# ECE 385

Spring 2017

Final Project

# Report

Hongyi Li, Qianwei Li

ABD Tuesday 6-8:50pm

Aakash Modi

# Introduction

For our final project, we make a Pacman game by using the FPGA board and SystemVerilog. In this game, the Pacman is able to win the game by eating all the beans without being caught by ghosts. To do this, we go through five steps in total.

Firstly, we inherit our files for Lab 8 as our starting point. This is because our final project is similar to Lab 8 during initial stages. For example, they all require keyboard interfaces. In addition, we need to design motion logic in both of them. In Lab 8, the ball needs to move according to keys pressed so that it can change directions with different keys and bounce at walls. In our final project, the Pacman needs to change directions when different keys are pressed and stop when it hits walls. In addition, it should be able to make turns according to old keys pressed. That is to say, if Pacman isn't able to turn left when the corresponding key is pressed because it's blocked by walls, it should turn left whenever it's not impeded by left walls anymore. In this way, users can have more convenience because they don't need to hold a key constantly when they want to change directions. Although the motion logic for our final project is more complicated than Lab 8, we can still adapt the overall code structure in Lab 8.

Secondly, we design the motion logic for Pacman. Our strategy is to first check if there are walls in the directions of current motions. If there are, Pacman will stop and stay still immediately. We then will check if any valid keys are pressed. If direction keys are pressed, we first check if there are walls in the directions represented by the keys. If so, we just let Pacman keep its original motions. Otherwise, we will allow Pacman to change directions according to the pressed keys. If no valid direction keys are pressed, we still first check if there are walls in current motion directions. If so, Pacman stops immediately. Otherwise, we continue checking if Pacman can change into directions represented by old keys, which are pressed during the last or the last few cycles. (This means there are no blocking walls and Pacman is at right positions.) If Pacman cannot change directions according to old keys, it will just keep original motions.

Thirdly, we design the background that is a maze, in which Pacman will move around. We accomplish this by hardcoding the maze structure that we design. This will be discussed later in more details.

Fourth, we draw out the ghost images and design their motion logic. After drawing out images and converting them to RAM files, we give the three ghosts different starting locations. At every clock cycle, ghosts will always analyze if there are walls on their top, bottom, left and right. Among the directions without walls, they will follow the directions that are closest to Pacman. To make their paths different and the game more fun to play, one ghost will track Pacman itself. Another will follow the new horizontal position of Pacman, while the last one will track the new vertical position of Pacman. In

this way, Pacman will sometimes be surrounded by ghosts in three different directions, which makes it harder to escape.

Lastly, we use the font and sprite files online to construct score and life keeping tables. To be more specific, scores will update every frame clock cycle, while Pacman has two lives, which means players have two chances to eat all beans and win the game. If they lose the first live, eaten beans will come back and they have to start from the scratch.

Design details about these aspects will be revealed in later sections of this report.

# Game Rules and Features

Here is a list of rules and features about the game.

1.Users control Pacman via the keyboard. “W” represents moving up. “S” represents moving down. “A” represents moving to the left. “D” represents moving to the right.

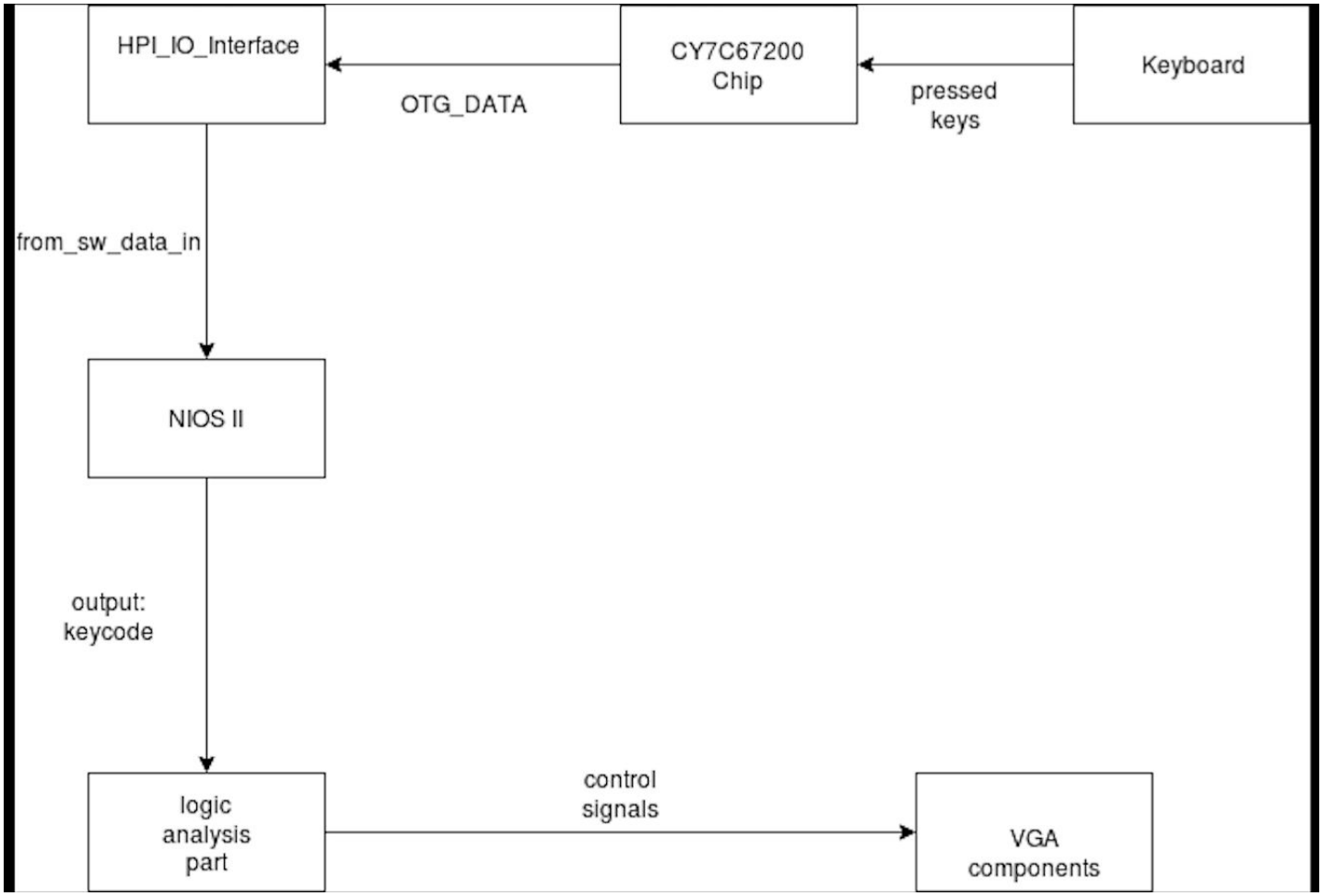
2.When the game starts, ghosts won’t move from their starting positions until Pacman starts to move.

3.There are 2 kinds of beans. One kind is red and smaller. When Pacman eats one of them, it will get 1 point, which will be shown by the score. The other sort is in green and larger. When Pacman eats a special bean like this, it will gain three points. In addition, it will become able to attack ghosts for 8 seconds. Originally, the three ghosts are in red, green and pink respectively. After the special bean is eaten, they will all become blue for the following 8 seconds. The blue color indicates that they are vulnerable to Pacman during the time periods. Also, during the special time periods, they will always move along paths that will make them further away from Pacman. If they are eaten by Pacman, they will just return to their positions at the beginning of the game. After 8 seconds, the ghosts return to original colors and Pacman is no longer able to attack them.

4.Scores are recorded in both hex displays and on the screen. 1 point will be gained if a normal bean is eaten, 3 points will be gained if a special bean is eaten and 5 points will be gained if a ghost is eaten. Life chances are also recorded on the screen by two red hearts. When users initially start the game, they will have two chances, represented by the hearts. If Pacman gets caught by ghosts, one heart will be dropped and the game will start all over again. If Pacman is caught by ghosts again, the remaining heart will disappear, which means that users lose the game. In this way, people can always know how many chances they have left.

# Overall Architecture

Generally, we organize the structure of our machine in this way. The keyboard will provide necessary information of pressed keys to the logic analysis part in FPGA so that Pacman and ghosts can move in right ways. However, keyboard does not transmit information directly to FPGA. Instead, the USB chip (CY7C67200) will first accept the information of keys pressed by its Serial Interface Engines (SIEs). It will then transmit the data to the keycode port of NIOS II, via a special HPI interface of input and output. NIOS II finally gives the keycode information to the logic analysis part by its output, keycode. After analyzing the necessary information, the logic part will send various control signals to the VGA components so that pixels can be painted with right colors. The picture below illustrates the architecture.




# Written Descriptions of Circuits

## The Whole System

In general, we divide the system into four parts:  the EZ-OTG chip and USB interface, NIOS II system, logic analysis hardware and the VGA monitor and interface. Below is how these four parts interact with each other. When a keyboard is connected

to the DE2-115 board, it's actually connected to one of the SIEs on the EZ-OTG chip, which acts as the front end of the USB controller. When we press a button, the corresponding keycode information will be captured by SIE1. (In our project, only four direction keys are relevant. Therefore, whichever is pressed first will be first captured and analyzed.) The USB keyboard enumeration process in the main function will then set up various descriptors, configure SIE1 as the host and eventually poll keyboard information. This will make sure that CY7C67200 chip can have right information whenever keys are pressed. We can then do some reading and writing with the USB chip so that keyboard data can get into the hardware site. The connections between the USB chip and the FPGA board are made through the Host Port Interface (HPI), which contains 4 registers (HPI_DATA, HPI_MAILBOX, HPI_ADDRESS and HPI_STATUS). To access the content in the chip's RAM, we need to put data into these four registers first. To read some data, we first put the desired addresses into HPI_ADDRESS. CY7C67200 will then put data represented by the addresses into HPI_DATA, which we should read from. To carry out writing, we put destination addresses into HPI_ADDRESS and data into HPI_DATA. The USB chip will then put data into the locations represented by the addresses. In this way, data can be transported between NIOS II system and the USB interface.

