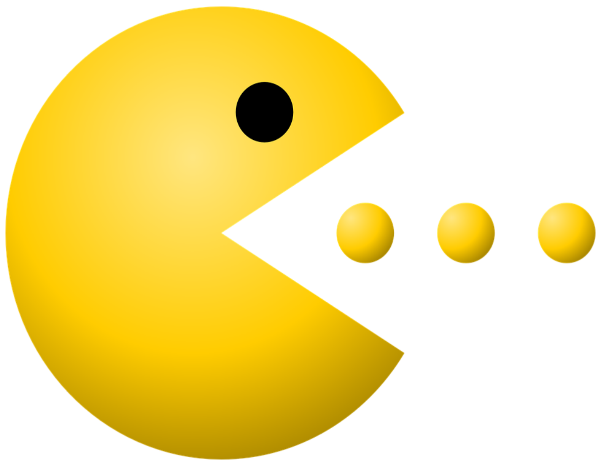
ECE 385

Spring 2018

Final Project

Final Project Report:

PAC MAN



Aditi Patange

Lohitaksh Gupta

Lab Section ABL/ Thursday 8 AM

Xinying Wang, Vibhakar Vemulapati

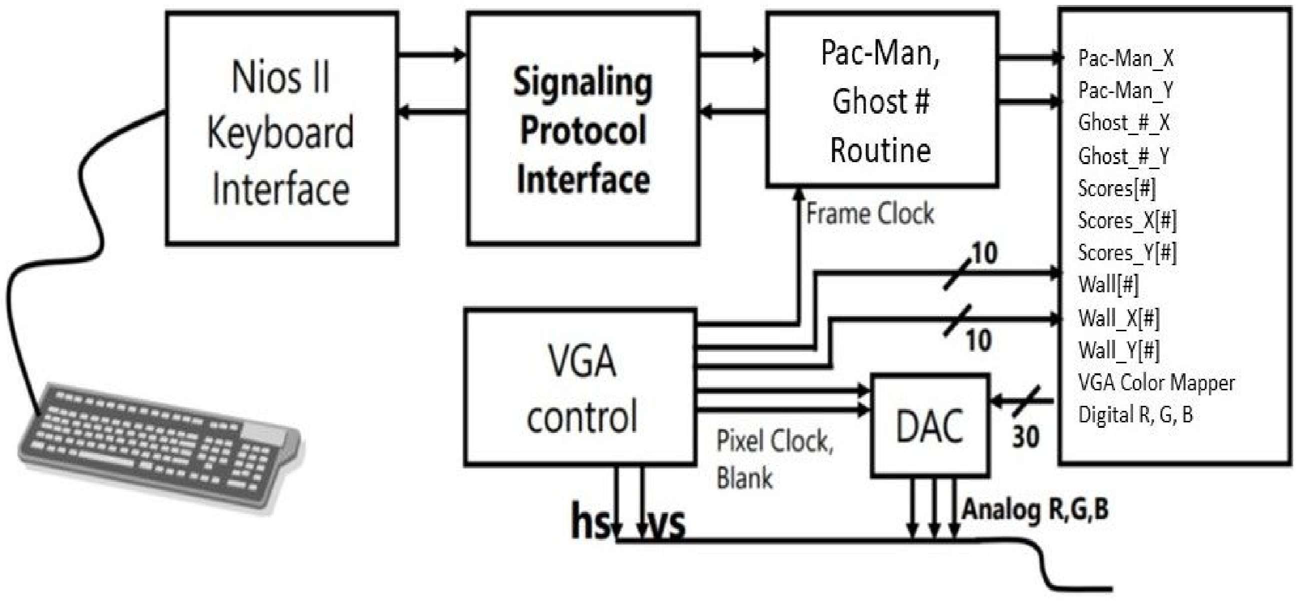
INTRODUCTION:

We design and implement a Pac-Man video game based on the existing well-known game. We implemented our Pac-Man game without the assistance of any code provided online. The entire game was implemented using System Verilog. The project relies on the integration of both software and hardware. With both, hardware and software, working together we will be able to handle inputs from the USB keyboard (as done in lab 8), on-chip memory and output to VGA. For the purpose of interfacing and integration, we will use NIOS II CPU.

Just like in lab 8, We will connect the monitor to the VGA port and the keyboard to the USB port and depending on the key pressed on the keyboard, the Pac-Man will move in the maze, either the X or Y direction on the monitor screen. Pac-Man and ghost movements are limited through multiple boundary conditions.

We used sprites to display our characters and text on the screen, giving them more fluid motion and multiple colors.

DESCRIPTION OF PAC-MAN SYSTEM:



Note: # - means ‘number of’

Block Diagram of the basic Implementation of the project.

The game is a slightly modified from the original version of Pac-Man game. The goal of the circuit is to move Pac-Man and 3 ghost in the maze. Pac-man has gone through a slight make-over and is now bigger. The three ghosts are also bigger, re-designed and of different colors. The maze is different than the original version, allowing the bigger characters to move around smoothly. In our version, Pac-Man can eat one pallet or food at a time and the score he earns for the eaten pellets is displayed on bottom center of the screen. Just like in the original game, Pac-Man must avoid the ghost on his quest to eat all the pellets on the screen or face termination. If it collides with any ghost, we stop the movement of Pac-Man and 3 ghosts. Then, 'Game Over' and final score is then displayed in the bottom. The game can be restarted by pressing the reset button on FPGA (as shown in figure 1).

