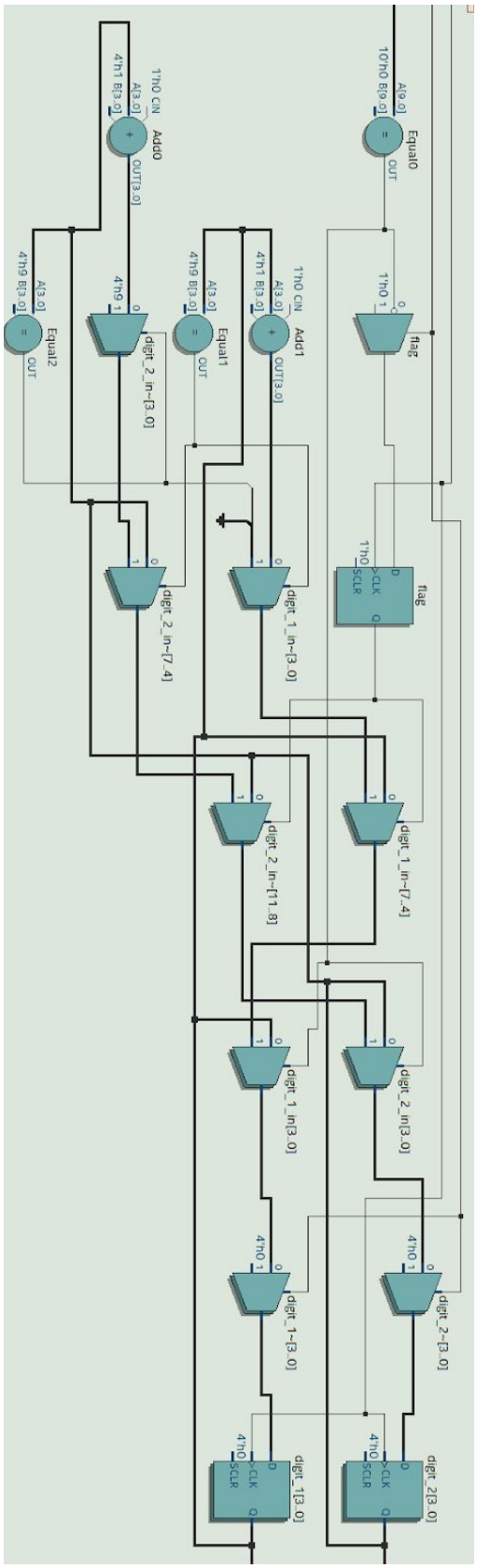

Fig. 10. points Block Diagram

- OCM (OCM1.v, OCM2.v, and OCMmenu.v)
    - *Inputs*: `clock,`
      `[17:0] address` (OCM1.v and OCM2.v)
      or `[18:0] address` (OCMmenu.v)
    - *Outputs*: `[2:0] q`
    - *Description*: This is the module that generates a 1-port ROM partition in the on-chip memory for sprite storage. The memory is intialized using a .mif file, where each line (pixel) of the file corresponds to an address in the on-chip memory. `clock` synchronizes the on-chip memory with the system clock. `address` specifies which address we are reading from. `q` is the output, or value stored at that address.
    - *Function*: The module that creates a partition within the OCM for us to store sprites/pixel color indices using .mif files.