With the information from the USB chip, NIOS II can then transmit the keycode data to the hardware logic components by its output port, keycode. Knowing which key is pressed, the logic analysis part will decide which directions Pacman and ghosts should move along. As a result, positions of Pacman and ghosts and existences of beans will be updated according to their motions. These motion and position information will finally go to Color Mapper and other VGA components,which will paint different colors to pixels according to positions of the characters, objects and background. This is basically how the system is organized.

## USB Protocols

To carry out writing and reading with the USB chip, we write 4 C functions, which are attached in the appendix. The first one is IO_Write, which allows us to write value to a single register. To achieve this, we first set *otg_hpi_cs (active low) to 0 so that the chip is turned on. Then we put the register address (00, 01, 10 or 11) into *otg_hpi_address and data into *otg_hpi_data. After these two are ready, we turn the writing enable on. The EZ-OTG chip will automatically put data into specified registers. After the writing process is finished, writing and chip select are turned off consequently.

The second one is IO_Read function, which allows us to read value from a single register. Like IO_Write,we first turn on the chip. Then we put the target address into *otg_hpi_address. After this, we can turn reading on and EZ-OTG will fetch data from

the specified register and put data into *otg_hpi_data. Once the reading process is finished, we turn both reading and chip select off.

The third one is the USBRead function, which allows us to read from a RAM location. To accomplish this, we first use an IO_Write function to put the 16-bit address into 2, which is 10 or HPI_ADDRESS. We then use an IO_Read function to 0, or 00, which represents HPI_DATA register.

The last one is USBWrite function, which allows us to write values to a RAM location. To do this, we first put the destination address into HPI_ADDRESS by an IO_Write function. We will then use another IO_Write function to put the 16-bit data into HPI_DATA register. EZ-OTG will then store the data into the specified address location.

NIOS II System

We first inherit the NIOS II we construct in Lab 7 because the majority part is very similar. Nevertheless, we add several components. For example, we add the JTAG UART peripheral, which enables us to use printf and scanf statements in C. This becomes helpful when the main function is printing some error messages so that we can know which part of the keyboard and USB interface doesn't work. We also add some input and output signals. These include Keycode(output, 16 bits), otg_hpi_cs(output, 1 bit), otg_hpi_address(output, 2 bits), otg_hpi_data(inout, 16 bits), otg_hpi_r(output, 1 bit) and otg_hpi_w(output, 1 bit). These are signals that are used to control the USB chip so that data can be read from and written into it. Otg_hpi_address is 2-bit wide because we have 4 registers to write into and read from, as mentioned above. Otg_hpi_data is 16-bit wide because all the data and addresses are 16-bit wide. Another interesting thing to notice is that FPGA board doesn't really support inout signals. As a result, otg_hpi_data is split into an input signal and an output signal in generated SystemVerilog codes, although we declare it as an inout signal in Qsys.

## Hardware Logic Analysis Components

The hardware logic analysis components generate logic for motions of characters, existences of beans, score keepings and life recordings. They will follow the rules mentioned above to do these jobs. For the motion logic of ghosts and Pacman, the 2-always model is adapted. In the always_comb block, the logic will analyze and generate the next-cycle values, while in the always_ff block, variables will get their new values for the current clock cycles.

## Color Mapper and Other VGA Components

The VGA controller module will keep going through the pixels on screen in a row-major order and output the coordinates of the current pixel to the color mapper. The

ball module (motion logic for Pacman) and ghost modules will also give the color mapper the current coordinates of the Pacman or ghosts. With these information, color mapper will draw different things at different locations on the screen. For a given pixel, it will first check if it's in the range of Pacman, ghosts, normal beans, the special bean, the score-keeping table, life recording table or walls. Otherwise, the pixel must be in the black path, in which characters move and beans are located. After determining identities of pixels, it will paint corresponding colors. For walls and beans, which we hard-code, the colors are fixed and we just give corresponding values to R, G and B components. For other complicated images such as Pacman, ghosts, number fonts and hearts, painting may include reading the sprites from the ROM in the on-chip memory.

### Other Parts

Another important part is the sprite. Because images like Pacman and ghosts are too complicated for us to hardcode, we instead generate sprites of these images. For Pacman and ghosts, we first create an 8x8 file and draw out the images by using the image workshop software on EWS machines. We then use Rishi's python codes to convert the images into files with 64 lines, each of which contains the RGB values (6 hex digits, 2 for each). To some extent, the 2D image is converted to a 1D sprite array. In Quartus, we then instantiates modules that set up the on-chip ROM to store these sprites. Therefore, we can just import proper address index values and get corresponding color values out. As for hearts, fonts and numbers, we use the font file provided in the course website.That file contains 16x8 sprites for 128 common characters. We use this file to instantiate a ROM, so that we can get sprites we want by inputting appropriate addresses.

## Implementations of Features

### Background Maze

We design the background maze by ourselves. As we all know, the screen is 640x480. Because of this, we make 8 pixels as a unit and divide everything by 8. Therefore, there are 80 units in the horizontal direction and 60 units in the vertical direction. Based on this simplification, we start to design the maze. We always make paths 1 unit, or 8 pixels wide so that the size of Pacman and ghosts can fit into the paths. In other areas, we make squares of different sizes as walls. Finally, the entire screen consists of walls and paths. Especially, we make walls along all the edges of the screen so that Pacman and ghosts won't go out of the screen and come back later at the other ends.

### Pacman

At the beginning of the game, we always press the Reset button, which we define as the inverted KEY1, so that all components are at original positions on the screen. Now that all wall units and paths are 8-bit wide, we always make the center of the Pacman located at the center of the path. For example, the area of 0<=X<=7 is a wall and the area of 16<=X<=23 is a wall, too. The Pacman center then needs to be at X=11, which is the center of the path between the two walls. Similarly, if the areas of 456<=Y<=463 and 472<=Y<=479 are walls, the Pacman center needs to be at Y=467, which is also the path center in between. This calibration will make it more natural to make turns in the maze. In this report, when we talk about Pacman positions, we always refer to the center positions, unless stated otherwise.

Initially, Pacman will be at X=627, Y=467. When keys are pressed, it will move accordingly. In the module for Pacman, we construct an always_ff block, in which we give variable “oldkey” the value of the current keycode. In this way, we can record the previous pressed keys. We also instantiate 4 wall decider modules, each of which determines if there is wall ahead if the Pacman is moving up, down, left or right. If any of these says yes, the X and Y motion values will become zero. Aside from these, we instantiate another 4 wall decider modules, each of which determine if there is wall at the top, bottom, left and right of Pacman. This will be used when we determine if Pacman can change directions when different keys are pressed. The walls in new directions prevent Pacman from turning, Pacman will just keep original motions. This process applies to all of “W”, “S”, “A” and “D”. If no keys or invalid direction keys are pressed, the Pacman module will still check for walls in current directions according to the results from the wall decider modules. If this doesn’t make Pacman stop, the machine will then check if oldkey is x001A (W). If so, it will then check the result from the wall decider to see if there is wall at the top. If wall isn’t there, Pacman will just begin to move up to finish the motion changes that can’t be finished because of walls several frame cycles ago. Similar checking processes will happen for oldkeys that are x0016(S), x0004(A) and x0007(D). If none of the above can make Pacman change motions, it will just keep original motions. Specifically, X_Motion_in and Y_Motion_in will just get the values of X_Motion and Y_Motion, respectively.

One thing to notice is that Pacman can’t go into walls like Lab 8 because the Pacman center needs to be constantly on central lines of paths. As a result, we use X_motion_in and Y_Motion_in to update positions, unlike Lab 8.

Another important trait of Pacman is its mouth, which keeps opening and closing. To make this happen, we draw and create five pictures and sprites. One corresponds to a yellow circle, which corresponds to Pacman with closed mouth. The other four correspond to open mouth to the top, bottom, left and right respectively. Note that our sprite is an 8x8 square while our shapes, which are in yellow, are not regular. As a result, we just paint other areas as black, so that those pixels can be accommodated

into the black paths. To make the open-to-close cycles happen, we declare a 6-bit counter, which will increment itself by 1 at every clock cycle. Whenever the leading bit is 0, closed mouth will be displayed, while open mouths will be shown when the leading bit is 1. As a result, the mouth will be closed for 32 frame cycles and open for another 32 frame cycles. Now that the frame clock is 60Hz, the mouth will be closed for half second and open for half second, which is pretty perceivable as we test. When we print Pacman, we first check if the leading bit of counter is 0. If so, we use the sprite for close mouth. Otherwise, we use different open mouth sprites according to motion directions. If Pacman is not moving, we will print close mouth.

## Ghosts

We have three ghosts in the map. For similar reasons of calibrations, we also establish the three ghosts as 8x8 sprite array and locate their centers along the central lines of paths. Also, to make ghosts respond promptly, we always use X_Motion_in and Y_Motion_in values to update positions. The red ghost starts from X=11, Y=11. The green ghost starts from X=587, Y=11. The pink ghosts starts from X=27, Y=331. As mentioned above, the red ghost can't move until Pacman starts to move at the beginning of the game. To make this happen, we declare a variable "move" in the red ghost module. At every clock cycle, move will become 1 from 0 if the x motion value or the y motion value is not zero. We will then only assign new motion values to the red ghost when move is 1. The basic logic for the red ghost is to chase Pacman. In another word, it will always move to the point that is closest to Pacman. To do this, we first instantiate four wall decider modules, to determine if there is wall at the top, bottom, left and right of the red ghost. We instantiate four distance modules to calculate the square of distances between Pacman center and the up, bottom, left and right points of the red ghost respectively as well. The results from the wall decider and distance modules will then go to a comparator module that determines which point is closest to Pacman without walls. The red ghost module will then move according to this result. As for the green and pink ghost, the overall logic is the same, except for their target. The green ghost is following the Pacman with the new horizontal position. (It's trying to approach (PacmanX+xmotion, PacmanY)) The pink ghost is chasing the Pacman with new vertical coordinates. (It's trying to approach (PacmanX, PacmanY+ymotion)) In this way, ghosts can move in an organized AI.