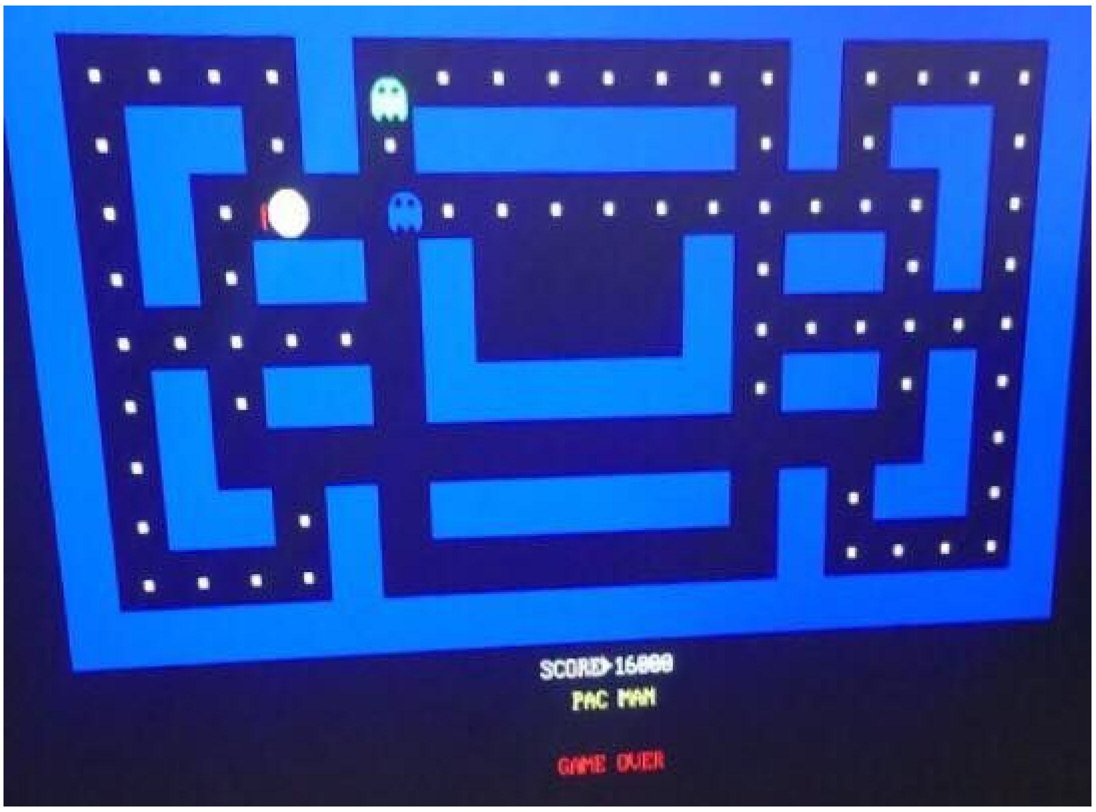


Figure 1: 'GAME OVER' is displayed in the bottom when the Pac-Man collides with the red ghost. Also, Pac-Man mouth closes on collision and the final score is also displayed at the bottom.

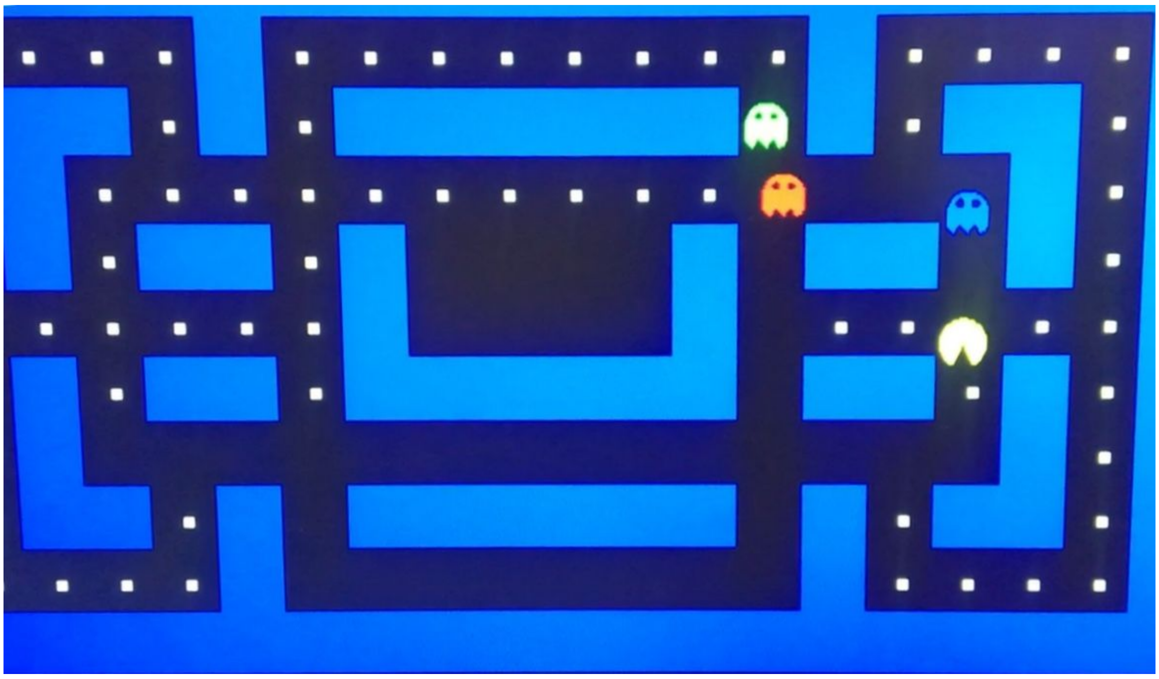


Figure 2: Pac-Man eating the food and ghost are following it based on *Artificial Intelligence.*

The Pac-Man and ghosts can either move in the X (horizontal) direction or the Y (vertical) direction. (Remember that on the monitor, Y=0 is the top and Y=479 is the bottom!)

When the program starts, a stationary yellow ball (Pac-Man with closed mouth) should be displayed in the center of the screen. The ghost are seen stationary in their resting area. The Pac-Man should be waiting for a direction signal from the keyboard. As soon as a direction key (W-A-S-D) is pressed on the keyboard, it will start moving in the direction specified by the key.

W	Up
S	Down
A	Left
D	Right

When the Pac-Man reaches the edge of the screen and left, right, up or down wall of the maze, it should stop moving. It then waits for inputs from keyboard. Also, Pac-Man continues it movement (even if user doesn’t press any key) when there is no wall in front of him.

The Pac-Man will keep moving in the same direction until either another command is received from the keyboard or it encounters a wall in the maze. When a different direction key is pressed, it should start moving in the newly specified direction (of course, if there is no immediate wall in that direction) immediately, without returning to its original position in the maze.

We are using sprites (as shown in Figure 3) to display the moving characters on the screen. Each sprite moves one pixel at a time creating a smooth movement on the screen. The sprites are twenty-six by twenty-six pixels in size, allowing 24 bits per pixel. These bits form 24-bit wide RGB colors (this is our palette). This allows us to use multiple colors for Pac-Man and the ghosts. Another advantage to using sprites is their independent movements from each other.

