ECE 385

Spring 2019

Final Project

PAC-MAN WITH USB AND VGA INTERFACE IN SYSTEMVERILOG

Yash Tyagi, Aayush Patel

ABJ, Friday, 2:00-5:00 PM

Xinbo Wu, David Zhang

Introduction & Purpose of Circuit

In this final project, we created the classic game called Pac-Man. This game was created all through hardware, with only using the software for reading the inputs from the keyboard. The initial setup of this lab was very similar to that of lab 8, where we had the ball being controlled by the inputs of the keyboard. Pac-Man was controlled by the W, A, S, and D keys on the keyboard, which were to move Pac-Man up, left, down, and right, respectively. There were multiple steps to this project, which will all be described below. We utilized the Universal Serial Bus (USB) and the Video Graphics Array (VGA) ports on the board, in order to interact with the game as well as display it onto a monitor. We also added ghosts to this game, which would chase Pac-Man in an attempt to eat him. If eaten, he would lose a life and the game would start over. These ghosts were made with various levels of difficulty, where one knew exactly where Pac-Man was and would take the shortest path to get to him, to one which simply chose a random direction to go when coming to an intersection with more than two available routes. We also included the food pellets that Pac-Man can eat in order to collect extra points. This project can be broken up into various parts: the visual drawings (which were done with sprites), the collision detection with the pellets, the collisions with the wall, the moving algorithm for Pac-Man, the score increasing, and the movement of the ghosts. There is also the use of the USB and VGA interfaces, which allows us to read the inputs from the keyboard. All of these sections are described below.

The Universal Serial Bus (USB) Interface

Like lab 8, the USB and VGA interfaces are crucial to the functionality of the lab. The USB interface is actually mostly done through the software, although the hardware is an important part of it. The CY7C67200 USB Controller on the DE2-115 board is what gives us the ability for data transmission. The USB controller for this particular lab acts as a host controller, which means that it controls the devices, rather than the device controlling the actual processing. For the process of the lab, the keys W, A, S, and D controlled the movements of Pac-Man. The way this is processed starts with the USB controller processing the keystroke, which will then interact with the rest of the program to have the correct movement. The EZ-OTG in our Platform Designer provides us with the functions IO_write and IO_read, which works alongside with the parallel inputs and outputs. These are called by USBWrite and USBRead, which by the name, write and read to the memory. The functionality of the keyboard interacting with the rest of our game was done with four functions: IO_read, IO_write, USBread, and USBWrite. IO_read's first step was to set the otg_hpi_address to the corresponding Address, which was the parameter of this function. The HPI registers were created by the Platform Designer. Because chip select and read are both active low signals, we set both of them to 0. A temporary variable was used to store the data so that it is not lost. After read and chip select are deactivated (by setting them to 1), we returned the temporary variable, which had the data from the otg_hpi_data. IO_write is completed in a very similar manner. Data was set to the value of otg_hipi_data, and Address was set to otg_hpi_address, which were both parameters of this function. Chip select and write were set to active. Once the data is written, we set write and chip select back to 1, deactivating their

functions. USBWrite and USBRead call IO_write and IO_read, respectively. Both these functions give the processor permission to access the on-chip memory, where we can read and write to. In order to optimize the functionality of these functions, the pointers auto-increment in the read and write process.

The Video Graphics Array (VGA) Interface

The VGA functionality does not require as much set up as the USB did. However, it is still crucial to understand how the VGA functions. The VGA has a screen refresh rate of 60 Hz, which means that some information must be transmitted 60 times a second. This information can be the same information. The Color_Mapper module takes the locations of the objects and colors them in a certain precedence. The VGA_controller contains the horizontal and vertical timing synchronizations that the VGA monitor requires. The screen is 640 x 480 pixels, and by the timing syncs, we can let the electron beam focus on a particular area. This is very essential to the functionality as this is what allows us to know the exact coordinates of a pixel at any given time. Many of the parts in this project like the ghosts and the food pellets rely on knowing a particular location. The collision algorithms for the both of these require us to know exactly what pixel Pac-Man is at, so that we can track whether or not the pixels overlap, thereby telling us whether or not there is a collision. These algorithms and detections are described below in more detail.

Drawing the Sprites & Mapping the Colors/Location

The first step in making this project was to draw our characters and the map. To do this, we create a section on our Read-Only Memory (ROM) for the particular sprite and stores this on the chip. These sprites were encoded by making the drawings out of 1's and 0's, which would help us symbolize the color that the particular pixel should be. We did this for the maze, Pac-Man, as well as the ghosts. These memory addresses are important for the module Color_Mapper. This module is what allows us to draw all the sprites and the food pellets that Pac-Man can eat at any given time. Color_Mapper has an input of DrawX and DrawY, which are the current pixel locations on the screen, pac_x and pac_y, which are the coordinates for Pac-Man's location, pacDir, which are the different direction of Pac-Man's sprite, dotShow, which is a 308 bit array of the food pellets that Pac-Man can eat for more points, and the VGA_R, VGA_G, and VGA_B, which could assign the color to the different sprites and the background. This is done by checking the boundaries for the pixels of a particular sprite. For instance, when we see that the pixels are less than 256 pixels in the DrawX and DrawY is less than 384 pixels, then we know that it is part of the maze, which means that we can draw for the pixels. We have a variable called is_maze to indicate that this is part of the maze, and in this section, we go to the corresponding address where the maze is and access the data to draw the rest of the maze with the correct wall placement.

This process was also used for drawing Pac-Man. We checked if DrawX and DrawY were within an 8-pixel radius of Pac-Man's center. This condition was used in order to see

whether or not the location of Pac-Man was within one of our 16-pixel boxes. If it was, the Pac-Man sprite was drawn. We also used this condition to check which direction Pac-Man was facing. If the direction of Pac-Man was left, for example, then the address would jump to the address where the left sprite was creating, using an offset from the address of the ROM in 16-bit intervals. This was done for all four directions Pac-Man could face (up, down, left, and right).

Last but not least, we mapped the food pellets so that Pac-Man could eat them for extra points. This was done by first seeing if the pellets were in the scope of the map (256 pixels by 384 pixels) and then converting the two-dimensional array that we had to a one-dimensional array, corresponding to the number of rows and columns that we had. This formula was the current y-position times the number of columns plus the x-position. We omitted the last 16-bits because we wanted to have these dots every 16 pixels, and we ensured that they were in the center of the block by checking if the last four bits were less than four pixels. This mapped dots all throughout our map, so that there is are equally spread dots throughout the maps. Additionally, the dots behind walls are not able to be collected as the collision (described in detail below) prevents them from ever being accessed.