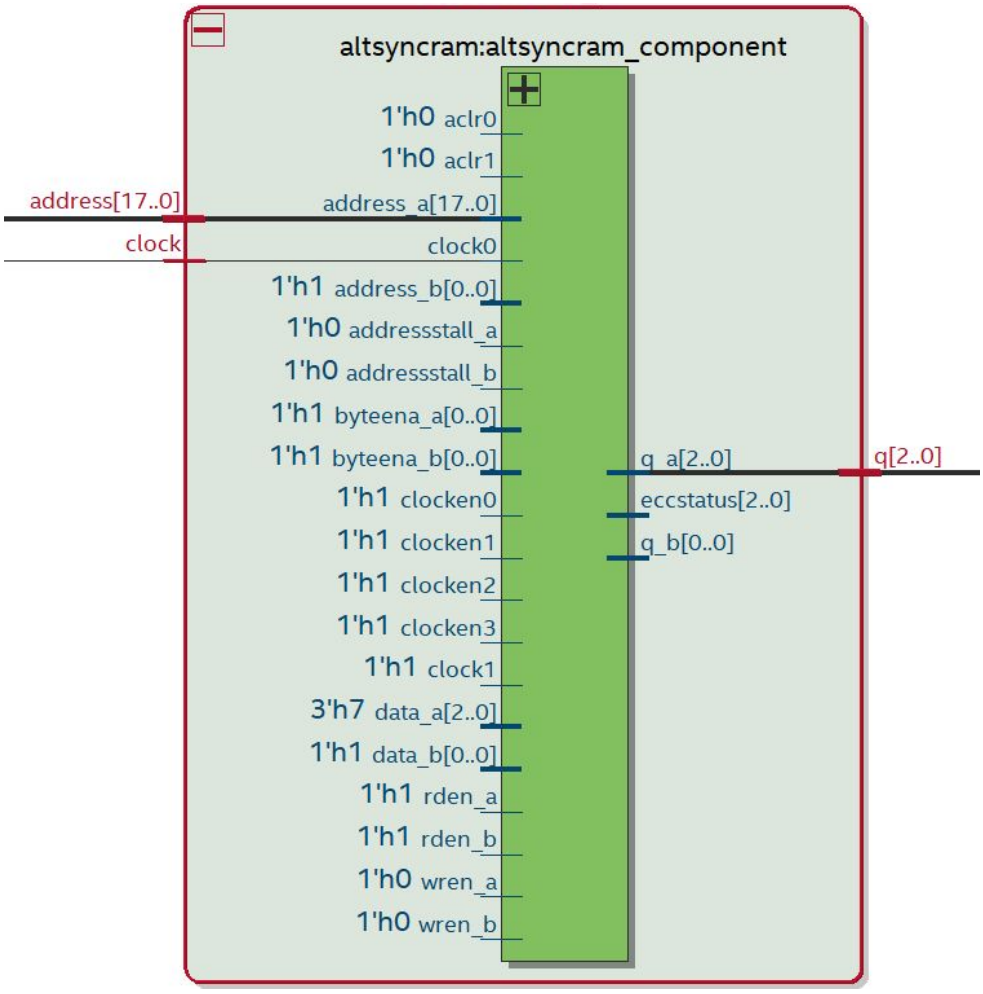

Fig. 11. OCM Block Diagram

- keyboard (keyboard.sv)
    - *Inputs*: `Clk, psClk, psData, reset`
    - *Outputs*: `[7:0] keycode,`
      `press`
    - *Description*: This is the PS/2 keyboard driver written by Marie Liu and Sai Ma. It utilizes a series of counters to synchronize the PS/2 clock and the system clock, flip flops to detect the edge of the PS/2 clock, and two 11-bit shift registers that are constantly updated with the latest make or break codes from the keyboard. `Clk` is the main system clock. `psClk` and `psData` are the clock and data inputs, respectively, from the PS/2 port on the DE2. `reset` is used to reset all the shift registers and flip-flops, deleting the currently stored keycode. The `keycode` and `press` outputs indicate the code of the last key that was interacted with, and whether that key was pressed or released based on the form of the code (make code vs. break code).
    - *Function*: The module that allows the FPGA to interface with the PS/2 port on the DE2 and receive keycodes.