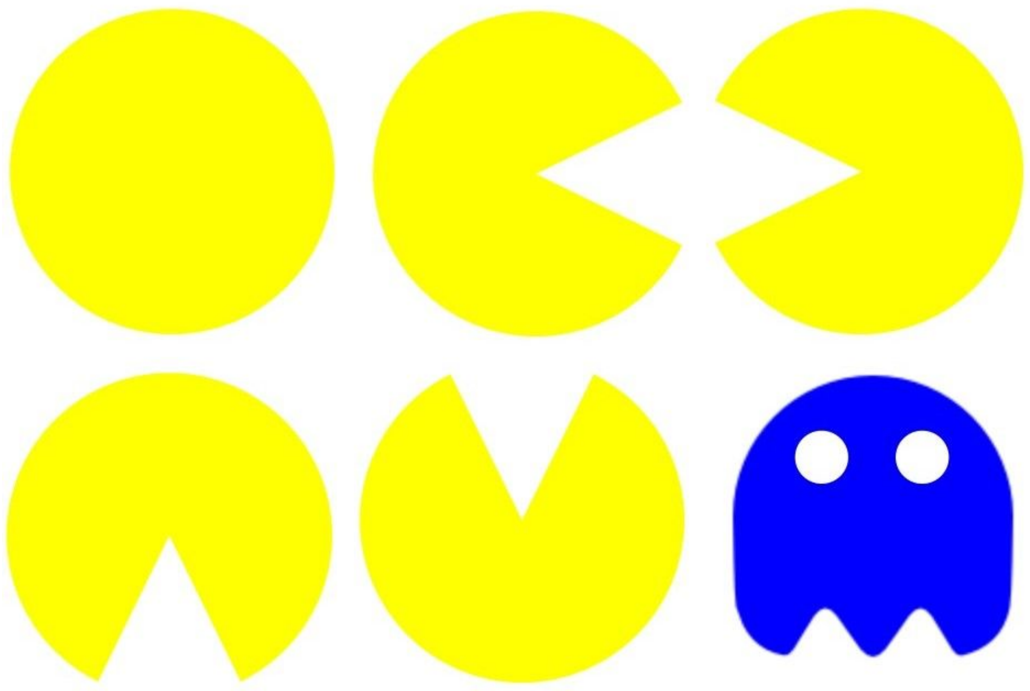


Figure 3: Sprites used in the game
Note: The colors seen in these sprites, can be easily changed in Color_Mapper module.

Each sprite is composed of 676 lines, 24 bits each. Each line corresponds to each pixel of 26 by 26 (=676) pixel square on the screen. The sprites are loaded on on-chip memory. We used the concept of ROM thought in 385 helper tools, in order to access the on-chip memory and store our sprites on it. We have a faster reading time and there is no delay in displaying of the sprite on screen, as we are using on on-chip memory.

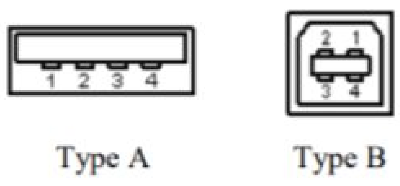
On-chip memory has the highest throughput, lowest latency memory possible in an FPGA-based embedded system. It typically has a latency of only one clock cycle. Memory transactions can be pipelined, making a throughput of one transaction per clock cycle typical.

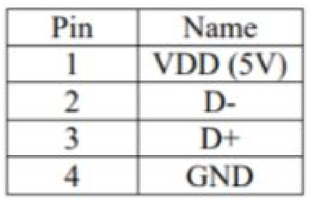
Throughout the implementation, it is important to track the time and the address currently displayed on the screen. It is important to keep track of the horizontal and vertical lines on the screen in order to synchronize loading and displaying. The function of VGA has been talked in detail in the section, '*How the VGA Monitor works.*'

How the USB Keyboard Works:

The Universal Serial Bus (USB) standard defines the connection and communication protocols between computers and electronic devices. It also provides power supply to the connected devices. Due to the compatibility with a wide variety of devices, the USB standard has become prevalent since its introduction in the 1990s.

A USB cable has either a type A or a type B port on its ends. A USB port consists of four pins: VDD, D-, D+, and GND. Figure 1 shows the configuration. The VDD and GND pins are power lines, and D- and D+ are data lines. When data is being transmitted, D- and D+ take opposite voltage levels in a single time frame to represent one bit of data. On low and full speed devices, a differential '1' is transmitted by pulling D+ high and D- low, while a differential '0' is a D- pulled high a D+ pulled low.



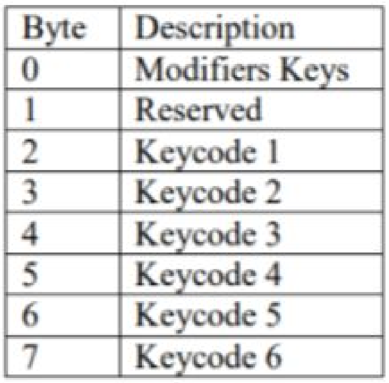


Pin	Name
1	VDD (5V)
2	D-
3	D+
4	GND

The USB port on the DE2 board is equipped with the Cypress EZ-OTG (CY7C67200) USB Controller, which handles all the data transmission via the physical USB port and manages structured information to and from the DE2 board. CY7C67200 can act as either a HostController, which controls the connected devices, or a Device Controller, which makes the DE2 board itself a USB device. In this lab, we will be using CY7C67200 as a Host Controller.

A USB keyboard is a Human Interface Device (HID). HIDs usually do not require massive data transfers at any given moment, so a low speed transmission would suffice. (Other USB devices such as a camera or a mass storage device would often need to send large files, which would require bulk transfers, a topic not covered in this lab) Unlike earlier standards such as PS/2, a USB keyboard does not send key press signals on its own. All USB devices send information only when requested by the host. In order to receive key press signals promptly, the host needs to constantly poll information from the keyboard. In this lab, after proper configuration, ISP1362 will constantly send interrupt requests to the keyboard, and the keyboard will respond with key press information in report descriptors. A descriptor simply means a data structure in which the information is stored.

Table 1 shows the keyboard input report format (8 bytes). In this format, a maximum of 6 simultaneous key presses can be handled, but here we will assume only one key is pressed at a time, which means we only need to look at the first key code. Each key code is an 8-bit hex number. For example, the character A is represented by 0x04, B by 0x05, and so on. When the key is not pressed, or is released, the key code will be 0x00 (No Event).



Byte	Description
0	Modifiers Keys
1	Reserved
2	Keycode 1
3	Keycode 2
4	Keycode 3
5	Keycode 4
6	Keycode 5
7	Keycode 6

Table 1

A more detailed explanation of how the keyboard works (and all the key code combinations) can be found in the INTRODUCTION TO USB AND EZ-OTG ON NIOS II (IUQ). In particular, the USB 2.0 Specification and the HID Device Class Definition (refer to the ECE 385 course website) are two documents that define all the behavior of a USB keyboard.