Wall Collision Detection

The next step in the game was to create the collisions between the walls and Pac-Man. In order to do this, we first scaled down our maze from a 256 by 384 pixels to a 16 by 24-pixel maze. This was done so that we could deal with the blocks of the maze in 16x16 pixel blocks. This maze was created as a read-only maze, which means it only had an output so that it could be used by other modules. It is also generalized to more than just Pac-Man so that it could also be used for the wall collision detection for the ghosts. We then created another module called findWalls. This module had an input of the new 16x24 maze, the current x-position and y-position, and also an output of up, right, down, and left, which signify whether or not there is a wall in that direction. We first created temporary variables for the x-position and the y-position which maps 16-pixel increments for each block. Then for every direction, we converted each data of the wall into the corresponding one-dimensional location. For example, the left wall had the temporary y-position multiplied by the number of columns plus the x-position subtracted by one, which indicated the left block of the current one. This was done for each wall, which were then set to the associated output variables. These variables were used to restrict the movement of Pac-Man as he continues to move, so that he does not go through a wall. This process is described in more detail below.

Pac-Man Movement

After setting up all the sprites and our wall collision detection, it was time to make the movements of Pac-Man. The inputs for the Pac-Man module are the clock, reset, frame_clk, keycode, which contains the configuration of the W, A, S, and D keys for Pac-Man's movement, and wallData, which is the 16x24 maze which contains 16-pixel blocks. The outputs are pac_x and pac_y, which are the current x and y positions of Pac-Man, u, l, d, and r, which are the up,

left, down, and right walls (mostly used for debugging), pacDir, which is the current direction of Pac-Man, and crossing, which tells us if we are at an intersection in the maze or not. The first thing that we did was set default locations for Pac-Man and defined all of our constants, like the minimum and maximum of the of x and y positions. We assigned the directions accordingly and also made our output variable crossing to be the position which is in the center of a 16-pixel block. We then mapped each of the keycodes for W, A, S, and D to each matching direction (up, left, down, and right). The turn_in variable set to a one for whatever direction it matched. The next step is to check if we are at a crossing, so that Pac-Man has the capability of turning if no wall is in the direction of desired movement. This is done by checking if the turn variable is a certain direction and if there is no wall in that particular direction, which is using the variables that came from the findWalls module. While doing this, we noticed that Pac-Man would be one pixel off, so this was accounted for by simply adding one pixel in the other directions. We repeated this step for every direction, making sure to add or subtract one pixel depending on the direction that is received. We decided to have the game where if no direction is pressed, then there is no movement and Pac-Man does not move. This was because the wall detection would glitch when there was no input, and we figured it makes the game a bit harder to play. If no crossing is found, then Pac-Man will continue in whatever direction until a crossing is encountered.

Food Pellets (with Score)

The next major part of our project was to configure all the pellets in the game so that they could be eaten and added to the score based on how many pellets were consumed. The first module that we created was the dot module. This required inputs of the clock, reset, dotX and dotY, which were the x and y positions of the dots, pacX and pacY (the x and y positions of Pac-Man), and an output logic called show, which tells us if the dot has been eaten or not (whether it is seen or not seen). In order to update the show variable for each dot, we decided to make a radius around Pac-Man signify a pellet has been eaten rather than the coordinates matching up perfectly. This radius was six pixels. The extra condition at the end was to check if the pellet was even still there, as it may have already been eaten. If these conditions were all satisfied, we changed the show variable to one, in order to indicate that the pellet has been eaten. From here, we made rows of dots, for each row in our maze. This code is in the dotRow module. Here we map each dot in a row to be evenly spaced out and in the center of every block, by starting at pixel 24 and incrementing by 16 each time. This module was then used in dotGrid, which makes multiple rows for each row that we had in our game (22 rows). By the end of this process, we have 208 edible dots that are equally spaced throughout the map and that will update the array as we continue. The last thing was to add up the score. We had another module called scoreCounter, which took inputs of a clock and the 308 show bits for each pellet. The output was a simple number that would contain the value of all the pellets that are not seen, thereby indicating that the pellet has been eaten. We made a for loop that would go through every bit in the array, and for every pellet which has a show variable value of one, we added it to a temporary variable that was then stored in the output. From here, we followed the tutorial that

was on the website for fonts. After making the modules that contained the numbers zero through nine, we mapped each number to the corresponding score. We decided that the score would be every four pellets you get would give you one point. This means that there is a possibility to get 50 points. In Color_Mapper, we used the algorithm that was provided in the tutorial in order to map the correct numbers. We also made them have an offset so that they would be to the side of our maze. Each score was mapped to the corresponding score. For example, the score 17 was mapped to have the first digit be the sprite 1 and the second digit to be 7. Each digit had a value to see if it was active, and if it was, the correct pixels were mapped. By the end of this process, we had the score display correctly to the right of the maze.

Super Pellet

In addition to the food pellets that Pac-Man can eat, we added super pellets that, once eaten, would be able to stop the ghosts' movement for about five seconds. We added four of these pellets to various locations on the map and changed the color of these pellets in order to signify their special ability. We simply added an if statement to the very beginning of the movement code that contained if (~freeze). This would indicate that the ghosts should not be frozen and that they could continue normal movement. However, once the ghosts have been frozen due to the freeze pellet, they would not be able to move or kill Pac-Man if he touches them in this time. This means that Pac-Man is allowed to travel through the ghosts if it is during the time of a freeze pellet. To control the time of the freeze pellet, we did some math based off the clock speed. Because the clock speed is 50 MHz, we new that we would need at least 26 bits to count up to this value. However, if we only counted up to this, that would signify a freezing time of 1.3 seconds. So the closest value that would give us five seconds was using 28 bits, and counting up to this value. Once a freeze pellet was eaten, it would count up to this number with a clock speed of 50MHz, which would be about 5.3 seconds (2^{28}/50,000,000).