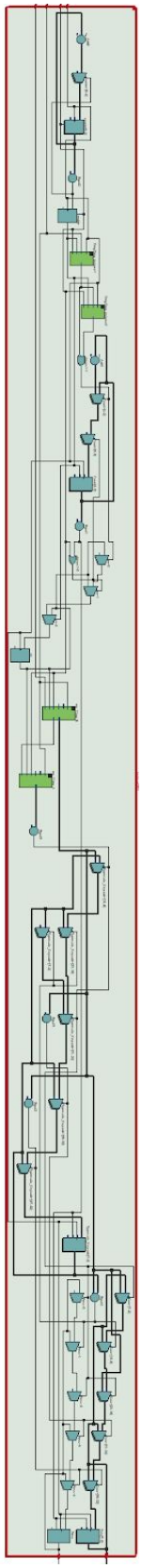

Fig. 12. keyboard Block Diagram

- Dreg (Dreg.sv)
    - *Inputs*: `Clk, Load, Reset, D`
    - *Outputs*: `Q`
    - *Description*: This is a combinational module that behaves as a one-bit register, or flip-flop. At the positive edge of `Clk`, given that Load is active, the output Q will be updated with the input D. If Reset is active, the output Q is set low. Otherwise, the output Q retains its value. This module was part of the keyboard driver written by Marie Liu and Sai Ma.
    - *Function*: The module that is used to create an edge detector for the PS/2 clock that specifies when it is time to update the shift registers with the current keycode of the PS/2 keyboard.