How the VGA Monitor works:

VGA (Video Graphics Array) Standard: The screen is organized as a matrix of 640 horizontal x 480 vertical lines. An Electron Beam “paints” each pixel from left to right in each row, and each row from top to bottom. Screen refresh rate is 60 Hz, which means that we must generate VGA signal 60 times a second. In this lab, we used a simple color mapper combined with the VGA controller to draw simple shapes. Color mapper needs to have as inputs the horizontal and vertical position counters, and maps output color either to foreground color (e.g. red) or background color (e.g. white). This is shown in figure 2

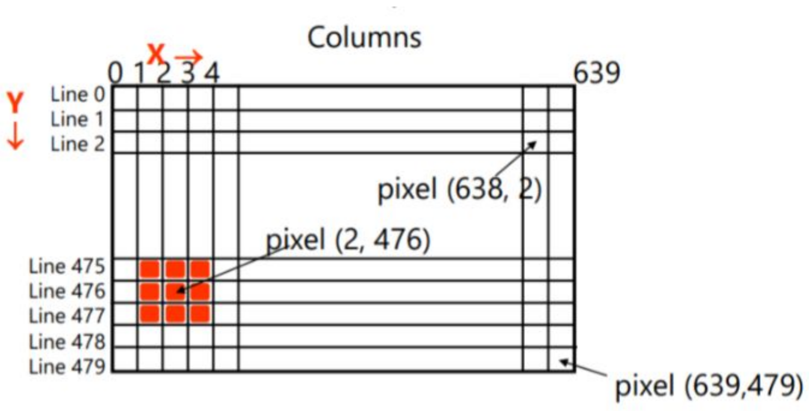


Figure 2

For detailed explanation on how the VGA monitor works, please refer to the lecture slides and section 4.10 of the DE2-115 User Guide available on the ECE 385 website.

Description of USB Protocol:

In Lab 8, the USB protocol is handled in the software on the Nios II, and the extracted keycode from the USB keyboard is then sent to the hardware for further use. We build a Qsys interface to handle to I/O interface to communicate with the USB-OTG chip. Then we import existing NIOS II software and make changes to it to get the keyboard to work correctly.

The Cypress EZ-OTG (CY7C67200) chip handles the USB protocol. OTG stands for On-TheGo, which is an add-on specification on top of the standard USB 2.0 standard that allows not only PCs but also other mobile devices to act as a USB Host. The EZ-OTG chip itself contains a RISC microprocessor (CY16), RAM, and ROM (BIOS), as shown on the left in Figure 3. The RAM can be accessed through direct memory access (DMA) at addresses 0x0000 – 0x3FFF. The Serial Interface Engines (SIEs) are the front end of the USB controller and are where the USB ports are attached. The connections between EZ-OTG and the DE2-115 FPGA board are made through the Host Port Interface (HPI), which are shown as the connections in the center of Figure 9. The noteworthy pins are HPI_D[15:0], which is the 16-bit parallel data bus; HPI_INT, which indicates the event of interrupt; HPI_A[1:0], which indicates the addresses to four port registers on the chip. Each of the port registers control some important functionalities on EZ-OTG, some of which will be used later. The port registers and the addresses are listed in Table 2.

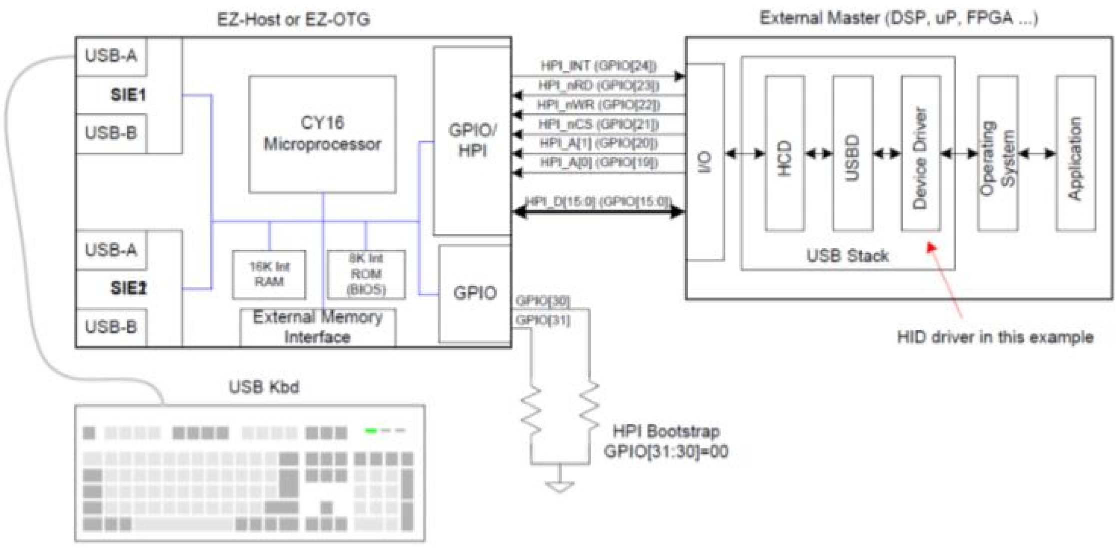


Figure 3

Port Registers	HPI A [1]	HPI A [0]	Access
HPI DATA	0	0	RW
HPI MAILBOX	0	1	RW
HPI ADDRESS	1	0	W
HPI STATUS	1	1	R

Table 2

HPI has only 4 hardware addresses (ADDR[1], ADDR[0]) to reduce pin-count. IO_read()/IO_write() reads/writes into these 4 addresses, it interfaces with PIO, just like what we did in Lab 7. USBRead()/USBWrite() reads/writes into full EZ-OTG memory space. It has multiple calls to IO_read()/IO_write(). This is shown in Figure 4.

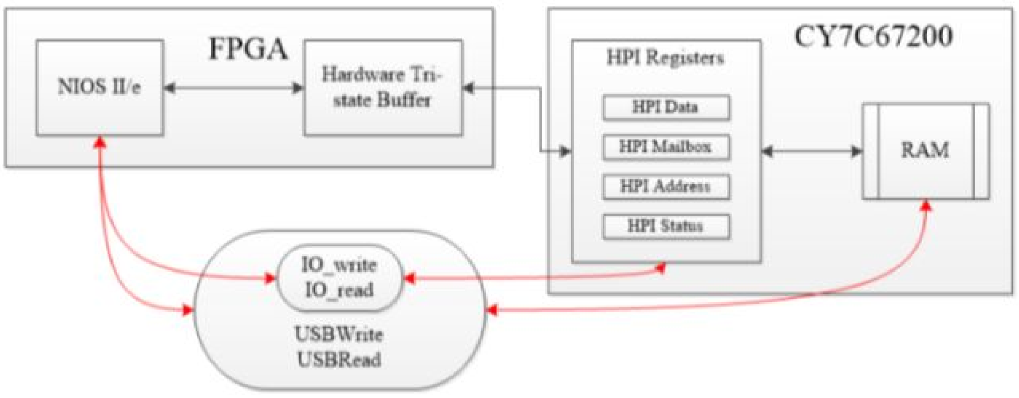


Figure 4

OPERATIONS:

USB Read Operation:

USB read operation is defined as follows (from BIOS manual under HPI):

First set address register (0b10) to address (in EZ-OTG memory space).