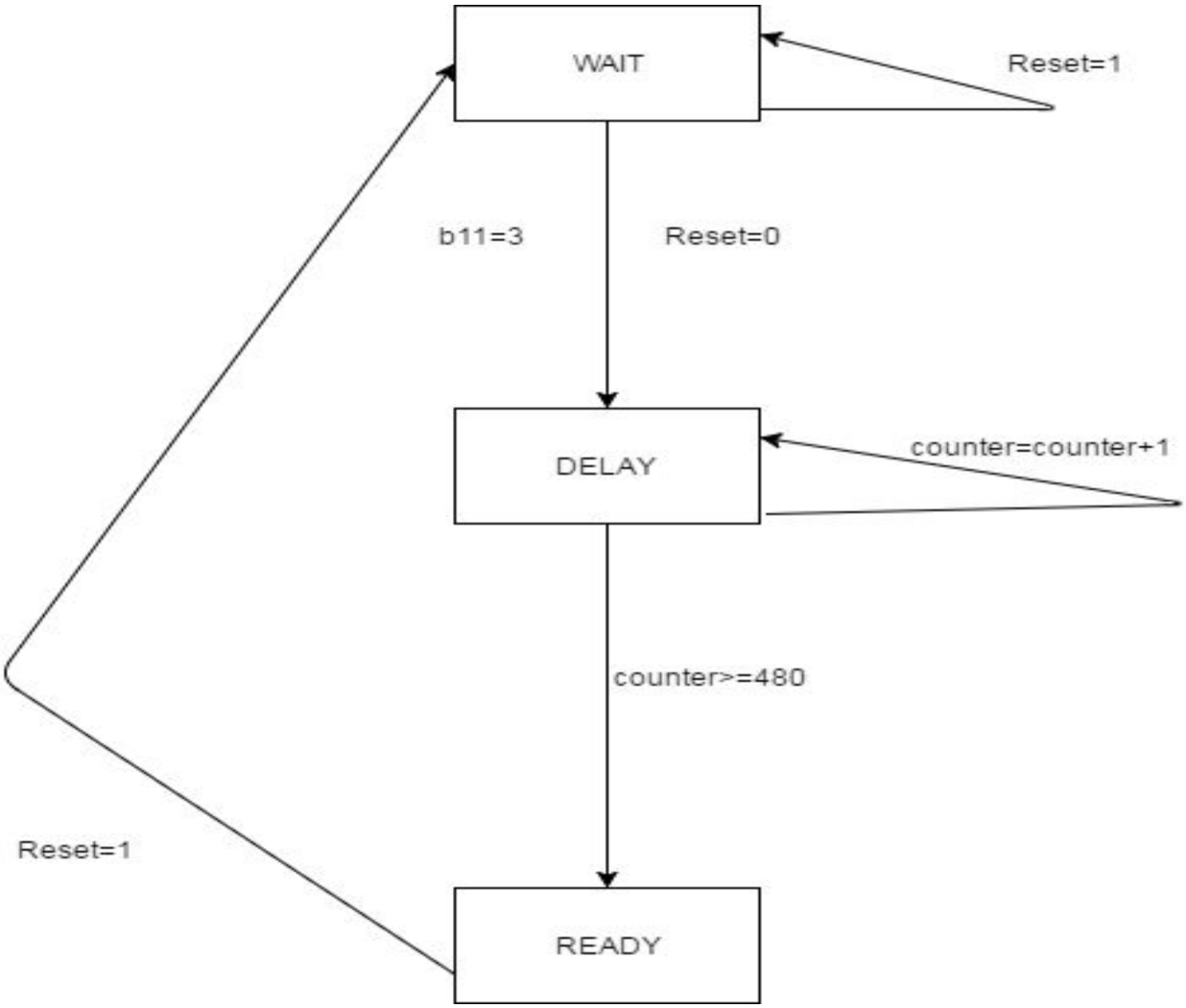
As for their images, we first draw them out in the 8x8 frame by the photo workshop. Then we convert them to sprites and store into the on-chip memory.

## Beans

We put 10 normal red beans and 1 special green beans in the map. Normal beans are circles of radius 2 and the special bean is a circle of radius 3. They are all located along central lines of path. In the color mapper module, we check if a specific pixel is in the range of beans by calculating the square of distance between the pixel and bean centers. We then paint corresponding colors. For every normal bean, we declare a  marking variable to record if the bean center is 4 units within the Pacman center. If so, it will become 0 from 1 in the current cycle and never change back to 0 unless being reset. If a variable like this is 1, the corresponding bean won't be drawn out, which means it's eaten by Pacman. For the special bean, the number is 3. The sum of all such 11 variables will be the total score, which is connected to and shown on the hex displays.

## Pacman's Attacking Ghosts

For this special effect, we build a state machine. The state diagram looks like following:



Here, Reset is the inverted version of KEY1 (active low). B11 is the marking variable we mentioned above for the special bean. At the beginning when we press KEY1, the state machine will start from WAIT state. The state machine won't go to DELAY state unless b11 becomes 3, which means that the special bean is eaten. Initially, counter is reset to zero. As a result, to go from DELAY to READY, counter has to increment itself by 480 times. Given that the frequency of the frame clock is 60Hz, the machine will stay in DELAY state for 8 seconds. In this state, the output, "blue", will become 1, which will then affect the behaviors of three ghosts. When the machine finally jumps to READY state, blue drops to 0. The output blue will then be connected to the three ghost modules. Whenever blue is 1, the ghost modules will move according to the results from another comparators that determines which point is the farthest from the Pacman without walls. In this way, ghosts will move away from the Pacman for 8 seconds.

When blue is high, color mapper will do different things as well. When we access the ghost sprites, we need to analyze colors first. If it's black, that will be the black path or ghost eyes, which should not be changed. Otherwise, it will be in the ghost body and we need to paint them as blue instead of original colors during those eight seconds.

Finally, Pacman is able to attack ghosts in this situation. To do this, we set up a module which detects if Pacman center has the same coordinates as any of the three ghosts. If so, it will send an "over" signal to the corresponding ghost module. The over signal will reset the ghost module(s) and reset the ghost(s) to the initial position(s). This is how Pacman eats ghosts during the special 8-second periods.

## Score Keeping on Screen

To record scores on the screen, we use the space inside a thick wall to print "Score:" and the corresponding numbers. We accomplish this by utilizing the font file. As we discussed previously, we use hex displays to hold the sums of all marking variables for beans, which sum to all the points gained by eating beans. Nonetheless, we also need to take points of eating ghosts into account. We instantiate a module, which records the number of ghosts eaten by the Pacman. Then the counter variable in this module will add itself with five times the number of ghosts eaten. As a result, the points of eating ghosts are accumulated in this way.

The variable which holds points for eating ghosts will then be output to the top level, in which it's added to the points of eating beans. This is the total score. To determine what numbers to print on the screen, we first divide the total score by 10. The result is the first number to print on the screen. We then take the remainder of the previous computation, which is the second number to print.

## Life Recording on Screen

Similar to score-keeping, we pick up a space inside a thick wall to display the number of lives. As before, we first print “Life:” on the first line and two red hearts on the second line. To record how many chances Pacman has left, we instantiate a module, which detects if Pacman center has the same coordinates as any of the ghosts when blue is 0. This represents the scenario in which Pacman is caught by ghosts. When this happens, the module will send out an “over” signal to the Pacman module, which resets everything including Pacman’s position, ghosts’ positions and beans. A heart will be dropped if there is any left. To make this happen, we declare a variable inside the color mapper module that records the times Pacman returns to original positions after the game starts. This is because whenever the Pacman is caught by ghosts, it will return to its original position. Therefore, when Pacman returns to its original position, it must have been caught by ghosts. When this variable is 0, two hearts will be printed. When this variable is 1, only one heart will be printed. When the number is 2 or larger, no hearts will be printed. This will tell users that they lose the game and need to press KEY1 to reset the game. This is how life is recorded on the screen.

# Written Descriptions of Modules

Here are the modules we use.

**Module**: lab8_soc
**Inputs**: clk_clk, reset_reset_n, [15:0]otg_hpi_data_in_port
**Outputs**: [15:0]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_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 module is generated by Qsys, where we do some changes to our Lab 7 setup. For example, we add a JTAG_UART block which allows us to use print and scan statements. Also, we add some output or inout PIO blocks including Keycode, otg_hpi_cs, otg_hpi_address, otg_hpi_data, otg_hpi_r and otg_hpi_w. These are the signals that are going to interact with EZ-OTG.
**Purpose**: This module represents the NIOS II processor, which is the base of our project. As a result, we can control the SDRAM chip on the FPGA board by the signals this module sends out, so that our software program can be run properly. In this way, we can use it to interact with other parts in the system.

**Module**: font_rom
**Inputs**: [10:0]addr
**Outputs**: [7:0]data

**Description**: This font file establishes a ROM, in which sprites for 128 images are stored and they are organized in this way. Every 16 lines consist the sprite for one image, which represent 16 locations. Now that there are 128 sprites in total, the total number of ROM locations is 128*16=2^(7+4)=2^11. That's why the address input has 11 bits. Inside a location, there are 8 binary bits, among which 0 represents background and 1 represents colors. That's why the data output has 8 bits.
**Purpose**: This module provides us with a useful resources of drawing images. We just read from the ROM file and draw out irregular shapes according to the bits.