Ghost Movement

The ghost movement required the most logic in terms of figuring out how we would make our ghosts have an artificial intelligence to chase Pac-Man. To begin, the ghosts, like Pac-Man, has a x and y position, and were additionally fed in Pac-Man's location. The ghosts were also coded with wall detection, in a similar manner to Pac-Man. This was to ensure that the ghosts were not able to go through the wall when trying to catch Pac-Man. We then had multiple signals that were used for tracking the previous turn, and for creating a movement pattern for the ghosts to use. The ghosts decide which path to take based on Pac-Man's location. These cases include if Pac-Man is right and up, if he is left and down, if he is right and down, and if he is left and up. These cover the diagonal cases for Pac-Man, relative to the location of the ghosts. We then made them have a case for when Pac-Man is right, left, up, and down, relative to their locations. These cases were coded in order to ensure that they knew how to take the path to Pac-Man. In order to give a better understanding of these, here is an example of the left and up case. To check these conditions, we first see if pac_x (the x location of Pac-Man) is less than the

ghostX (the x position of the ghost) and we do the same for the y position. If this condition is true, we then check if there is a wall to the left. If there is no wall to the left, the ghost would turn left. This was copied for the remaining directions. However, when we ran this code, we saw that there was an issue where the ghosts would just bounce repeatedly, because Pac-Man's location would be reached by going through the wall. To fix this, we added a previous signal, which contains the previous direction that the ghost tried. So, for the left wall, we checked if there was a left wall and if the previous turn was not right, indicating that it was trying to bounce. We repeated these steps for all the other cases mentioned above.

Once we implemented this, we realized that all the ghosts would behave in the same way, because of the fact that the Artificial Intelligence that we made would be followed by every ghost. In order to make the game more enticing, we decided to implement a random function. To do this, we added four Parallel Input/Output (PIO) block that was mapped to the rest of our circuit. This gave us the capability to use software. In the C code, we used the srand() function and the rand() function. We added three different PIOs so that there would be different randomized directions and numbers for each ghost. In this C code, we added a seed for srand() that was based off of the time. There were two different randomized variables. The first one was a 2-bit export called rnd_dir, which was for the four randomized directions that the ghosts would be able to take (up, down, left, and right). The other one was called rnd_percent, which was a 3-bit export used to determine how often the ghosts would take the random path and how often they would take the artificial intelligence path. Because these were exported into our hardware, we were then able to use these as inputs for the ghost movement. In order to make the game more enjoyable and difficult, we gave each ghost a different level of randomness. For example, the red ghost in our game would always take the artificial intelligence path, whereas the orange ghost would take the randomized path half the time, and the artificial intelligence path the other half of the time. This difficulty was controlled by a simple check of the random generated. Because the rnd_percent variable was three bits, the number generated would be between zero and seven. So for the orange ghost, we said that it would take the rnd_dir direction if the rnd_percent was greater than three, as this would be half the time. This process was repeated for the remaining ghosts, until each of them had a particular combination of randomness and artificial intelligence.

Game Settings

The last thing that we did was make functionality to control how the game was played. To signify that the game has started, we had the ghosts and Pac-Man default to a certain position, and to not have them move until a key was pressed on our keyboard. This would then release the ghosts from the ghost box and move Pac-Man in the corresponding direction of the key. The game would continue until one of two conditions: either all 200 food pellets were collected, or the ghosts would catch Pac-Man. Once one of these situations occurred, a text block that said game over would be displayed on the screen, on top of the score. To adjust the difficulty of the game, we mapped each ghost to a particular switch. This meant that the number of ghosts that were active during a game could be controlled. In addition, because each ghost had a different

level of artificial intelligence and randomness, the difficulty of the game could be adjusted by choosing which ghosts were desired. Once the game over signal was detected, Pac-Man would not be able to move regardless of what keys are pressed on the keyboard, and the score would have the final score. To start the game again, the first key (key[0]) could be pressed to reset the game and to decide how many ghosts could be added. The ghosts could be added anytime during gameplay but cannot be subtracted. The ghosts would catch Pac-Man depending on a hit box that was set up around the ghosts and Pac-Man. If the ghosts came inside twelve pixels of Pac-Man from any direction, that would indicate that Pac-Man has been caught and that the game over banner should be displayed.

Written Description of all .sv Modules

hpi_io_intf.sv

Module: hpi_to_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
Bidirectional: [15:0] OTG_DATA
Description: This module is the interface between the NIOS II and the EZ-OTG chip.
Purpose: This module is used so that the USB controller can interact with the NIOS II interface. This is what allows us to use the keyboard to interact with the board, making pac-man move depending on the input that is provided.

VGA_controller.sv

Module: VGA_controller
Inputs: Clk, Reset, VGA_CLK
Outputs: VGA_HS, VGA_VS, VGA_BLANK_N, VGA_SYNC_N, [9:0] DrawX, [9:0] DrawY
Description: This module allows for the program to interact with the VGA monitor and for it to function properly.
Purpose: This module gives us the ability to display sprites such as pac-man and the ghosts. In this module, there is a vertical and horizontal timing sync so that the electron beam can be focused and the location is known at any time. It displays the pixels between the ranged 640x480, which is the size for VGA.

pacman_tp.sv

Module: pacman_tp
Inputs: CLOCK_50, [3:0] KEY, [17:0] SW, [17:0] LEDR, OTG_INT
Outputs: [6:0] HEX0, [6:0] HEX1, [6:0] HEX2, [6:0] HEX3, [6:0] HEX4, [6:0] HEX5, [6:0] HEX6, [6:0] HEX7, [7:0] VGA_R, [7:0] VGA_B, [7:0] 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, DRAM_CAS_N, DRAM_CKE, DRAM_CS_N, [3:0] DRAM_DQM, DRAM_RAS_N, DRAM_WE_N, DRAM_CLK

Bidirectional: [15:0] OTG_DATA, [31:0] DRAM_DQ
Description: Top level module that combines NIOS II system with the DE2 board hardware. It also lets the VGA and USB to connect to the board and function properly.
Purpose: Connects the buttons, hex displays, VGA driver, USB interface, and memory on the board to the NIOS II system generated by the platform designer. This also connects all of our internal hardware such as the pac-man and ghost control modules.

HexDriver.sv

Module: HexDriver
Inputs: [3:0] In0
Outputs: [6:0] Out0
Description: This module maps the binary bits into hexadecimal values.
Purpose: This module allows us to map any 4-bit registers to the hexadecimal drivers. In order to display the binary bits loaded into the switches to the hexadecimal displays, we mapped every 4-bit value from 0 to 15 (unsigned) into 7-bit hexadecimal "line" values, which program the HEX displays to show the number that we are used to seeing. Used mostly for debugging.