Read data out of data register (0b00).

Address may auto increment (therefore, we will sometimes see multiple IO_read()following a USBRead().

USB Write Operation:

USB write operation is defined as follows (from BIOS manual under HPI):

First set address register (0b10) to address (in EZ-OTG memory space)

Set data register (0b00) with data to be written.

Address may auto increment (therefore, you will sometimes see multiple IO_write()following a USBWrite().

IO Read:

IO Read timing is similar to SRAM from Lab 6.

Note: Like SRAM, DATA pins are bidirectional (we will need some type of tristate buffer, either in/out or use old tristate.SV and control both ports via PIOs).

Access time is much faster than CPU clock cycle (6ns) so reads can happen in one cycle.

IO Write:

Write timing is similar to asynchronous SRAM as well.

Note: We need to toggle WR (write enable) through PIOs.

CPU is not able to operate at full HPI speed, irrelevant because keyboard has low data rate (might consider making a faster interface if using other USB profiles).

List of Features:

Features implemented for the project:

- Main character Pac-Man should move up, down, right and left on the screen based on the keyboard input.
- Pac-Man and ghost characters should not pass through the walls.
- Pac-Man should die if a ghost character intersects with it.
- Basic graphics which show the correct shape of Pac-Man, ghost characters, score and wall.
- The scoreboard will be displayed on the VGA monitor.
- The ghost characters should move on their own based on some advanced AI. Concepts like Manhattan distance were used in order to implement the AI movements of ghost.
- Regeneration of Pac-Man if it collides with a ghost character.
- Pac-Man can also eat ghost characters.
- Developing an intuitive and more appealing UI with better and advanced graphics

Timeline followed:

Project Week 1: 4/2/2018 - 4/8/2018:

- Finalize the project by discussing with the TA
- Research on course website for understanding the intricacies of the project
- Collect the resources from the course website useful for our project

Project Week 2: 4/9/2018 - 4/15/2018:

- Make a detailed block diagram.
- Understand the driver code and other basic I/Os
- Make sure that everything from lab 8 is working and we can build our project on top of it

- Discuss the doubts and difficulties with the TA on Thursday

Project Week 3: 4/16/2018 - 4/22/2018:

- Complete the leftovers from week 2
- Implement the basic graphics through our sprite
- Show the random moving of pac-man and ghost characters
- Prepare for mid checkpoint.
- DEMO developments on Thursday (4/19/2018)
- Discuss the doubts and difficulties with the TA on Thursday

Project Week 4: 4/23/2018 - 4/29/2018:

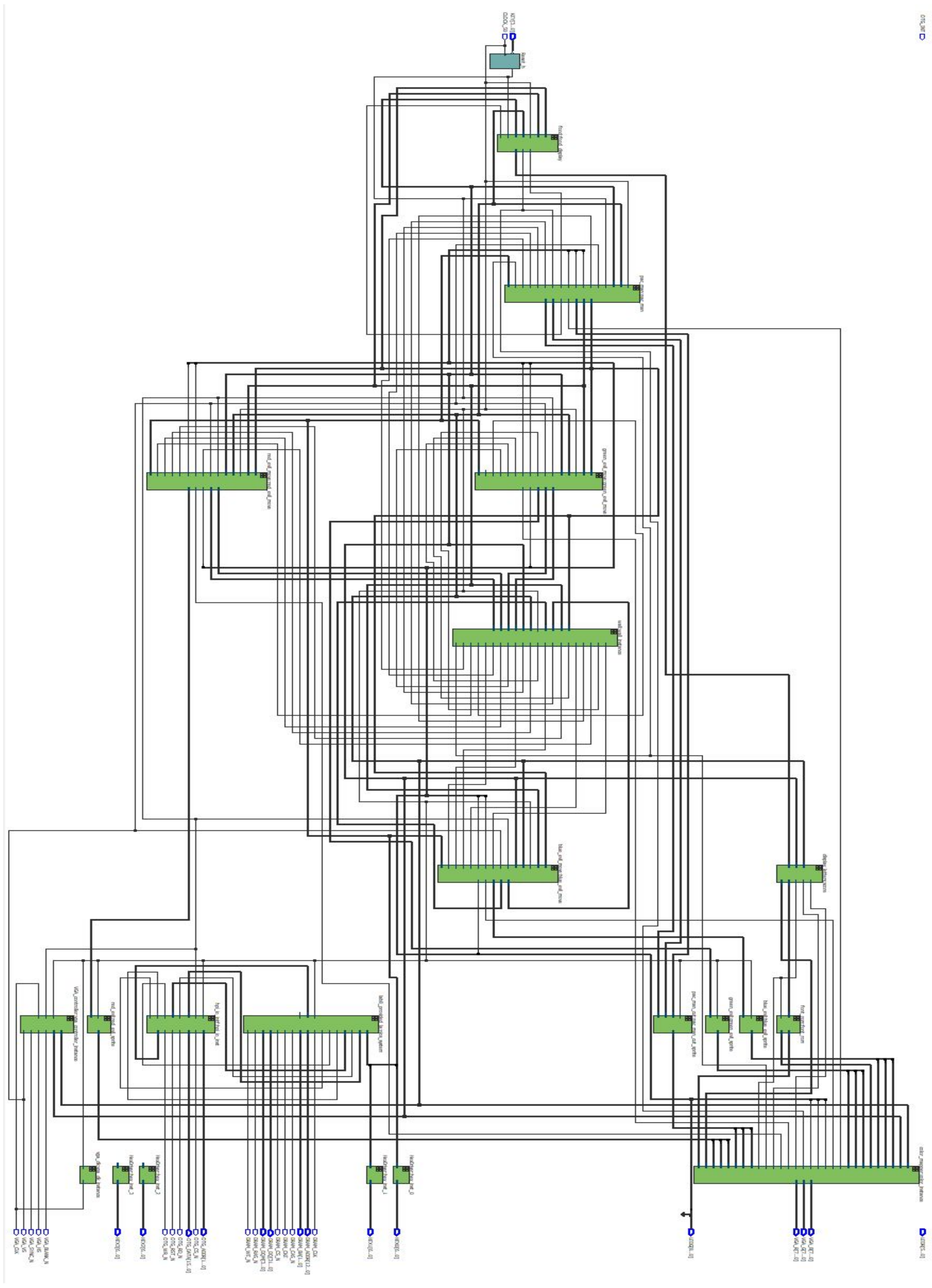
- Complete the leftovers from week 3
- Put the walls and limit the movement of pac-man and ghost characters accordingly.
- Implement a primitive AI in order to autonomously control the movement of ghost characters.
- The collision of any ghost and pac-man should stop the game and display ‘Game Over’
- Discuss the doubts and difficulties with the TA on Thursday