**Module**: eatghost
**Inputs**: Reset, over, frame_clk, blue, [10:0]Ghostrx, [10:0]Ghostry, [10:0]Ghostgx, [10:0]Ghostgy, [10:0]Ghostpx, [10:0]Ghostpy, [10:0]BallX, [10:0]BallY
**Outputs**: [23:0]eaten
**Description**: When frame_clk or blue rises, this module will always check if the coordinates of Pacman (BallX and BallY) are the same as any of the ghosts. Actually, it will count the number of ghosts which have the same coordinates as Pacman. It will then add eaten to 5 times of the number to record the points gained by eating ghosts.
**Purpose**: This module keeps updating points of eating ghosts when the special bean is eaten. This is helpful for calculating the total score.

**Module**: delay
**Inputs**: Reset, over, frame_clk, [1:0]b11
**Outputs**: blue
**Description**: When the special bean is eaten or b11 becomes 3, the state machine in this module will enter DELAY state from WAIT state. In DELAY state, it will send out blue signal as 1 so that ghosts can become blue and run away from Pacman. After spending 480 cycles in DELAY state, it will then go to READY state.
**Purpose**: This module constructs the state machine, which controls the logic of the special 8-second period after the special bean is eaten.

**Module**: ghostp
**Inputs**: Reset, frame_clk, over, overp, blue, [10:0]BallX, [10:0]BallY, [10:0]xmotion, [10:0]ymotion
**Outputs**: [10:0]GhostX, [10:0]GhostY
**Description**: In this module, the pink ghost will try to follow the new vertical position of Pacman. That is to say, it will follow (BallX, BallY+ymotion). (BallX and BallY are coordinates of Pacman, while xmotion and ymotion are motion values of Pacman.) To do this, it will compare the squared distance between the target point and the top, bottom, left and right point of the pink ghost center. It will then go to the closest point

without blocking walls. Obviously, this is when blue is 0. When blue is high, it will carry out the same comparison but go to the point that is farthest from Pacman without walls. When it is attacked by Pacman or overp is high, the pink ghost will return to its original position (27,331).
**Purpose**: This module constructs the AI of the pink ghost so that it can move orderly.

**Module**: ghostg
**Inputs**: Reset, frame_clk, over, overg, blue, [10:0]BallX, [10:0]BallY, [10:0]xmotion, [10:0]ymotion
**Outputs**: [10:0]GhostX, [10:0]GhostY
**Description**: In this module, the green ghost will try to follow the new horizontal position of Pacman. That is to say, it will follow (BallX+xmotion, BallY). (BallX and BallY are coordinates of Pacman, while xmotion and ymotion are motion values of Pacman.) To do this, it will compare the squared distance between the target point and the top, bottom, left and right point of the green ghost center. It will then go to the closest point without blocking walls. Obviously, this is when blue is 0. When blue is high, it will carry out the same comparison but go to the point that is farthest from Pacman without walls. When it is attacked by Pacman or overg is high, the green ghost will return to its original position (587,11).
**Purpose**: This module constructs the AI of the green ghost so that it can move orderly.

**Module**: gameover
**Inputs**: [10:0]BallX, [10:0]BallY, [10:0]GhostX, [10:0]GhostY, [10:0]GhostXg, [10:0]GhostYg, [10:0]GhostXp, [10:0]GhostYp, [3:0]sum, blue
**Outputs**: over
**Description**: This module will check if Pacman has the same coordinates as any of the ghosts when blue is low, which represents fact that Pacman is eaten by ghosts. It will also check if sum is 13, which means that Pacman has eaten all beans. When either of these two happens, it will send a high over signal, which basically resets the whole system.
**Purpose**: This module generates the logic of ending the game or losing one heart.

**Module**: ghostover
**Inputs**: [10:0]BallX, [10:0]BallY, [10:0]GhostX, [10:0]GhostY, [10:0]GhostXg, [10:0]GhostYg, [10:0]GhostXp, [10:0]GhostYp, blue
**Outputs**: overr, overg, overp
**Description**: This module checks if Pacman has the same coordinates as any of the ghosts when blue is 1. If that happens, it will then send over signals (overr, overg or overp) to corresponding ghost modules to reset their positions.

**Purpose**: This module generates the logic of attacking ghosts for Pacman after it eats the special bean.

**Module**: walldecider
**Inputs**: [10:0]DrawX, [10:0]DrawY
**Outputs**: wall_on
**Description**: Now that we design the background maze by ourselves, we can monitor every single pixel and decide if it's in a wall. That's what this module does. If it 's in the wall, it will send out wall_on as 1.
**Purpose**: This module will be used a lot when we are designing motion logic for Pacman and ghosts. In this way, Pacman can have effective collision detections and ghosts won't go into walls.

**Module**: ghost
**Inputs**: Reset, frame_clk, over, overr, blue, [10:0]BallX, [10:0]BallY, [10:0]xmotion, [10:0]ymotion
**Outputs**: [10:0]GhostX, [10:0]GhostY
**Description**: In this module, the green ghost will try to follow the center position of Pacman. That is to say, it will follow (BallX, BallY). (BallX and BallY are coordinates of Pacman, while xmotion and ymotion are motion values of Pacman.) To do this, it will compare the squared distance between the target point and the top, bottom, left and right point of the red ghost center. It will then go to the closest point without blocking walls. Obviously, this is when blue is 0. When blue is high, it will carry out the same comparison but go to the point that is farthest from Pacman without walls. When it is attacked by Pacman or overr is high, the red ghost will return to its original position (11,11).
**Purpose**: This module constructs the AI of the red ghost so that it can move orderly.

**Module**: frameRAM
**Inputs**: [18:0]read_address, Clk
**Outputs**: [23:0]data_Out
**Description**: The module establishes a ROM on the on-chip memory, that stores the sprites of Pacman with open mouth, facing up. When we input address, it will return 24-bit data at positive edges of clock signals. (8 bits for red, 8 bits for green and 8 bits for blue)
**Purpose**: This module will output color values when color mapper is drawing Pacman with open mouth, that is moving up.

**Module**: frameRAMa

**Inputs**: [18:0]read_address, Clk
**Outputs**: [23:0]data_Out
**Description**: The module establishes a ROM on the on-chip memory, that stores the sprites of Pacman with open mouth, facing down. When we input address, it will return 24-bit data at positive edges of clock signals. (8 bits for red, 8 bits for green and 8 bits for blue)
**Purpose**: This module will output color values when color mapper is drawing Pacman with open mouth, that is moving down.

**Module**: frameRAMb
**Inputs**: [18:0]read_address, Clk
**Outputs**: [23:0]data_Out
**Description**: The module establishes a ROM on the on-chip memory, that stores the sprites of Pacman with open mouth, facing to the left. When we input address, it will return 24-bit data at positive edges of clock signals. (8 bits for red, 8 bits for green and 8 bits for blue)
**Purpose**: This module will output color values when color mapper is drawing Pacman with open mouth, that is moving to the left.

**Module**: frameRAMc
**Inputs**: [18:0]read_address, Clk
**Outputs**: [23:0]data_Out
**Description**: The module establishes a ROM on the on-chip memory, that stores the sprites of Pacman with open mouth, facing to the right. When we input address, it will return 24-bit data at positive edges of clock signals. (8 bits for red, 8 bits for green and 8 bits for blue)
**Purpose**: This module will output color values when color mapper is drawing Pacman with open mouth, that is moving to the right.

**Module**: frameRAMd
**Inputs**: [18:0]read_address, Clk
**Outputs**: [23:0]data_Out
**Description**: The module establishes a ROM on the on-chip memory, that stores the sprites of Pacman with closed mouth. When we input address, it will return 24-bit data at positive edges of clock signals. (8 bits for red, 8 bits for green and 8 bits for blue)
**Purpose**: This module will output color values when color mapper is drawing Pacman with closed mouth.

**Module**: frameRAMghost

**Inputs**: [18:0]read_address, Clk
**Outputs**: [23:0]data_Out
**Description**: The module establishes a ROM on the on-chip memory, that stores the sprites of the red ghost. When we input address, it will return 24-bit data at positive edges of clock signals. (8 bits for red, 8 bits for green and 8 bits for blue) When we draw the green and pink ghosts, we still use this sprite. The only difference is that we paint green or pink at red pixels.
**Purpose**: This module will output color values when color mapper is drawing the three ghosts.

**Module**: distanceu
**Inputs**: [10:0]GX, [10:0]GY, [10:0]BallX, [10:0]BallY
**Outputs**: [32:0]distance
**Description**: This module calculates the squared distance between the ghost and top point of Pacman. It does this by first taking the differences between horizontal and vertical coordinates. Then it calculates the sum of the squares of the two differences and gives the value to distance.
**Purpose**: This module will be used when we design the motion logic.

**Module**: distanced
**Inputs**: [10:0]GX, [10:0]GY, [10:0]BallX, [10:0]BallY
**Outputs**: [32:0]distance
**Description**: This module calculates the squared distance between the ghost and bottom point of Pacman. It does this by first taking the differences between horizontal and vertical coordinates. Then it calculates the sum of the squares of the two differences and gives the value to distance.
**Purpose**: This module will be used when we design the motion logic.

**Module**: distancel
**Inputs**: [10:0]GX, [10:0]GY, [10:0]BallX, [10:0]BallY
**Outputs**: [32:0]distance
**Description**: This module calculates the squared distance between the ghost and left point of Pacman. It does this by first taking the differences between horizontal and vertical coordinates. Then it calculates the sum of the squares of the two differences and gives the value to distance.
**Purpose**: This module will be used when we design the motion logic.

