ECE 385

Fall 2017

Final Project

Pratheek Muskula, Hoesuh Jeong

AB5/ Thursday 11:00 AM - 1:50 PM

Zhenhong Liu

Proposal:

We proposed to implement a horizontal side scrolling platformer on the FPGA as a System-on-chip. Initially we hoped to make a game where a user controlled player traversed through a level, avoiding obstacles in order to collect points. As the character gained more points, more obstacles will be generated to make scoring harder, and the game will continue until the character dies. We said the baseline features of our project would be: using a frame buffer, having sprites, having animations, having simple game logic and AI, implementing gravity, scrolling, and collision detection. The extra features, we wanted to include were audio, more advanced AI, an advanced frame buffer, and some facial recognition stuff. To implement this project we hoped wanted to use the NIOS II CPU to interface with the USB keyboard and VGA Monitor, SRAM, and have most of the logic of our project be done in C using the NIOS II processor.

What We Ended Up Making:

In the end our project end up filling our baseline features for the most part. I say for the most part because we said we would use a frame buffer, but we never got to the point in our project where we needed to do animations that were complicated enough to use a frame buffer, so we ended up just using the fixed functions implementation of doing graphics. Other than that we met all the base guidelines we set.

We have a game where a character could move left, right, and jump in a simple 2-D world we created. The world has 1 enemy and one platform. When you jump proper gravity was implemented, and when you reach the right most edge of the screen after the game begins you will scroll the game map to the right and would scroll back left if you character wanted to go back. You can jump on to the platform and move around properly on it, and if you jump to the bottom of the platform you will be bounced back down. There is a single enemy that walks back and forth in a set range, and if you collide with it the game will end. The game ends by stopping the motion of the character and the enemy and not allowing any movement. To restart the game, you will need to press the reset button. There are movement animations for the character you control and the enemy, based on if they are moving and the direction they are facing. The game also has sprites for the background, the character, the enemy, and the platform.

Reflection on Differences Between the Proposal and the End Result

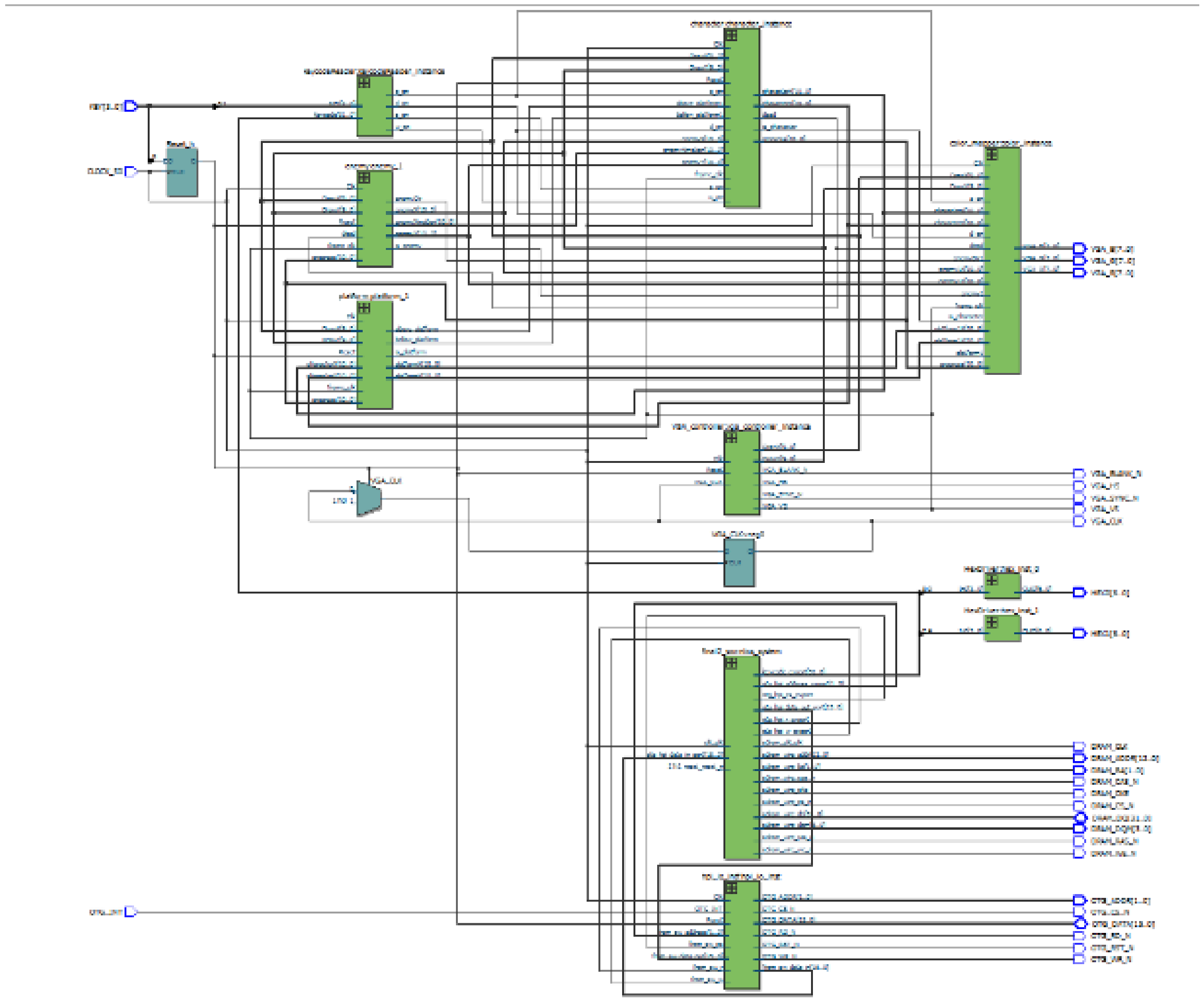
Aside from functionality aspects we hoped to make a better game overall, but we simple didn’t have enough time to do so. We did not have a start screen, a game over screen, or a point system implemented like we initially wanted to. In addition, we only created 1 enemy and 1 platform, to show that we had base functionality working when interacting with those game objects. Since we didn’t have time to flesh out all the game functionality we wanted, it’s obvious to say we didn’t have time to work on any of the extra features we said we would have.