Color_Mapper.sv

Module: color_mapper
Inputs: [9:0] DrawX, [9:0] DrawY, [9:0] pac_x, [9:0] pac_y, [3:0] pacDir, [9:0] ghost1X, [9:0] ghost1Y, [9:0] ghost2X, [9:0] ghost2Y, [9:0] ghost3X, [9:0] ghost3Y, [9:0] ghost4X, [9:0] ghost4Y, [307:0] dotShow, [7:0] score, ggShow, sd1Show, sd2Show, sd3Show, sd4Show
Outputs: [7:0] VGA_R, [7:0] VGA_G, [7:0] VGA_B
Description: This module decides which color to be outputted onto the FPGA monitor for every pixel on the screen.
Purpose: This module sets the color for sprites such as pac-man, ghosts, and the background. This is done by using inequalities to set zones around a sprite's coordinates. When drawing in this zone, the color will be chosen from a sprite ROM. Is also used to give priority for drawing layers. (Ghosts have to be drawn on top of dots)

pac2.sv

Module: pac2
Inputs: Clk, Reset, frame_clk, [7:0] keycode, [383:0] wallData, ggShow
Outputs: [9:0] pac_x, [9:0] pac_y, [3:0] pacDir, start
Description: Pac-man controller module.
Purpose: Uses keyboard inputs to control all movements of pac-man. It uses the findWalls module to check wall collisions before moving. Also keeps track of direction so that it can be drawn accordingly by the color mapper.

pacman_soc.v

Module: pacman_soc
Inputs: clk_clk, [15:0] otg_hpi_data_in_port, 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, [1:0] rnd_dir1_export, [1:0] rnd_dir2_export, , [1:0] rnd_dir3_export, [2:0] rnd_percent_export
Bidirectional: [31:0] sdram_wire_dq
Description: The SoC module generated by platform designer.
Purpose: Connects all the internal modules and peripherals that the designer generated into one final CPU. Connects to top level module which connects this CPU to its memory and IO.

pacman_soc_sysid_qsys_0.v

Module: pacman_soc_sysid_qsys_0
Inputs: address, clock, reset_n
Outputs: [31:0] readdata
Description: Ensures software and hardware compatibility with the NIOS II
Purpose: Makes sure that the software that is being loaded onto the CPU is compatible and not a different version/configuration.

pacman_soc_sdram_pll.v

Module: pacman_soc_sdram_pll
Inputs: [1:0] address, areset, clk, configupdate, [3:9] phasecounterselect, phasestep, phaseupdown, read, reset, scanclk, scansclkena, scandata, write, [31:0] writedata
Outputs: c0, c1, locked, phasedone, [31:0] readdata, scandataout, scandone
Description: Takes care of SDRAM synchronization
Purpose: Since the NIOS II is a SoC, we must connect the SDRAM chips on the DE2 board to the FPGA and CPU. Due to the timing delay caused by the physical location of the memory and FPGA, the CLK signal is purposely shifted to account for this.

pacman_soc_sdram.v

Module: pacman_soc_sdram
Inputs: [24:0] az_addr, [3:0] az_be_n, az_cs, [31:0] az_data, az_rd_n, az_wr_n, clk, reset_n
Outputs: [31:0] za_data, za_valid, za_waitrequest, [12:0] zs_addr, [1:0] zs_ba, zs_cas_n, zs_cke, zs_cs_n, [3:0] zs_dqm, zs_we_n
Bidirectional: [31:0] zs_dq
Description: Connects CPU to external memory
Purpose: Since the on-chip memory is too small, external memory is used. This module is the controller that connects the SoC to the SDRAM on the DE2 board. It also makes sure the SDRAM is constantly refreshed to keep its data.

pacman_soc_rnd_percent.v

Module: pacman_soc_sdram
Inputs: [24:0] az_addr, [3:0] az_be_n, az_cs, [31:0] az_data, az_rd_n, az_wr_n, clk, reset_n
Outputs: [31:0] za_data, za_valid, za_waitrequest, [12:0] zs_addr, [1:0] zs_ba, zs_cas_n, zs_cke, zs_cs_n, [3:0] zs_dqm, zs_we_n
Bidirectional: [31:0] zs_dq
Description: PIO block that constantly generates a 3-bit random number and outputs it.
Purpose: Used to determine difficulty of ghosts. The 3-bit number (0-7) is used as a percentage to decide whether the ghost uses AI to make a choice or randomness to make a choice.

pacman_soc_otg_hpi_data.v

Module: pacman_soc_otg_hpi_data
Inputs: [1:0] address, chipselect, clk, [15:0] in_port, reset_n, write_n, [31:0] writedata
Outputs: [15:0] out_port, [31:0] readdata
Description: This module is what allows the data to be transferred and used in the rest of the program.
Purpose: This module is one of the registers that are needed for IO_write and IO_read to be implemented. This is one of the HPI registers which is then passed into the Hardware Tri-state Buffer, which is used for the NIOS II to run the program and function properly. This communicates with the RAM so that the correct memory is taken and implemented in the rest of the system.

pacman_soc_otg_hpi_cs.v

Module: pacman_soc_otg_hpi_cs
Inputs: [1:0] address, chipselect, clk, reset_n, write_n, [31:0] writedata
Outputs: out_port, [31:0] readdata
Description: This module is what allows the chip select to be activated so that the data is written and read from only when we allow it to.
Purpose: This module is one of the registers that are needed for IO_write and IO_read to be implemented. This is one of the HPI registers which is then passed into the Hardware Tri-state Buffer, which is used for the NIOS II to run the program and function properly. This communicates with the RAM so that the correct memory is taken and implemented in the rest of the system.

pacman_soc_otg_hpi_address.v

Module: pacman_soc_otg_hpi_address
Inputs: [1:0] address, chipselect, clk, reset_n, write_n, [31:0] writedata
Outputs: [1:0] out_port, [31:0] readdata
Description: This module contains the correct address of what data is written and read from and ensures the successful running of our interface.
Purpose: This module is one of the registers that are needed for IO_write and IO_read to be implemented. This is one of the HPI registers which is then passed into the Hardware Tri-state Buffer, which is used for the NIOS II to run the program and function properly. This communicates with the RAM so that the correct memory is taken and implemented in the rest of the system.

pacman_soc_onchip_memory2_0.v

Module: pacman_soc_onchip_memory2_0
Inputs: [1:0] address, [3:0] byteenable, chipselect, clk, clken, freeze, reset, reset_req, write, [31:0] writedata
Outputs: [31:0] readdata
Description: The on-chip memory for the NIOS II
Purpose: Creates a small amount of RAM on the chip as a placeholder. Since it would be inefficient to create large memory on the FPGA, we use external SDRAM.

pacman_soc_nios2_gen2_0.v

Module: pacman_soc_nios2_gen2_0
Inputs: clk, reset_n, reset_req, [31:0] d_readdata, d_waitrequest, [31:0] i_readdata, i_waitrequest, [31:0] irq, [8:0] debug_mem_slave_address, [3:0] debug_mem_slave_byteenable, debug_mem_slave_debugaccess, debug_mem_slave_read, debug_mem_slave_write, [31:0] debug_mem_slave_writedata
Outputs: [27:0] d_address, [3:0] d_byteenable, d_read, d_write, [31:0] d_writedata, debug_mem_slave_debugaccess_to_roms, [27:0] i_address, i_read, debug_reset_request, [31:0] debug_mem_slave_readdata, debug_mem_slave_waitrequest, dummy_ci_port
Description: The base version of NIOS II without any peripherals connected.
Purpose: The CPU is in this module which will be later used to connect peripherals such as the otg chip, memory, buttons, hex displays.