**Module**: distancer
**Inputs**: [10:0]GX, [10:0]GY, [10:0]BallX, [10:0]BallY

**Outputs**: [32:0]distance
**Description**: This module calculates the squared distance between the ghost and right point of Pacman. It does this by first taking the differences between horizontal and vertical coordinates. Then it calculates the sum of the squares of the two differences and gives the value to distance.
**Purpose**: This module will be used when we design the motion logic.

**Module**: comparator
**Inputs**: hitu, hitd, hitl, hitr, [32:0]du, [32:0]dd, [32:0]dl, [32:0]dr
**Outputs**: [1:0]dir
**Description**: This module will determine which direction is the closest to the Pacman without walls. It will do this by first comparing the up and bottom point. If there is wall in a specific direction, it will count the distance as zero. It then compares the left and right points. Finally, it will compare the results from the previous two comparisons and determine which direction that represents. When dir is 00, Pacman should move up. When dir is 01, Pacman should move down. When dir is 10, Pacman should move to the left. When dir is 11, Pacman should move to the right.
**Purpose**: This module will be used when we design the motion logic for ghosts under normal conditions.

**Module**: comparatorf
**Inputs**: hitu, hitd, hitl, hitr, [32:0]du, [32:0]dd, [32:0]dl, [32:0]dr
**Outputs**: [1:0]dir
**Description**: This module will determine which direction is the farthest to the Pacman without walls. It will do this by first comparing the up and bottom point. If there is wall in a specific direction, it will count the distance as zero. It then compares the left and right points. Finally, it will compare the results from the previous two comparisons and determine which direction that represents. When dir is 00, Pacman should move up. When dir is 01, Pacman should move down. When dir is 10, Pacman should move to the left. When dir is 11, Pacman should move to the right.
**Purpose**: This module will be used when we design the motion logic for ghosts after the special bean is eaten by Pacman.

**Module**: ball
**Inputs**: Reset, frame_clk, over, [15:0]keycode,
**Outputs**: [10:0]BallX, [10:0]BallY, [10:0]BallS, [10:0]xmotion, [10:0]ymotion, ledl, ledr, ledd, ledu
**Description**: This module generates the motion logic for Pacman. Some parts of it are similar to Lab 8. We add collision detections and responses to old keys pressed.

**Purpose**: The module makes Pacman move in an organized way so that users can easily use keyboard to control its motions.

**Module**: color_mapper
**Inputs**: [10:0]BallX, [10:0]BallY, [10:0]BallS, [10:0]DrawX, [10:0]DrawY, [10:0]GhostX, [10:0]GhostY, [10:0]GhostXg, [10:0]GhostYg, [10:0]GhostXp, [10:0]GhostYp, [10:0]xmotion, [10:0]ymotion, [23:0]tenth, [23:0]oneth, frame_clk, Reset, clk, over
**Outputs**: [7:0]VGA_R, [7:0]VGA_B, [7:0]VGA_G, [3:0]sum, blue
**Description**: This module will basically paint colors to pixels according to the objects or characters they belong to. Sometimes, it will hard-code, while during other times, it will read from the on-chip memory.
**Purpose**: This module allows pixels to have different colors so that we can distinguish different things.

**Module**: HexDriver
**Inputs**: [3:0]In0
**Outputs**: [6:0]Out0
**Description**: This module will convert 4 bits of binary value to a corresponding 7-bit code, which will then be converted to the hexadecimal bit of the 4 bits.
**Purpose**: This allows to show values on the Hex displayers on the FPGA board. In this project, we will show keycodes and scores on hex displays.

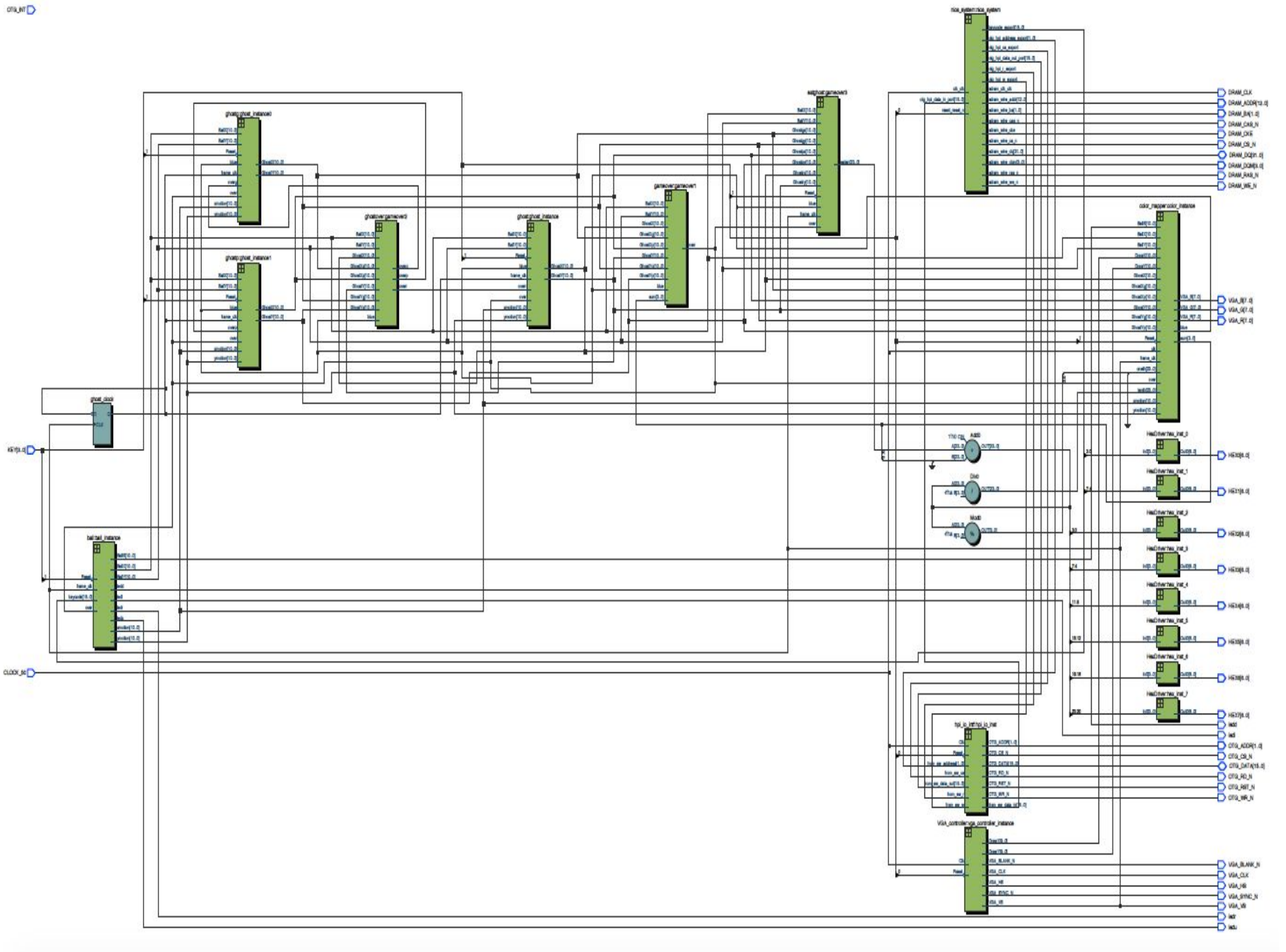
**Module**: hpi_io_intf
**Inputs**: Clk, Reset, [1:0]from_sw_address, [15:0]from_sw_data_out, from_sw_r, from_sw_w, from_sw_cs
**Outputs**: [15:0]from_sw_data_in, [1:0]OTG_ADDR, OTG_RD_N, OTG_WR_N, OTG_CS_N, OTG_RST_N
**Inout:** [15:0]OTG_DATA
**Description**: This module will assign data to and from the USB chip accordingly. At first, it creates an internal variable, from_sw_data_out_buffer, as a buffer to accept the value that might be written into EZ-OTG. When Reset is high, it will turn off the chip select, reading enable and writing enable signals. When Reset is low, it will set those signals as what come from NIOS II. Also, it will set the data to NIOS II, from_sw_data_in as the bidirectional value OTG_DATA and put the value from NIOS II, from_sw_data_out into from_sw_data_out_buffer. When writing is enabled, OTG_DATA will be the value of the buffer variable. In all other cases, OTG_DATA will be high impedance.
**Purpose**: This acts as the interface, or tri-state buffer between NIOS II and the USB chip. Because of this module, data flow in both directions between these two parts.

**Module**: VGA_controller
**Inputs**: Clk, Reset
**Outputs**: VGA_HS, VGA_VS, VGA_CLK, VGA_BLANK_N, VGA_SYNC_N, [9:0]DrawX,[9:0]DrawY
**Description**: Now that the whole screen is organized as many pixels in a row-major order, this module just keeps moving along these pixels in the row-major order. To be more specific, it will go over one pixel at every cycle of VGA_CLK. When a row is finished, it will set VGA_HS to 0, which is active low. When all rows are finished, it will set VGA_VS to 0, which is active low as well. Additionally, it will use DrawX and DrawY to represent the location of the current pixel.
**Purpose**: During the process of going over these pixels, other VGA components will use the pixel information to assign right colors so that the whole image can keep being updated continuously. To some extent, this will prevent the ball from going diagonally.