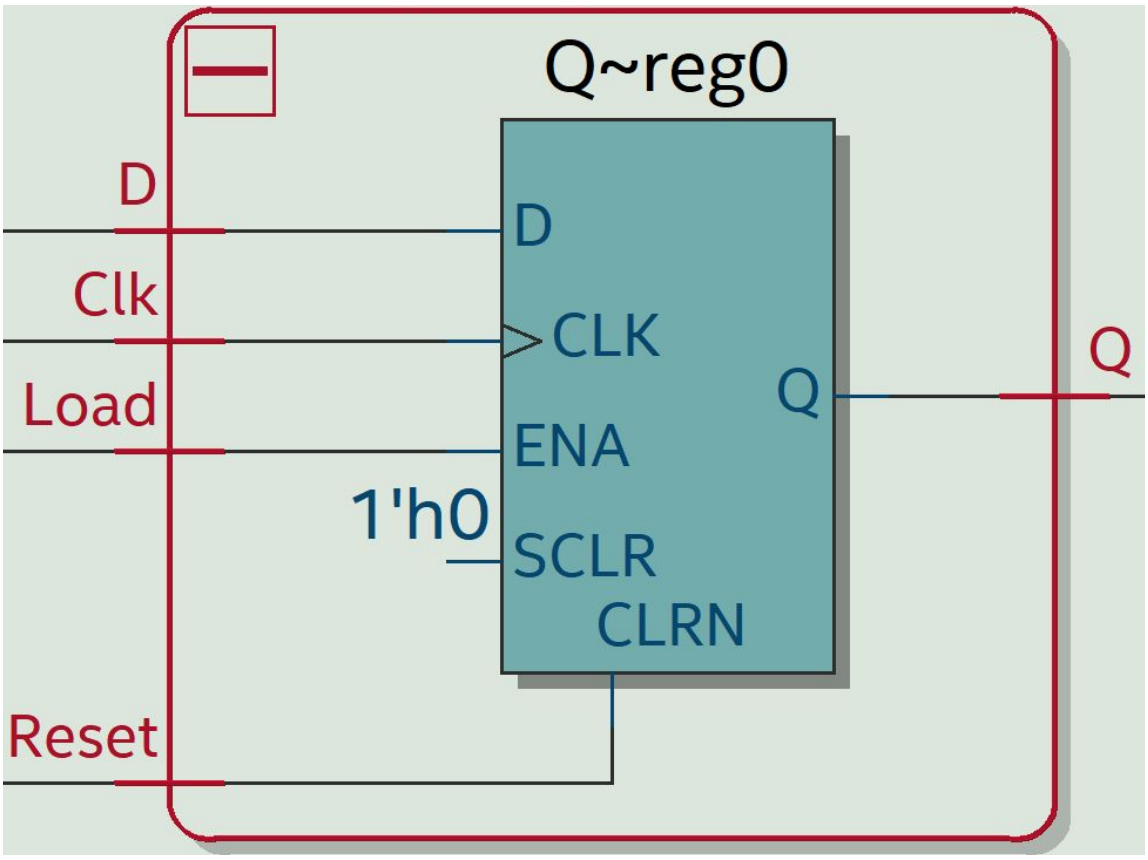

Fig. 13. Dreg Block Diagram

- 11_reg (11_reg.sv)
    - *Inputs*: `Clk, Reset, Shift_In, Load, Shift_En, [10:0] D`
    - *Outputs*: `Shift_Out, [10:0] Data`
    - *Description*: This is a sequential 11-bit shift register. There are three main flags that determine what the register does: `Reset`, `Load`, and `Shift_En`. For `Reset`, the value the register outputs is set to value 0. For `Load`, the value the register outputs is set to the value inputted by `D`. For `Shift_En`, the bits of the value currently being outputted by the register all shift to the right by 1 bit, and the input `Shift_In` is then shifted in as the most significant bit. For all of these commands, they commence only when the respective flag is active after a positive edge from `Clk`. The module then outputs the value the register currently contains, and additionally outputs a bit `Shift_Out`, which represents the original right-most bit. This module was part of the keyboard driver written by Marie Liu and Sai Ma.
    - *Function*: The module that is used to create an array of shift registers that store the make and break codes from the PS/2 keyboard.