pacman_soc_keycode.v

Module: pacman_soc_keycode
Inputs: [1:0] address, chipselect, clk, reset_n, write_n, [31:0] writedata
Outputs: [7:0] out_port, [31:0] readdata
Description: This module allows us to use the keyboard inputs and for it to be read.
Purpose: This module gives us the ability to read the inputs from the keyboard into software and for it to translate accordingly to the movement of pac-man on the screen.

pacman_soc_jtag_uart.v

Module: pacman_soc_jtag_uart_0
Inputs: av_address, av_chipselect, av_read_n, av_write_n, clk, rst_n, [31:0] av_writedata
Outputs: av_irq, av_waitrequest, dataavailable, readyfordata, [31:0] av_readdata
Description: This module provides the ability to communicate character streams with the rest of the interface.
Purpose: This module is relevant as it gives us the interrupt sender, which gives us the ability to set the IRQ. The interrupts allow for the transmitting and receiving of text to not block on the CPU. This is particularly useful for printf statements.

scoreChars.sv

Module: scoreChars
Inputs: [3:0] addr
Outputs: [47:0] data
Description: This module serves as a ROM (Read only memory) for the “SCORE:” sprite.
Purpose: Contains a 2-dimensional memory array which can be accessed by an address input. The address input will return a N-bit word in the data output. This data can be used in color mapper to draw a row of a sprite.

numbers.sv

Module: numbers
Inputs: [7:0] addr
Outputs: [7:0] data
Description: This module serves as a ROM (Read only memory) for sprite data

Purpose: Contains a 2-dimensional memory array which can be accessed by an address input. The address input will return a N-bit word in the data output. This data can be used in color mapper to draw a row of a sprite.

gameOver.sv

Module: gameOver
Inputs: [3:0] addr
Outputs: [71:0] data
Description: This module serves as a ROM (Read only memory) for the "Game Over" sprite.
Purpose: Contains a 2-dimensional memory array which can be accessed by an address input. The address input will return a N-bit word in the data output. This data can be used in color mapper to draw a row of a sprite.

maze.sv

Module: maze
Inputs: [8:0] addr
Outputs: [255:0] data
Description: This module serves as a ROM (Read only memory) for the background maze sprite.
Purpose: Contains a 2-dimensional memory array which can be accessed by an address input. The address input will return a N-bit word in the data output. This data can be used in color mapper to draw a row of a sprite.

sprites.sv

Module: sprites
Inputs: [6:0] addr
Outputs: [15:0] data
Description: This module serves as a ROM (Read only memory) for the pac-man sprite.
Purpose: Contains a 2-dimensional memory array which can be accessed by an address input. The address input will return a N-bit word in the data output. This data can be used in color mapper to draw a row of a sprite.

dotSprite.sv

Module: dotSprite
Inputs: [1:0] addr
Outputs: [3:0] data
Description: This module serves as a ROM (Read only memory) for the dot sprite.
Purpose: Contains a 2-dimensional memory array which can be accessed by an address input. The address input will return a N-bit word in the data output. This data can be used in color mapper to draw a row of a sprite.

ghostSprite.sv

Module: ghostSprite
Inputs: [6:0] addr
Outputs: [15:0] data
Description: This module serves as a ROM (Read only memory) for the ghost sprite.

Purpose: Contains a 2-dimensional memory array which can be accessed by an address input. The address input will return a N-bit word in the data output. This data can be used in color mapper to draw a row of a sprite.

mazeWalls.sv

Module: mazeWalls
Outputs: [383:0] wallData
Description: This module serves as a ROM (Read only memory) for all of the walls in the maze.
Purpose: Outputs a 1-dimensional memory array which can be used to determine where every wall is on the maze. Used by the findWalls module to check the surrounding walls of pac-man and the ghosts.

ghost.sv

Module: ghost
Input: Clk, Reset, frame_clk, [9:0] gStartX, [9:0] gStartY, [383:0] wallData, [9:0] pac_x, [9:0] pac_y, start, [2:0] AIpercent, [1:0] rnd_dir, ggShow, on, freeze
Outputs: [9:0] ghostX, [9:0] ghostY, ggOut
Description: Ghost controller module.
Purpose: Controls all decision making and movements of the ghost. Uses pac-mans location, randomness, and wall detection to make a turn decision. Also has logic for pac-man collision and end game procedures.

superDot.sv

Module: superDot
Input: Clk, Reset, [9:0] dotX, [9:0] dotY, [9:0] pacX, [9:0] pacY
Outputs: show, freeze
Description: Single super dot controller.
Purpose: Has an internal control FSM to wait until it collides with pac-man, then disappear. When it collides with pac-man it also controls a freeze signal for the ghosts. This is done by a counter that lasts about 5 seconds.

dot.sv

Module: dot
Input: Clk, Reset, [9:0] dotX, [9:0] dotY, [9:0] pacX, [9:0] pacY
Outputs: show
Description: Single dot controller.
Purpose: Used in dotRows to make a row of dots to put on the maze. Has an internal control FSM to wait until it collides with pac-man, then disappear.

dotGrid.sv

Module: dotGrid
Input: Clk, Reset, [9:0] pacX, [9:0] pacY
Outputs: [307:0] show
Description: Controller for all 200 dots.

Purpose: Uses dotRow module to assemble all 200 total dots together. The output show array is used by scoreCounter to count the number of hidden dots to calculate the score. The array is also used by color mapper to draw every dot. Each dot is assigned a Y coordinate.

Module: dotRow
Input: Clk, Reset, [9:0] dotY, [9:0] pacX, [9:0] pacY
Outputs: [13:0] show
Description: Controller for a 14 dot row.
Purpose: Uses the dot module to assemble a row of 14 dots. The output show array is used in the larger dotGrid to control all of the eventual dots. Each dot is assigned a X coordinate.