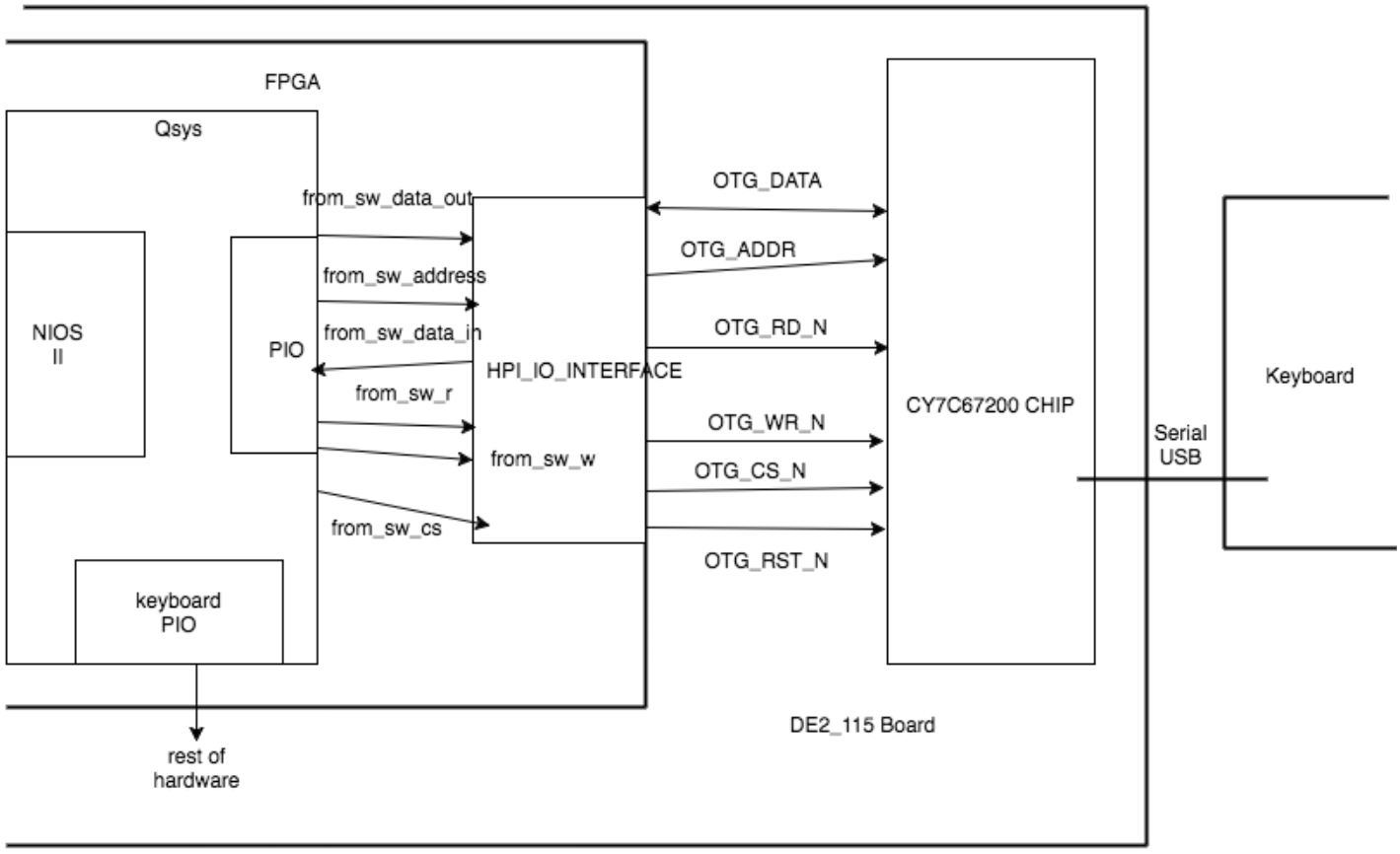
**Module**: lab8
**Inputs**: CLOCK_50, [3:0]KEY,
**Outputs**: [6:0]HEX0, [6:0]HEX1, [7:0]VGA_R, [7:0]VGA_G, [7:0]VGA_B, VGA_CLK, VGA_SYNC_N, VGA_BLANK_N, VGA_VS, VGA_HS, [1;0]OTG_ADDR, OTG_CS_N, OTG_RD_N, OTG_WR_N, OTG_RST_N, [12:0]DRAM_ADDR, [1:0]DRAM_BA, [3:0]DRAM_DQM, DRAM_RAS_N, DRAM_CAS_N, DRAM_CKE, DRAM_WE_N, DRAM_CS_N, DRAM_CLK, ledl, ledr, ledu, ledd, [6:0]HEX2, [6:0]HEX3, [6:0]HEX4, [6:0]HEX5, [6:0]HEX6, [6:0]HEX7
**Inout:** [15:0]OTG_DATA, [31:0]DRAM_DQ
**Description**: This module basically connects all the modules above together. In addition, it utilizes the HexDriver module to display the lower 8 bits of keycodes to the hex displays on board. It also displays the total scores on the other six hex displays.
**Purpose**: This module acts as the top level of the whole project. In this way, NIOS II, VGA components, USB CY7C67200 chip and logic analysis hardwares are organized in order.

# Block Diagram

This is the top-level diagram generated by Quartus.


OTG_INT
KEY[3..0]
CLOCK_50
ghost_clock
ball:ball_instance
ghost:ghost_instance1
ghost:ghost_instance2
ghostover:gameover3
ghost:ghost_instance
gameover:gameover1
wallover:gameover3
nios_system:nios_system
color_mapper:color_instance
Add0
Div0
Mod0
hpi_io_intf:hpi_io_inst
VGA_controller:vga_controller_instance
DRAM_CLK
DRAM_ADDR[12..0]
DRAM_BA[1..0]
DRAM_CAS_N
DRAM_CKE
DRAM_CS_N
DRAM_DQ[31..0]
DRAM_DQM[3..0]
DRAM_RAS_N
DRAM_WE_N
VGA_R[7..0]
VGA_G[7..0]
VGA_B[7..0]
HEX30[6..0]
HEX31[6..0]
HEX32[6..0]
HEX33[6..0]
HEX34[6..0]
HEX35[6..0]
HEX36[6..0]
HEX37[6..0]
OTG_ADDR[1..0]
OTG_CS_N
OTG_DATA[15..0]
OTG_RD_N
OTG_RST_N
OTG_WR_N
VGA_BLANK_N
VGA_CLK
VGA_HS
VGA_SYNC_N
VGA_VS

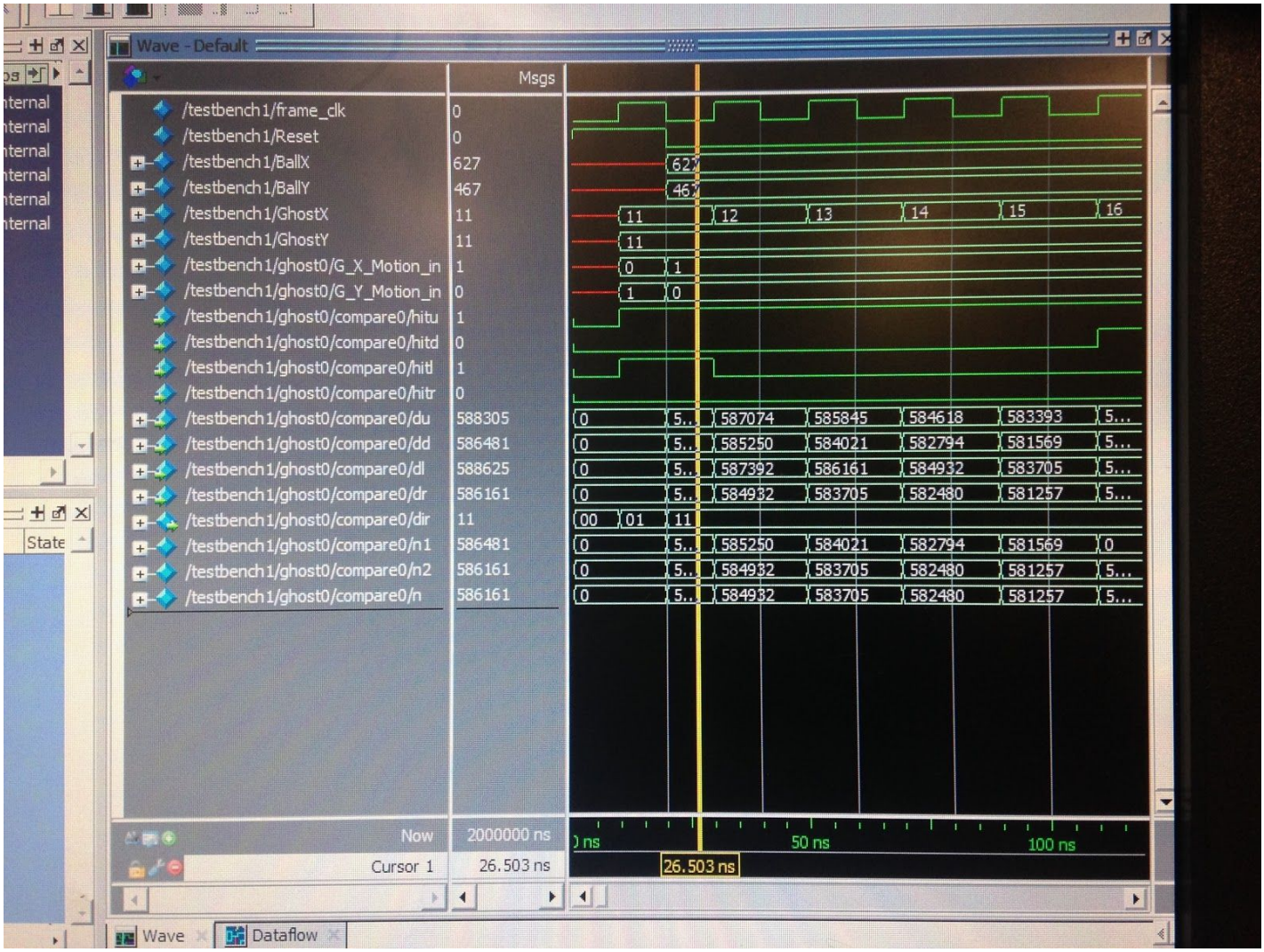
To present how various components interact with each other more clearly, we draw this diagram:



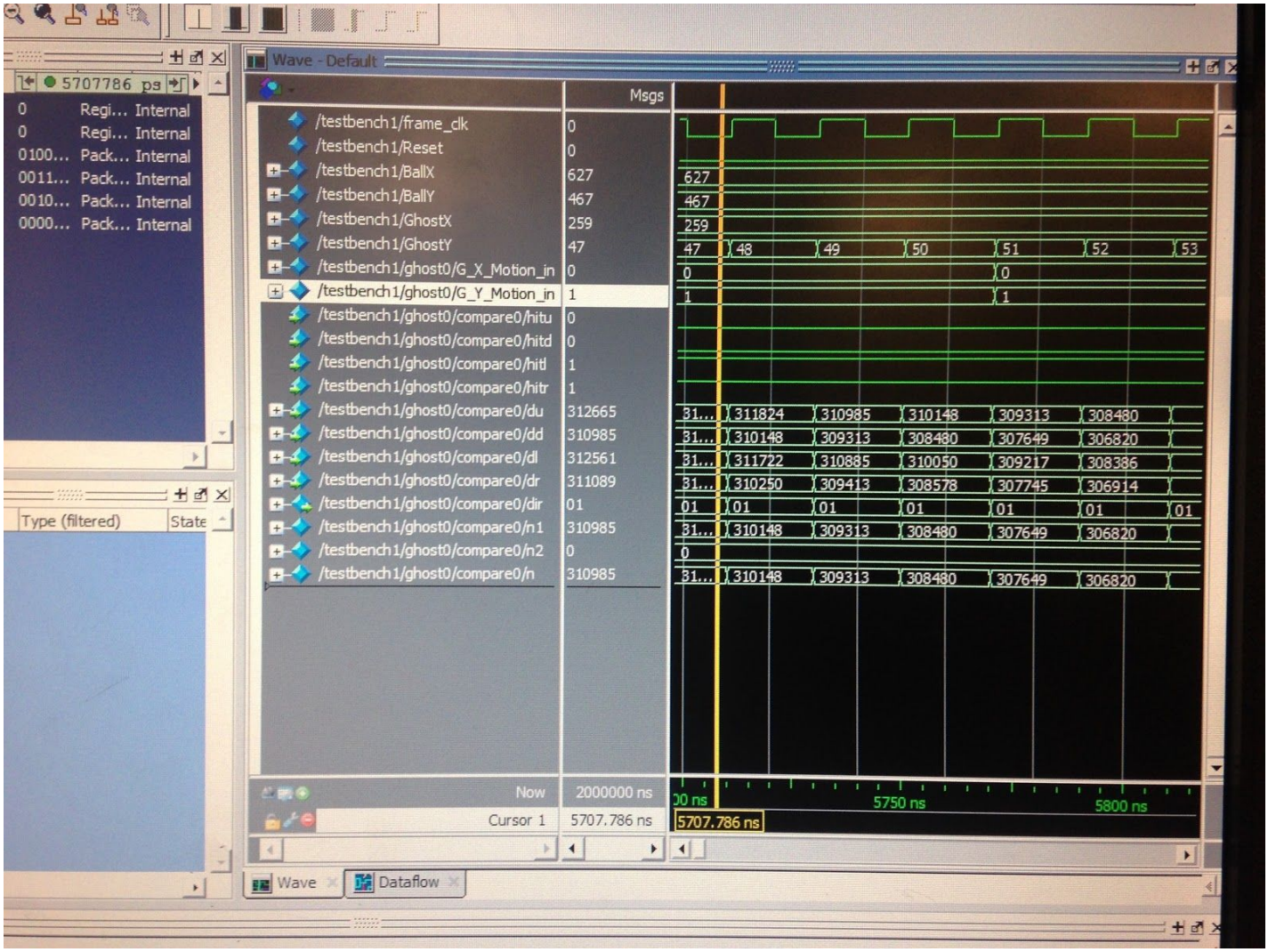

From this, we can see that NIOS II isn't in direct contact with the keyboard. Instead, HPI_IO_INTERFACE is buffering in between. The CY7C67200 chip is transmitting keycode data to the FPGA board.

# Simulations

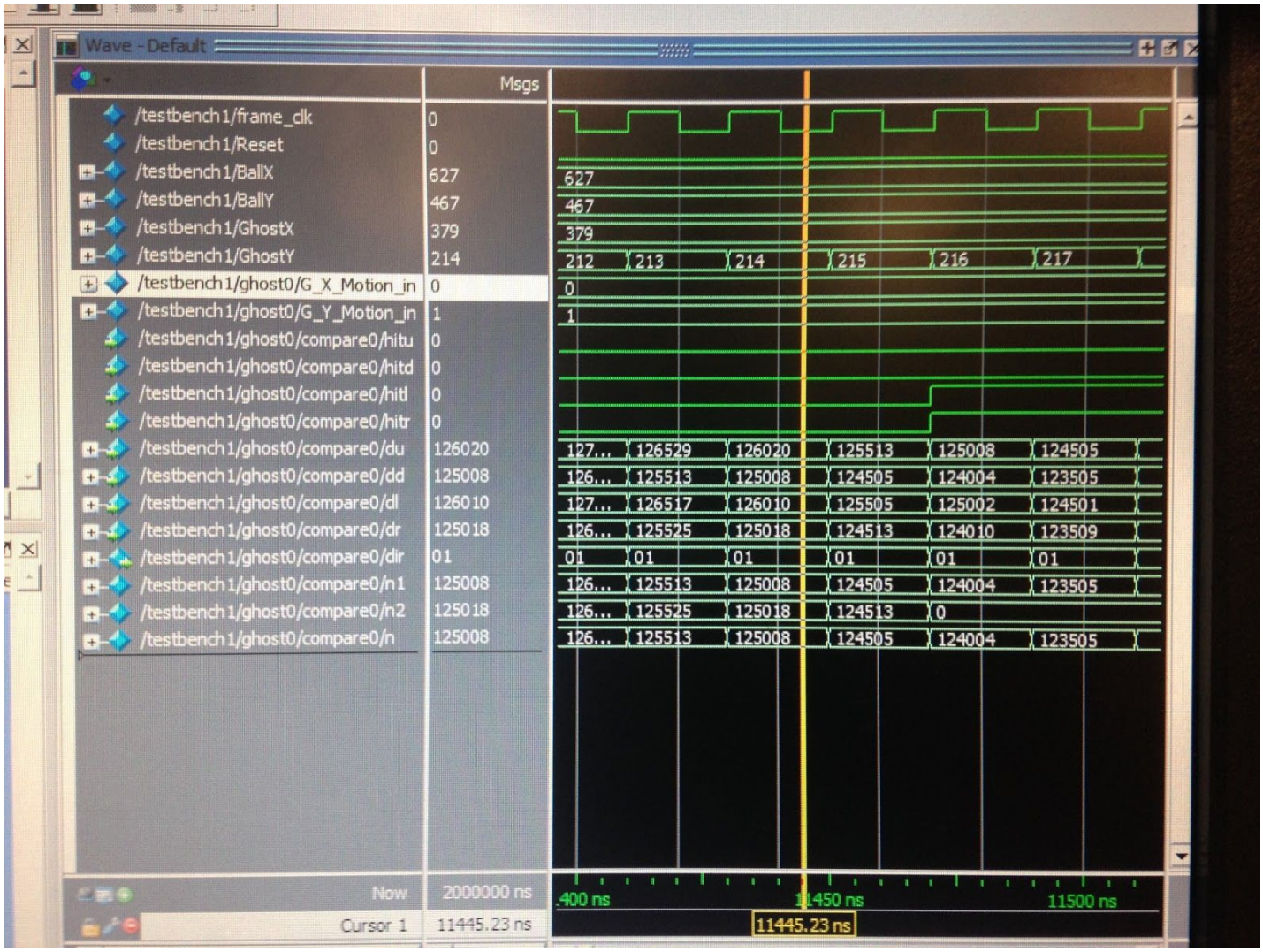
Before we compile our SystemVerilog codes, we first write a testbench to test our ghost module and gain the following waves. (Note: For simplicity, Pacman will just stay at its original position (627, 467) during simulations and the ghost will approach it from (11,11))