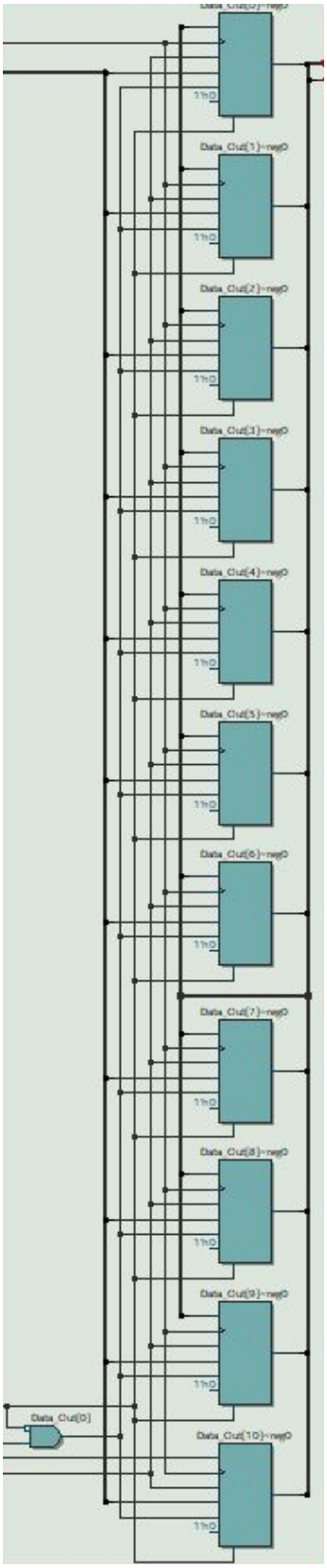

Fig. 14. 11_reg Block Diagram

- color_mapper (Color_Mapper.sv)
    - *Inputs*: `is_sprite1, is_sprite2, is_menu, paused`
      `[3:0] pix_color1, pix_color2,`
      `[9:0] p1_health, p2_health,`
      `[2:0] menu_color,`
      `[9:0] DrawX, DrawY`
    - *Outputs*: `[7:0] VGA_R, VGA_G, VGA_B`
    - *Description*: This is the combinational module that maps an RGB value to a specified pixel on the VGA monitor. `DrawX` and `DrawY` correspond to the coordinates of the current pixel whose color is being updated, and `VGA_R`, `VGA_G`, and `VGA_B` are the respective R, G, and B values that are set and outputted to the specified pixel on the VGA monitor. `is_sprite1` and `is_sprite2` specify if the VGA controller is within the borders of a sprite to be printed. It chooses a different set of colors to map to each index based on whether it is drawing sprite1 or sprite2. Similarly, `is_menu` accesses its own set of RGB values to map to each color index if we are drawing the menu. `paused` is an additional variable that specifies if the game is in a paused state, so the color mapper knows to print the start and end screens over the sprites. `pix_color1` and `pix_color2` are the color indices that are retrieved from the OCM for sprites 1 and 2, respectively. Likewise, `menu_color` is the color index retrieved from the menu OCM when printing a start or end screen. Lastly, `p1_health` and `p2_health` keep track of the health status of each player to ensure that the color mapper knows how to print the health bars.
    - *Function*: The module that outputs a color to a pixel on the VGA monitor. The pixel whose color it outputs is determined by given coordinates, and it decides what color to output based on the current state of the game, as well as the sprite which is being printed.