Module: dotRowBox
Input: Clk, Reset, [9:0] dotY, [9:0] pacX, [9:0] pacY
Outputs: [13:0] show
Description: Controller for a 8 dot row.
Purpose: Uses the dot module to assemble a row of 8 dots. This specific row is for the row that contains the ghost box. The output show array is used in the larger dotGrid to control all of the eventual dots. Each dot is assigned a X coordinate.

scoreCounter.sv

Module: scoreCounter
Input: Clk, [307:0] in
Outputs: [7:0] out
Description: Counts the number of 1's in an array
Purpose: Takes the dotShow array from dotGrid and counts the number of dots that are no longer showing. This turns out to be the score of the game.

scoreDisplay.sv

Module: scoreDisplay
Input: [5:0] score
Outputs: [7:0] numAddr1, [7:0] numAddr2
Description: Memory offset calculator for the numbers ROM
Purpose: This module takes the current score and outputs the number ROM addresses of the both score digits that need to be displayed. All numbers are stored in the same ROM so this module uses 16 bit increments to output the correct address for color mapper to use.

findWalls.sv

Module: findWalls
Input: [383:0] wallData, [9:0] xPos, [9:0] yPos
Outputs: up, right, left, down
Description: Wall collision controller
Purpose: Used by pac-man and the ghosts to do wall detection. This module takes in the coordinates of any sprite and outputs the status of the 4 surrounding walls. It does the calculations by using the output array of the wallData module.

Design Resources and Statistics

LUT	24586
DSP	0
Memory (BRAM)	6.912 kB
Flip Flop	2738
Frequency	144.84 MHz
Static Power	106.25 mW
Dynamic Power	0.76 mW
Total Power	199.13 mW

When doing this project, we were not at all concerned with efficiency. All of our sprites were created in ROM modules on-chip. It probably would have been more efficient to create these sprites on external memory like the SDRAM instead of letting them be instantiated on-chip. However, since the sprites were pretty small, the FPGA still had plenty of elements left and we had no shortage of resources. The one thing that we tried to make sure was limit our DSP block usage. We ended up using none. We used multiplications for converting between a two dimensional and one-dimensional array. To make sure all of our arithmetic and multiplications were not too big, we did all of our logic on a condensed 16x24 board then scaled it up to a 256x384 on the display. Limiting the size of our calculations while still keeping them accurate was important so that the game does not lag or use up unnecessary space.

Conclusion

This final project was definitely difficult and had a lot of debugging in the process, especially because we were starting with blank files as opposed to previous labs, where we were given some base files and a procedure to follow. However, when we were compiling our final project proposal, we had a broad timeline that helped us understand exactly how we need to progress and how to systematically approach making this game. However, there were certain parts of our project that took a lot longer than we anticipated. The first of these things was wall detection. When we first started to make the movement for Pac-Man, our wall detection did not work, as Pac-Man would float right through walls and even out of the map. After multiple attempts at different solutions, we created a 16x24 maze, that was a replica of the maze scaled down. This was done so that Pac-Man could move in 16-block increment, and the wall detection would be updated on each block. Like described earlier, we took this maze and converted it into a one-dimensional array that we could use for detecting whether a wall was present or not. Another difficulty that we faced was trying to make the movement for the ghosts. When we first started making our ghosts move with artificial intelligence, they would try to go through the wall

in order to reach Pac-Man, and this situation would cause them to bounce back and forth on a wall until Pac-Man reached a different point on the map. As mentioned before, we added a previous direction signal that would let us keep track of the direction that the ghost had travelled. With this signal, we were able to add the condition to tell the ghosts that they should not be able to travel backwards repeatedly. While doing this project, we did not only learn a lot about SystemVerilog, but also a lot about the design process of making a complicated game like this. We started designing this game by making the sprites and simply just running at the code, but we soon realized that when we took the time to see how it would be synthesized in SystemVerilog, it was quite a bit easier to write the code. As soon as we had the basic structure of our code down, we were also able to add a lot of the more advanced features easier. This is because we made a lot of our code modular, so that they could be easily reimplemented into the other modules that were similar. Overall, this project was quite successful as we were able to implement all of the features we discussed in our project proposal, as well as some additional features like artificial intelligence, on screen score display, and difficulty ranges of the ghosts.

Some things that we could add to this project in the future would be the mini-game that was discussed in our project proposal. This would require us to create another module for mouse compatibility, and a screen transition to another level. We could also add a multiplayer capability, so that two people could play on the same map at once. This would require us to add an n-key rollover so that more than one key would be read at a time. It would also be possible for us to add more difficult artificial intelligence that could have some type of implementation of a shortest path. Last but not least, we could add pellets that allow Pac-Man to eat the ghosts like the classic Pac-Man game. Overall, this project was very successful and was a great learning experience in hardware design and the design process.