Project Week 5: 4/30/2018 - 5/03/2018:

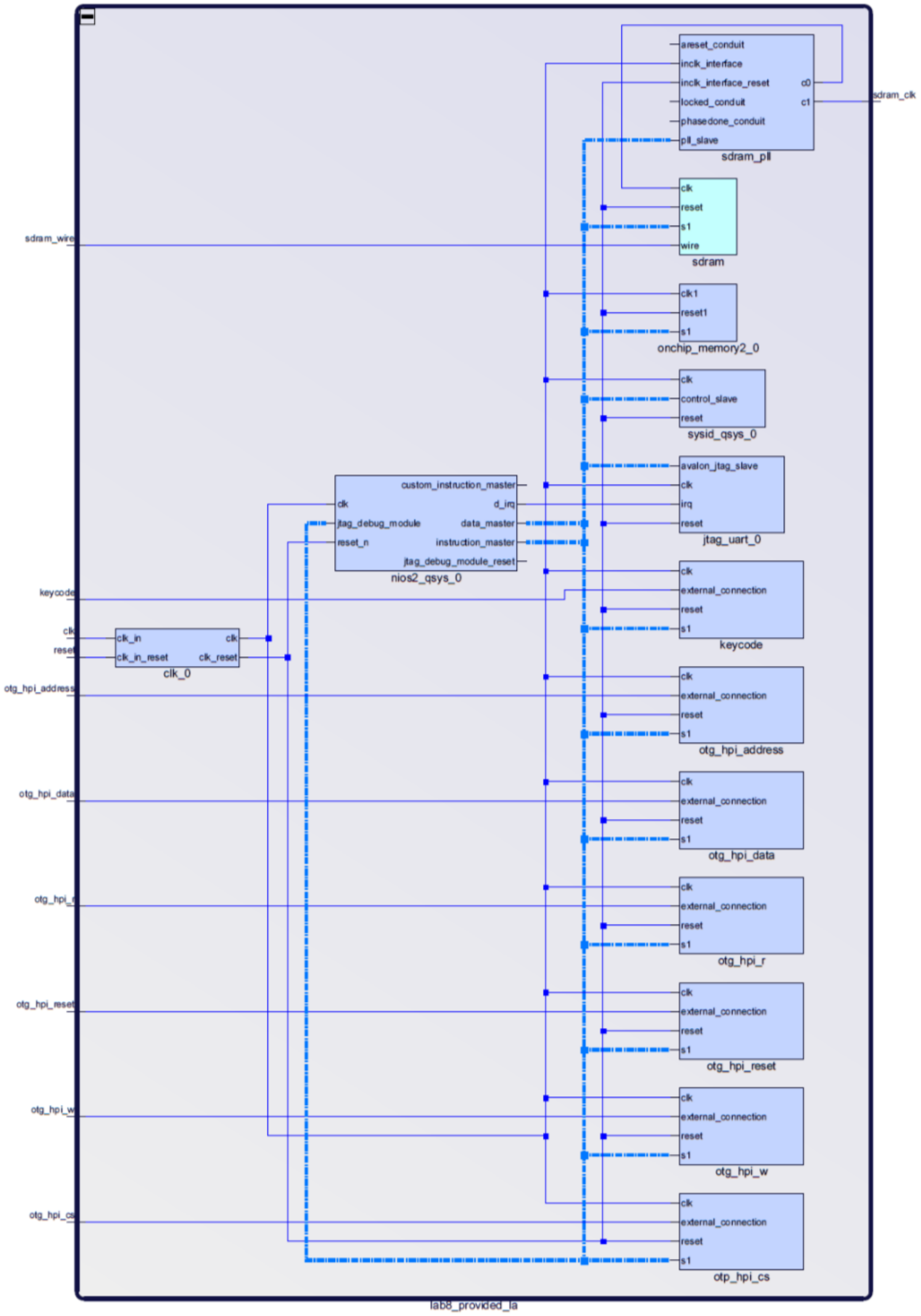
- Compete the leftovers from week 4
- Work for difficulty points
- Primary focus:
    - Implement audio in accordance with the graphics
    - Make the AI for ghost characters more advance
    - Regeneration of pac-man and death of ghost characters
- Prepare for Demo
- Demo on Thursday (5/03/2018)
- Work on report
- Go home :D

PLEASE SCROLL DOWN FOR BLOCK DIAGRAMS

BLOCK DIAGRAM:



Top Level RTL View of PAC-MAN Game (use the link for higher resolution)
https://drive.google.com/open?id=1nOf73LPrfOJnD3hc7zKADfGyU3Me4nEa



QSYS View of the NIOS II Processor

DESCRIPTION OF ALL .SV MODULES:

1) Module: lab8.sv
Inputs: CLOCK_50, [3:0] KEY, OTG_INT
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, [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
Inout: [15:0] OTG_DATA, [31:0] DRAM_DQ
Description: This file is the top-level entity in this project.
Purpose: This module integrates the NIOS II system with the rest of the hardware. We made connections between our hardware system created by Qsys and the other modules in this file. This file is used to create an instance of the hardware system as described in the lab8_provided_la.v file. It also creates instances of hpi_io_intf, vga_clk, VGA_controller, ball, color_mapper and HexDrivers.

2) Module: lab8_provided_la.sv
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
Inout: [31:0] sdram_wire_dq
Description: This is a Qsys generated file describing the hardware system designed in Qsys. Note: This is a Verilog file and not a SystemVerilog file. However, we can still use it because Verilog and SystemVerilog are compatible with each other.
Purpose: This file is used to call an instance of jtag_uart_0, keycode, nios2_qsys_0, onchip_memory2_0, otg_hpi_address, otg_hpi_data, otg_hpi_r, otg_hpi_reset, otg_hpi_w, otp_hpi_cs, sdram, sdram_pll, sysid_qsys_0 and irq_mapper. These are all the required modules in our hardware. All the connections between these modules are also made in this file.

3) Module: 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
Inout: [15:0] OTG_DATA
Description: In this file, we read from the USB buffer or write to the USB buffer based on the different input control signals.
Purpose: This module serves as an interface between NIOS II system and EZ-OTG chip.

4) Module: VGA_controller.sv
Inputs: Clk, Reset, VGA_CLK
Outputs: VGA_HS, VGA_VS, VGA_BLANK_N, VGA_SYNC_N, [9:0] DrawX, [9:0] DrawY
Description: The VGA control signals are assigned at the positive edge of VGA_CLK. Horizontal and vertical counters are set and always updated to ensure a clean output waveform. Pixels are displayed in the predetermined range of the monitor.
Purpose: The purpose of this file is to print the display on the VGA monitor. It prints the ball and the background at all times.