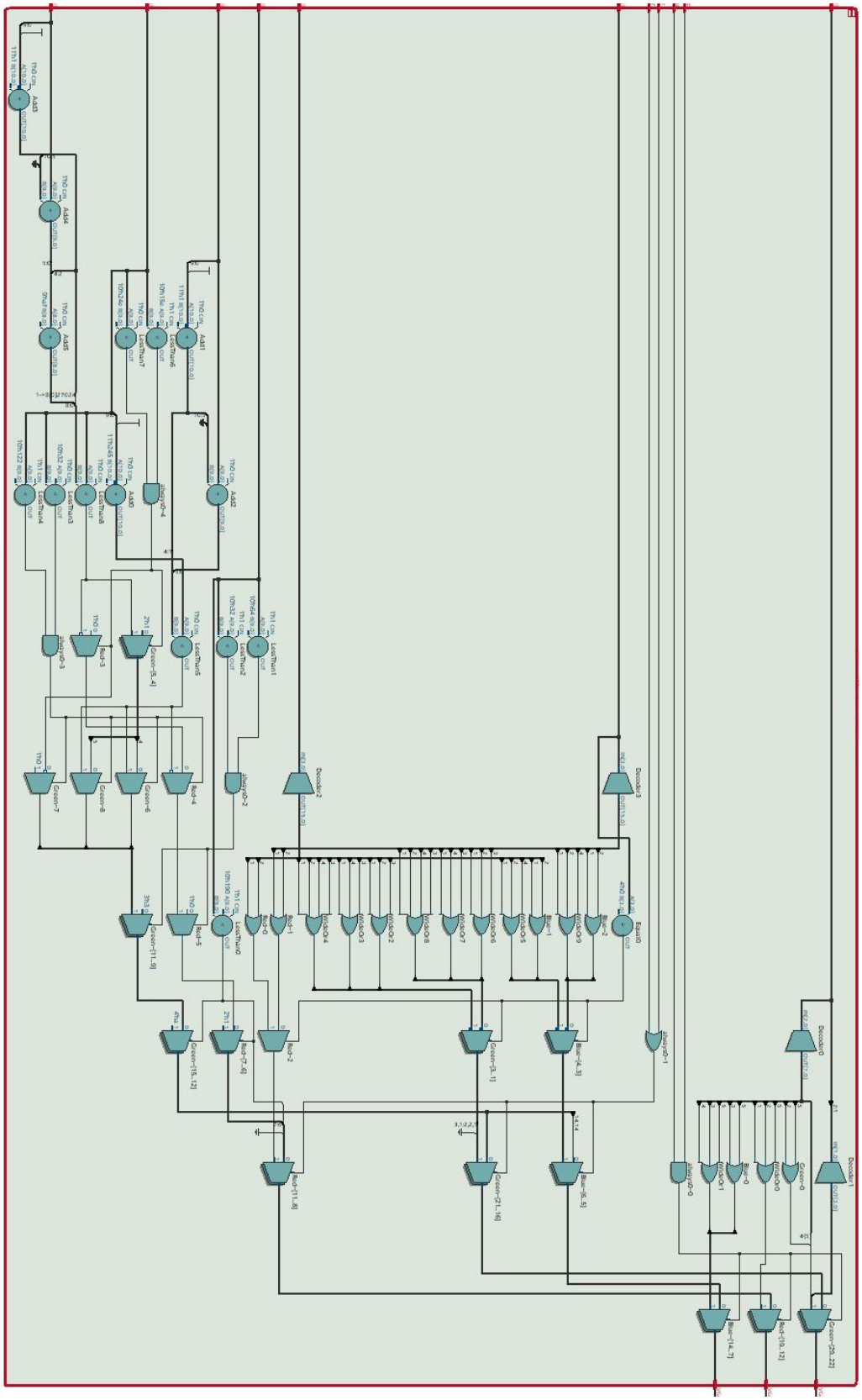

Fig. 15. color_mapper Block Diagram

PIO Block Descriptions

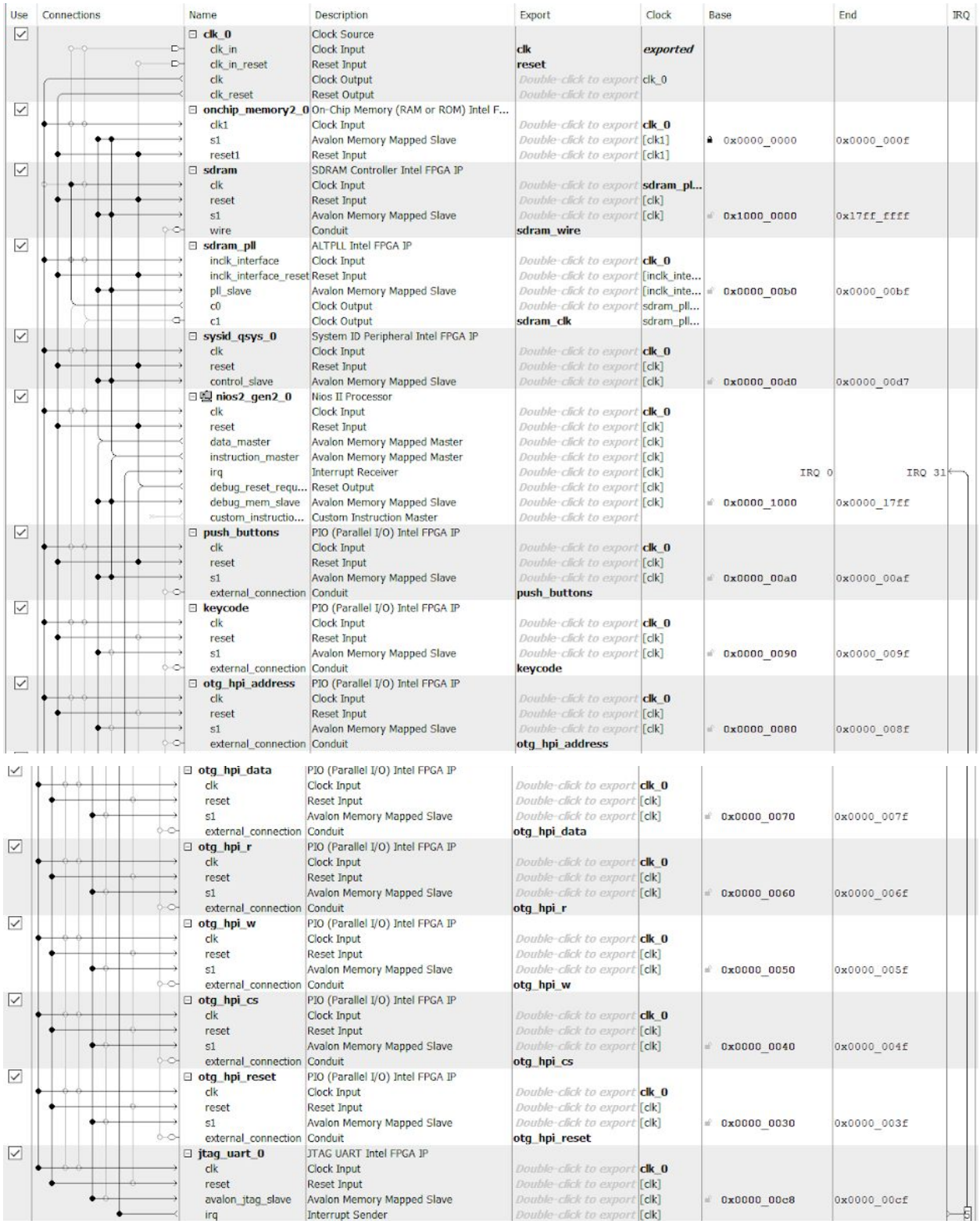

Fig. 16. System Level Block Diagram (Qsys View)

- **keycode**: A block that allows the keycode generated by the USB keyboard to be exported to the NIOS II processor.
- **otg_hpi_address**: A block that allows the intermediate address register of the USB controller chip to interface with the NIOS II.
- **otg_hpi_data**: A block that allows the NIOS II to write data to or read data from the USB controller's data register.
- **otg_hpi_r**: A block that allows the NIOS II to send a read request to the USB controller.
- **otg_hpi_w**: A block that allows the NIOS II to send a write signal to the USB controller.
- **otg_hpi_cs**: A block that allows the NIOS II to send a chip select signal to the USB controller.
- **otg_hpi_reset**: A block that allows the NIOS II to send a signal instructing the USB controller chip to reset.
- **jtag_uart_0**: A serial block that allows the user to communicate with the NIOS II via the console of the computer.

Bugs

We encountered many bugs during the design and implementation process for our final project. Some of the more complex issues we experienced involved overlapping sprite borders, in which one sprite's background was partially printed over the other sprite, seamless animations/timing, and proper incorporation of the SoC for the USB interface. The first problem was solved by instantiating two OCM partitions, one for each player. By this