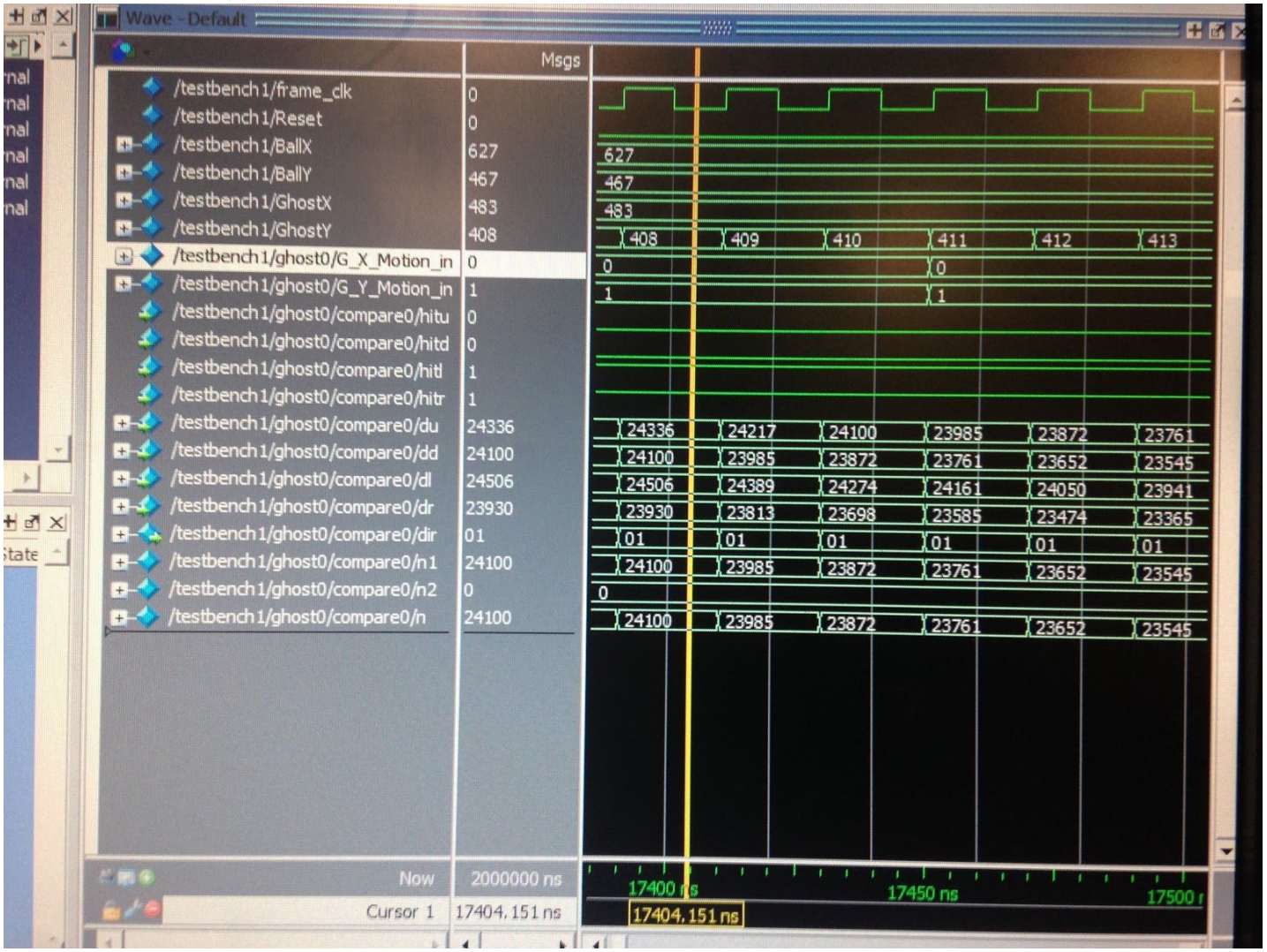

As we can see, after Reset becomes zero, the ghost starts to move towards Pacman. As we can see from the left side, right is the closest direction without wall (586161). As a result, dir becomes 11 and G_X_Motion_in is 1. Consequently, the horizontal coordinate increases by 1 at the next cycle.




At this point, moving down is the closest direction (310985). Therefore, dir is 01 and G_Y_Motion_in is 1. At the next cycle, the y coordinate increases by 1.




At this point, moving down is  still the closest direction (125008). Therefore, dir is 01 and G_Y_Motion_in is 1. At the next cycle, the y coordinate increases by 1.




At this point, moving down is  still the closest direction (24100). Although moving right will be closer (23930<24100), there is wall on the right. Therefore, dir is 01 and G_Y_Motion_in is 1. At the next cycle, the y coordinate increases by 1. In fact, the ghost is already very close to Pacman vertically by this point.

# Design Statistics

| | |
|---|---|
| LUT | 11388 |
| DSP | Simple Multipliers (18-bit): 62<br>Embedded Multiplier Blocks: 62<br>Embedded Multiplier 9-bit Elements: 124<br>Signed Embedded Multipliers: 62 |

| | |
|---|---|
| Memory (BRAM) | 55,296/3,981,312 (1%) |
| Flip-Flop | 2489 |
| Frequency | 91.81MHz (altera_reserved_tck) |
| Static Power | 102.63mW |
| Dynamic Power | 1.15mW |
| I/O Thermal Power Dissipation | 89.23mW |
| Total Power | 193.01mW |

From this table, we can see that the project consumes more LUTs, flip-flops and power than all the labs due to its large scale and complexity. Also, it consumes more DSP components than Lab 8 because the game logic and keyboard activities are much more complicated than Lab 8. Unlike some labs, the memory (BRAM) usage here is no longer zero because we use on-chip memory space to store sprites. Finally, as for the frequency, the number for Lab 8 is 134.44MHz. This is because the logic circuit here is much more complicated and thus more delays will happen in the circuit. As a result, the system cannot be run as fast as Lab 8.

When we test our designs, we initially have wrong motion logic. When we compare the waveform results, we find the problem: SystemVerilog will regard binary values as negative if the leading bit is 1. As a result, when we set up some if conditions, chaos messes up our circuit. We then adjust the data type and finally fix the problem.

In addition, our game suddenly ends by itself for two times, but this kind of situation is very rare. In our opinion, this is due to the glitches and unstable signals inside the FPGA circuit.

# Difficulty Evaluations

In overall, our game has the following features:

1.Motion logic of Pacman: keyboard responses, collision detections
2.AI of three ghosts: The ghosts are able to surround Pacman from different directions
3.Special Beans: Ghosts will become blue and run away from Pacman for 8 seconds after the special bean is eaten. Pacman will be able to attack ghosts during the 8 seconds.
4.Score keeping: 1 point for normal bean, 3 points for special bean, 5 points for eating ghosts
5.Life recording: two lives. One will be dropped when Pacman is caught by ghosts.
6.Multiple ghosts (more sprites) make the game more fun to play.

7.Old keys recording: For example, when users want Pacman to turn left, they only need to press the button once. Pacman will turn left whenever it can.
8.Pacman will keep opening and closing its mouth when moving, and its mouth is in the same direction as it moves.

In terms of our proposal, we finish the baseline and most of the additional features including special beans, scorekeeping, multiple ghosts and the ghost AI design, and also some more features that are not in the proposal (life recording and pacman's mouth). In our opinion, we deserve at least 7 points for difficulty.

# Post-Project Thoughts

In overall, we think we are very successful in designing the motion logic for Pacman and AI of the three ghosts. If we have more time, we can design some interfaces for the beginning and end of the game so that users can have nicer experiences for their winning or losing. We can also design a state machine, by which users can decide if they want the single-player mode or multi-player mode. In the multi-player mode, we can let the other player control one of the ghosts. The motion logic for the player-controlled ghost will be very similar to that of Pacman.

# Conclusion

For our final project, we design a Pacman game. To achieve this goal, we first inherit the overall structure from Lab 8 as the basis. This enables us to use the keyboard to interface with the FPGA board. We then design the background maze by calibrating the entire screen and hard-code the corresponding colors. We then use the photo workshop software to draw out images we need and convert them to sprite files by Rishi's helper tools. After storing the sprites into the on-chip memory, we design the motion logic of Pacman and AI of the three ghosts. We also add some additional features like score and life recording. Finally, our game works fine during the demo.

In conclusion, this is a very meaningful project, in which we get a chance to apply what we learn from ECE385 throughout the semester. This also deepens our understanding of hardware and software interfaces.

# Work Citations

1.We use Rishi's code to establish the on-chip memory storage ROM.
2.We use the font file from the course website as one of our modules.

# Appendix

IO_Write function

```
void IO_write(alt_u8 Address, alt_u16 Data)
{
	*otg_hpi_cs = 0;
	*otg_hpi_address=Address;  //Write into the address register
	*otg_hpi_data=Data;  //Write into the data register
	*otg_hpi_w = 0;  //Enable the chip writing here
	*otg_hpi_w = 1;  //Disable writing after finishing
	*otg_hpi_cs = 1;   //Turn off the chip
}
```

IO_Read function

```
alt_u16 IO_read(alt_u8 Address)
{
	alt_u16 temp;
	*otg_hpi_cs = 0;
	*otg_hpi_address=Address;  //Write into the address register
	*otg_hpi_r = 0;  //Enable the chip reading here
	temp=*otg_hpi_data;  //Read from the data register
	*otg_hpi_r = 1;  //Disable reading after finishing
	*otg_hpi_cs = 1;   //Turn off the chip
	return temp;
}
```

USBRead function

```
alt_u16 UsbRead(alt_u16 Address)
{
	alt_u16 temp;  //Declare a transient variable
	IO_write(2, Address);   //Write the address to the address register
	temp=IO_read(0);   //Read from the data register
	return temp;
}
```

USBWrite function

```
void UsbWrite(alt_u16 Address, alt_u16 Data)
{
	IO_write(2, Address);   //Write the address to the address register
```

```
        IO_write(0, Data);   //Write the data to the data register
}
```

The testbench file we use when we test the ghost module:

```
module testbench1();

timeunit 10ns;          // Half clock cycle at 50 MHz
                        // This is the amount of time represented by #1
timeprecision 1ns;

// These signals are internal because the processor will be
// instantiated as a submodule in testbench.
logic frame_clk = 0;
logic    Reset;
logic [10:0] BallX, BallY, GhostX, GhostY;

// Instantiating the DUT
// Make sure the module and signal names match with those in your design
ghost ghost0 (.*);

// Toggle the clock
// #1 means wait for a delay of 1 timeunit
always begin : CLOCK_GENERATION
#1 frame_clk = ~frame_clk;
end

initial begin: CLOCK_INITIALIZATION
    frame_clk = 0;
end

// Testing begins here
// The initial block is not synthesizable
// Everything happens sequentially inside an initial block
// as in a software program
initial begin: TEST_VECTORS
Reset = 1;              // Toggle Rest

#2 Reset = 0;
```

```
#6 BallX = 10'd627; // Toggle Run
 BallY = 10'd467;
end
endmodule
```