5) Module: HexDriver.sv
Inputs: [3:0] In0
Outputs: [6:0] Out0
Description: This is the same HexDriver file as used in the previous lab 6.
Purpose: It is used for displaying the keycode on the 7-segment LEDs of the FPGA. Data becomes more readable when it is displayed on a 7-segment LED as it is displayed in hexadecimal not binary.

6) Module: color_mapper.sv
Inputs: is_ball, [9:0] DrawX, [9:0] DrawY,
is_ball, is_wall, is_red_evil, is_green_evil, is_blue_evil, is_food, is_score_all_letters, is_zoom, is_game_over,// Whether current pixel belongs to ball, is_collision_blue, is_collision_red, is_collision_green,
[7:0] text_data, [15:0] data_16, [10:0] score_x,
[7:0] pac_man_cut_data_out_R, [7:0] pac_man_cut_data_out_G, [7:0] pac_man_cut_data_out_B, [7:0] red_evil_data_out_R, [7:0] red_evil_data_out_G, [7:0] red_evil_data_out_B, [7:0] green_evil_data_out_R, [7:0] green_evil_data_out_G, [7:0] green_evil_data_out_B, [7:0] blue_evil_data_out_R, [7:0] blue_evil_data_out_G, [7:0] blue_evil_data_out_B
Outputs: [7:0] VGA_R, [7:0] VGA_G, [7:0] VGA_B
Description: This file considers one pixel at a time and decides the color of that pixel based on whether the element (Pac-Man, red ghost, blue ghost,

green ghost, font characters, walls of maze and food) is present at that pixel during that time. The color is defined using the RGB scheme. For example, if a pixel has the Pac-Man, it is assigned the colors accordingly. If a pixel doesn't have the ball, it is assigned a background color.
Purpose: This file is required to instruct the VGA monitor about which color to print for each pixel. It specifies the color of all pixels on the VGA monitor.

7) Module: ball.sv
Inputs: Clk, Reset, frame_clk, [9:0] DrawX, [9:0] DrawY, [7:0] key, is_wall, is_wall_up, is_wall_down, is_wall_right, is_wall_left, is_food, is_food_big_no_color, inside_block, is_collision_red, is_collision_blue, is_collision_green
Outputs: is_ball, is_food_eaten, [9:0] pac_man_cut_read_address, [9:0] pac_man_full_read_address_special, [9:0] Ball_X_Pos_out, [9:0] Ball_Y_Pos_out, [2:0] direction
Description: This file decides the position of the Pac-Man at any given time. The computation for the location and movement of it at all times is done in this module. The direction of movement of it is dependent on the key input (i.e. the key pressed).
Purpose: This file is necessary to determine the location of the Pac-Man at any time. The output of this module, is_ball, is used by the Color_mapper module to determine the color of a pixel. is_ball signal determines whether a pixel corresponds to ball or background. Is_food_eaten tell the food module whether it has been eaten or not by the Pac-Man. Ball_X/Y_Pos_out are the X and Y coordinated of the top left corner of the Pac-Man. direction output is used by the ghost and sprite modules to decide the new direction for ghost movement and direction of the mouth of Pac-Man respectively.

8) red_evil.sv
Inputs: Clk, Reset, frame_clk, is_wall_up_red, is_wall_down_red, is_wall_right_red, is_wall_left_red, inside_block_red, is_collision_green, is_collision_blue, [9:0] DrawX, [9:0] DrawY, [7:0] key, [9:0] Ball_X_Pos_out, [9:0] Ball_Y_Pos_out
Outputs: [9:0] red_evil_read_address, [9:0] Red_X_Pos_out, [9:0] Red_Y_Pos_out, is_red_evil, is_collision_red
Description: The file decided the position of red ghost at any given time. The computation for the location and movement of it at all times is done in this module.This file is based on ball.sv. Therefore, it has similar functionality, except for the fact that movement is controlled through artificial intelligence.
Purpose: This file is necessary to determine the location of the red ghost at any time. The output of this module, is_red_evil, is used by the Color_mapper module to determine the color of a pixel. Is_red_evil signal determines

whether a pixel corresponds to ball or background. Is_collision_red tells the the remaining ghost modules and Pac-Man module that game is over and movements of everything should be stopped. Also, it tells Color_Mapper whether to display ‘GAME OVER’ or not.

9) blue_evil.sv
Inputs: Clk, Reset, frame_clk, is_wall_up_blue, is_wall_down_blue, is_wall_right_blue, is_wall_left_blue, inside_block_blue, is_collision_green, is_collision_red, [9:0] DrawX, [9:0] DrawY, [7:0] key, [9:0] Ball_X_Pos_out, [9:0] Ball_Y_Pos_out
Outputs: [9:0] blue_evil_read_address, [9:0] Blue_X_Pos_out, [9:0] Blue_Y_Pos_out, is_blue_evil, is_collision_blue
Description: The file decided the position of blue ghost at any given time. The computation for the location and movement of it at all times is done in this module.This file is based on ball.sv. Therefore, it has similar functionality, except for the fact that movement is controlled through artificial intelligence.
Purpose: This file is necessary to determine the location of the blue ghost at any time. The output of this module, is_blue_evil, is used by the Color_mapper module to determine the color of a pixel. Is_blue_evil signal determines whether a pixel corresponds to ball or background. Is_collision_blue tells the the remaining ghost modules and Pac-Man module that game is over and movements of everything should be stopped. Also, it tells Color_Mapper whether to display ‘GAME OVER’ or not.

10) green_evil.sv
Inputs: Clk, Reset, frame_clk, is_wall_up_green, is_wall_down_green, is_wall_right_green, is_wall_left_green, inside_block_green, is_collision_red, is_collision_blue, [9:0] DrawX, [9:0] DrawY, [7:0] key, [9:0] Ball_X_Pos_out, [9:0] Ball_Y_Pos_out
Outputs: [9:0] green_evil_read_address, [9:0] Green_X_Pos_out, [9:0] Green_Y_Pos_out, is_green_evil, is_collision_green
Description: The file decided the position of green ghost at any given time. The computation for the location and movement of it at all times is done in this module.This file is based on ball.sv. Therefore, it has similar functionality, except for the fact that movement is controlled through artificial intelligence.
Purpose: This file is necessary to determine the location of the green ghost at any time. The output of this module, is_green_evil, is used by the Color_mapper module to determine the color of a pixel. Is_green_evil signal determines whether a pixel corresponds to ball or background. Is_collision_green tells the the remaining ghost modules and Pac-Man module that game is over and movements of everything should be stopped. Also, it tells Color_Mapper whether to display ‘GAME OVER’ or not.