Block Diagrams & Pictures of Pac-Man

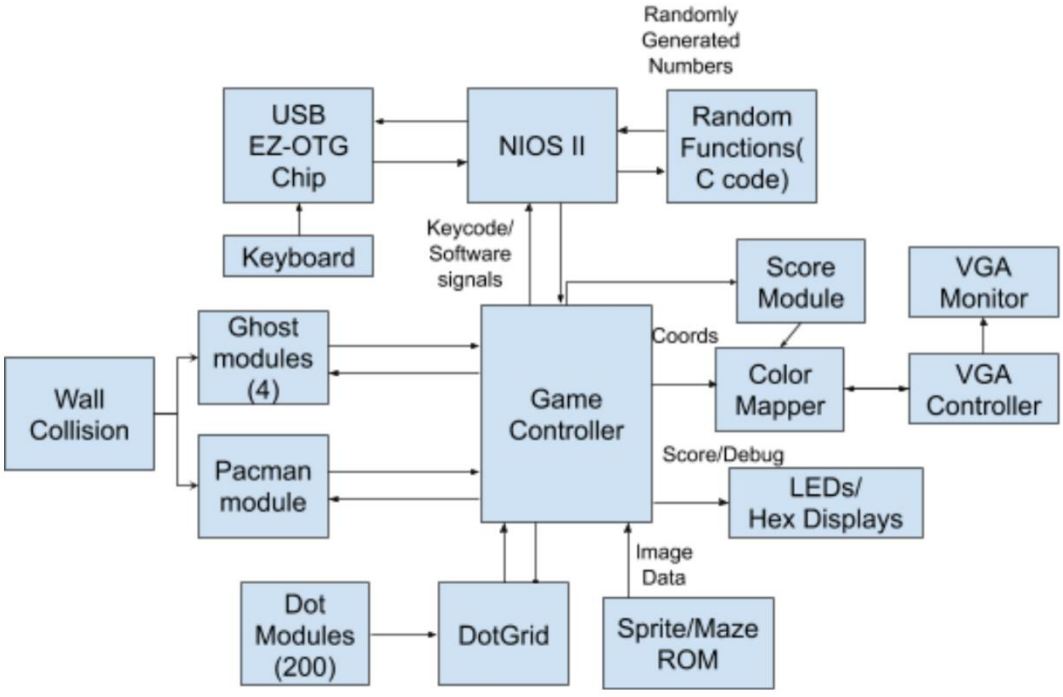

Figure 1: High-Level Block Diagram of All Modules

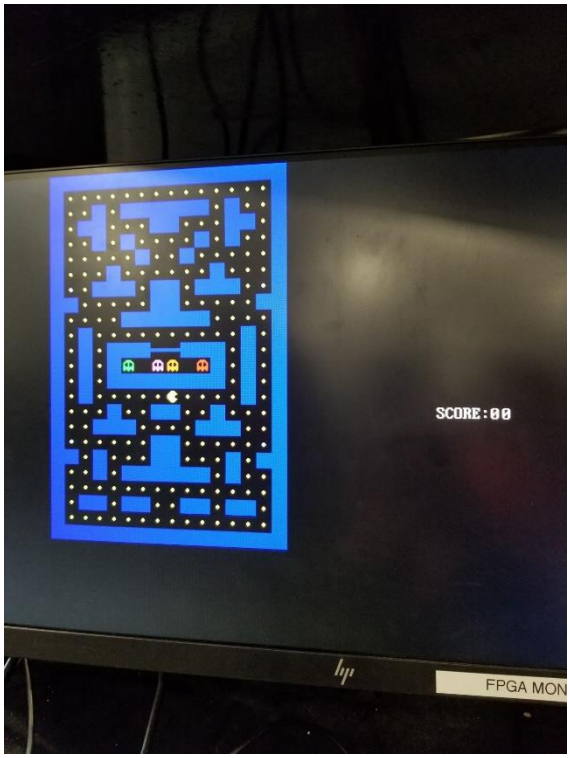

Figure 2: Starting Screen of Game

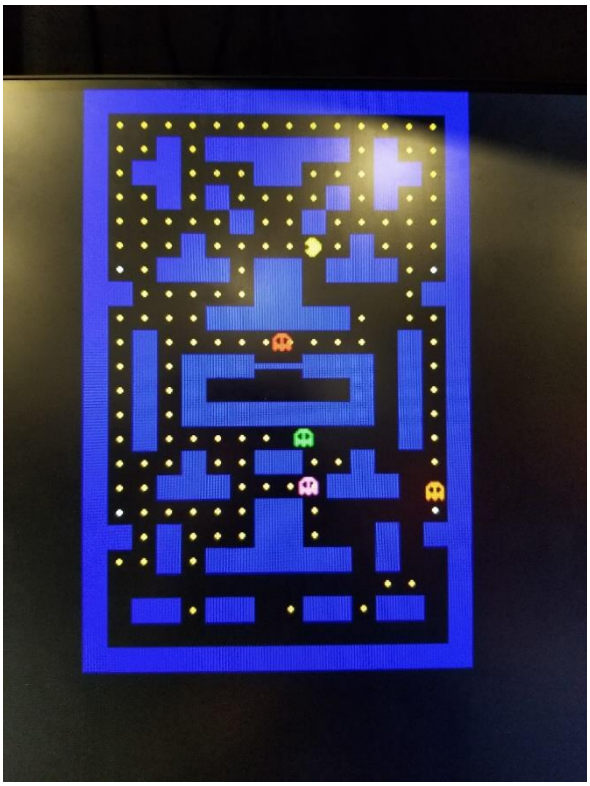

Figure 3: Mid-Game Screenshot with Four Ghosts

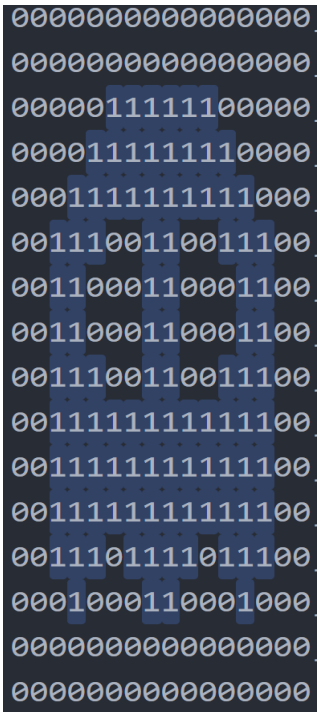

Figure 4: Sprite Drawing of Ghost

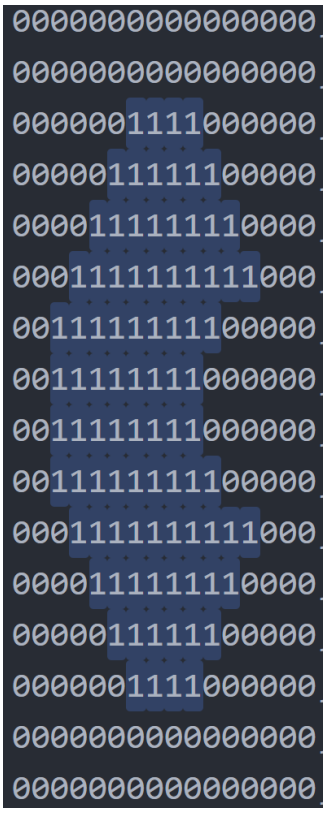

Figure 5: Sprite Drawing of Pacman (Facing Right)