method, we were able to receive two different OCM outputs and choose to prioritize one pixel over the other if the other's index corresponded to the background color.We solved the animation problem using a series of frame counters and conditions that choose which OCM address, and therefore which sprite frame, to print from. The SoC from Lab 8 was incompatible with our project at first due to several syntactic issues involving file names, outdated IPs, and duplicated folders within the filesystem. We ended up restarting Lab 8's SoC from scratch to solve this problem.

Design Resources and Statistics

Fighting Game with USB and VGA Interface

LUT	4363
DSP	14
Memory (BRAM)	3,603,456
Flip-Flop	2908
Frequency	53.23 MHz
Static Power	106.04 mW
Dynamic Power	110.06 mW
Total Power	325.09 mW

Fig. 17. Resources and Statistics for Fighting Game with USB and VGA Interface

Conclusion

Upon completing our final project and analyzing its design statistics, we noticed that it was relatively faster than Lab 8's ball routine, though at a cost of using significantly more resources and power. Our project had to take many factors into account in order to function as a playable bootleg fighting game for two players. Some of the most difficult issues to tackle involved incorporation of player sprites and menus into the game, game state logic to keep track of positions, actions, hitboxes, and health, and player animations. Despite these obstacles, we were able to produce a full-fledged fighting game with all the necessary basic features.