11) font_rom.sv
Inputs: [10:0] addr, is_zoom
Outputs: [7:0] data, [15:0] data_16
Description: Contains the sprite of all the text characters. The input is an 11 bit address. The output is an 8 bit chunk of data. As described before, the row number is input as the address, and the entire row is outputted. By looking at the sprite table, it should become apparent that each symbol takes up 16 rows in the ROM. The general formula for where a symbol resides in the font sprite table would be:

Starting address: 16*n
Endaddress: (16*n)+15

Where 16 and 15 are in decimal, but n corresponds to the hex code in the table. This way we traverse the font sprite table.
Purpose: Using font_rom.sv, we can easily display fonts on the screen as bits are logically assigned pixel by pixel.

12) display_letters.sv
Inputs: [9:0] DrawX, [9:0] DrawY, [7:0] score
Outputs: is_score_all_letters, is_zoom, is_game_over, [10:0] score_addr, [10:0] score_x
Description: In this module, is_score_all_letters is used to decide which pixels are to be used for displaying the score. is_zoom is used to decide which pixels are to be used for displaying the 'PAC-MAN'. Similarly, is_game_over is used to decide which pixels are to be used for displaying the 'GAME OVER'. All these signals are sent to Color_Mapper module so as to assign respective colors for each signal. In addition to that score_addr tells the font_rom.sv that which 8-bit address needs to be accessed now. score_x is used by Color_Mapper module to decide which bit in the 8-bit address is 1 or 0. The bit that are high get colored.
Purpose: This module decides respectives location of the letters and which letter's bit to print at which pixel of the screen.

13) food.sv
Inputs: Clk, Reset, [9:0] DrawX, [9:0] DrawY, [9:0] Ball_X_Pos_out, [9:0] Ball_Y_Pos_out, is_food_eaten
Outputs: is_food, [7:0] score_out
Description: is_food is used to assign food pixels in the maze. It is done using the DrawX and DrawY. The module knows whether the food has been eaten or not through is_food_eaten. If the food is eaten, the score_out is incremented and then sent as an output to display_letters.sv, so that it can be displayed in the bottom of the screen

Purpose: This module assigns food pixels for Pac-Man. It also calculates the score.

14) sprites.sv
Inputs: Clk, [2:0] direction, [9:0] pac_man_cut_read_address, [9:0] pac_man_full_read_address_special
Outputs: [23:0] pac_man_cut_data_out
Description: The modules in this file are based on ram.sv file available in the 385 developer tools on the course website. For reading the Pac-Man sprite, pac_man_cut_read_address and pac_man_full_read_address_special are set in ball.sv. It is given by:

Pac_man_cut_read_address = DistY*Pac_Size + DistX
Pac_man_full_read_address_special = (Pac_Size - DistY)*Pac_Size + (Pac_Size - DistX)

Note: Pac_man_full_read_address_special helps us in saving memory. It basically displays a flipped image of any sprite.
Pac_man_cut_data_out contains the 24-bit RGB value for each pixel of the respective sprite.
Purpose: Helps in displaying the images of Pac-Man (mouth close and mouth open - up, down, left, right), red ghost, green ghost and blue ghost.

15) wall.sv
Inputs: Clk, [9:0] DrawX, [9:0] DrawY, [9:0] Ball_X_Pos_out, [9:0] Ball_Y_Pos_out, [9:0] Blue_X_Pos_out, [9:0] Blue_Y_Pos_out, [9:0] Green_X_Pos_out, [9:0] Green_Y_Pos_out, [9:0] Red_X_Pos_out, [9:0] Red_Y_Pos_out,
Outputs: is_wall, is_wall_up, is_wall_down, is_wall_right, is_wall_left, inside_block, is_wall_up_blue, is_wall_down_blue, is_wall_right_blue, is_wall_left_blue, inside_block_blue, is_wall_up_red, is_wall_down_red, is_wall_right_red, is_wall_left_red, inside_block_red, is_wall_up_green, is_wall_down_green, is_wall_right_green, is_wall_left green, inside_block_green
Description: In this file, we declare and initialize a 20 * 15 register array. This register array stores the map of the walls. Each location (element) in the array corresponds to a 32 * 32 pixel block on the screen and if the element stored at a current location is 1 then the corresponding block is a wall and vice versa. This array is never edited in the execution of the game. Additionally, the location of the pacman, blue evil, red evil, green evil is sent as an input to this module. After checking the register array, it outputs 1 if the location is a wall and 0 otherwise.
Purpose: This file is required to generate a maze similar to the original PacMan game. It is used by the color_mapper module to print the maze at

appropriate locations. The movements of the PacMan and evils is highly dependent on and constraint by the outputs of this module. In order to check that none of the PacMan or evils move through a wall, this file tells each of them if there is a wall above, below, to the right or to the left of their current location.

DESIGN RESOURCES AND STATISTICS TABLE

LUT	5707
DSP	0
Memory (BRAM)	138,240 / 3,981,312 (3 %)
Flip-Flop	2,408 / 117,053 (2 %)
Frequency	110.47 MHz (altera_reserved_tck)
Static Power	105.36 mW
Dynamic Power	0.95 mW
Total Power	179.11 mW

Resources:

- Font file and Font Sprite Tutorial by Daniel Chen
- Rishi's 385 Helper Tools https://github.com/Atrifex/ECE385-HelperTools
- 385 Course Website - Final Project https://wiki.illinois.edu/wiki/display/ece385sp18/Final+Project
- 385 Course Website - Lab 8 https://wiki.illinois.edu/wiki/display/ece385sp18/Lab+8
- 385 Course Website - Sprite Drawing (Lecture 19) https://wiki.illinois.edu/wiki/display/ece385sp18/Lectures

Problems Encountered:

- After running image to text file python scripts (given in Rishi's Helper Tools), we got lot of garbage values in our converted text files. This gave us really low quality images. In order to fix this, we wrote our own python scripts. This gave us really enhanced images in the VGA display finally.

- Once we had our PacMan and evils moving on the constrained path (i.e. non-wall area of the maze), we had a few glitches. In some rare cases, our PacMan and evil and would move through wall and continue doing so until we hit Reset. After a lot of testing and brainstorming, we found out that there was a issue with our timing. The is_wall values were supplied to PacMan and evil one clock cycle late. Once we found out the issue, we were able to fix it easily and the rest of the logic was flawless.

CONCLUSION

The project was completed successfully. We were also able to deploy advanced feature beyond our basic features specified in the proposal. We were very happy after we implemented AI on our evil and it worked perfectly! The game works well and has nice character graphics. We never imagined that creating Pac-Man would be so hard and time consuming. Dealing with the boundary conditions, sprites and limitations of on-chip memory were the toughest part. After this project, we feel really confident about programming in SystemVerilog. We really liked the System on Chip part of the class.

Special thanks to Professor Zuofu Cheng, TA Xinying Wang and Kristi for helping us dealing with the complexities of this class.