Figure 6: Sprite Drawing of Condensed Maze

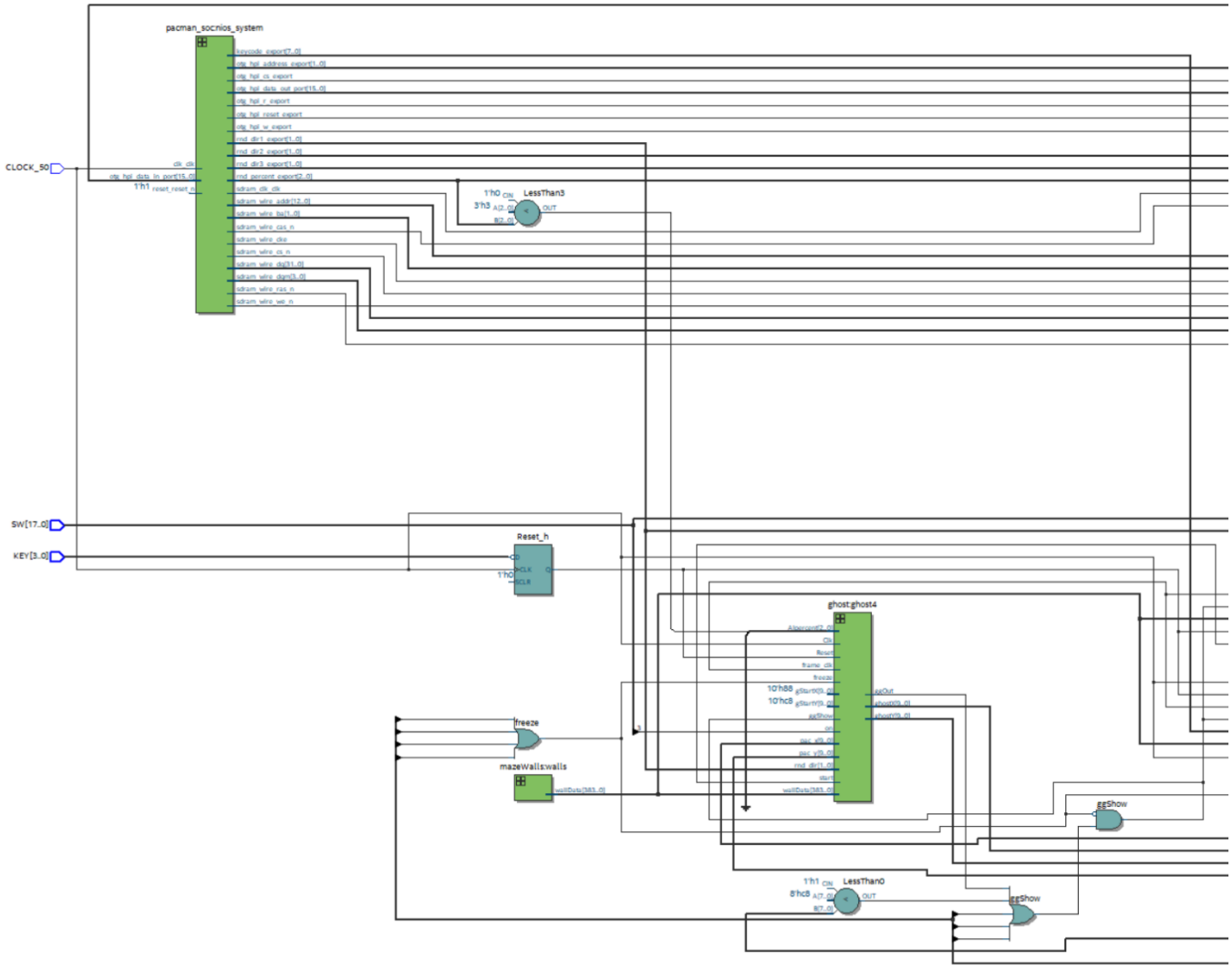

Figure 7: Left-Half of Quartus Generated Block Diagram

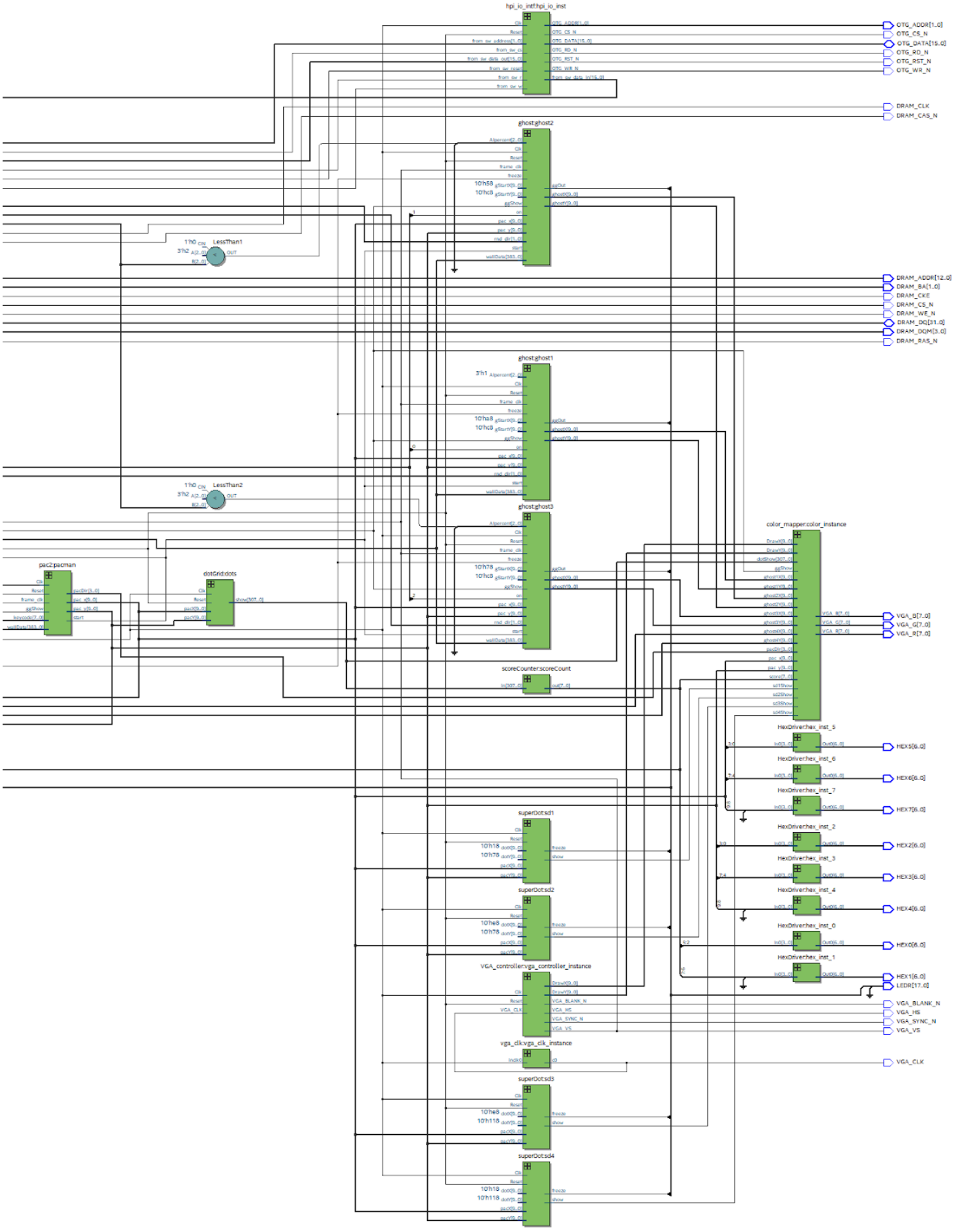

Figure 8: Right-Half of Quartus Generated Block Diagram