Another aspect that is important to note, is how we implemented the project. Initially we planned to do most of the game logic, AI, and gravity in C, and the hand shaking the necessary data to hardware. But we actually only ended up using C for the usb keyboard and nothing else, everything else was implemented in hardware. We also though the On-Chip memory would be too small for what we wanted to do, but since we used a color palette, we had more than enough memory.

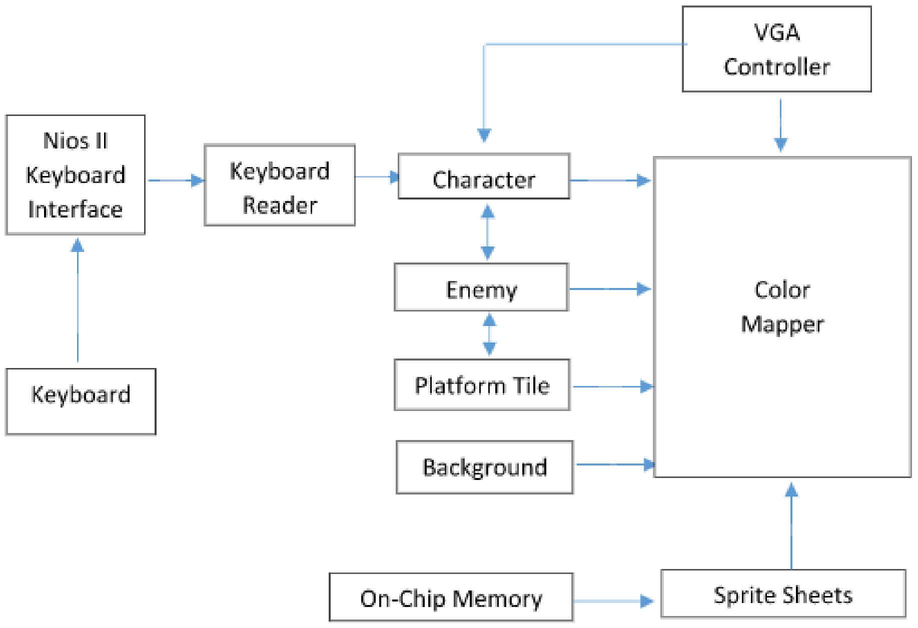
Despite the fact we did not have as impressive project as we wanted to, and implemented it differently than how we expected, all the base functionality we said we would implement, was implemented and working, and we'll elaborate after the Block Diagram section.

Block Diagrams:

Wired Circuit from RTL viewer:



General Block Diagram:



Character Movement:

Character movement is similar in our final project to how ball movement is implemented in lab 8, but there are some key differences. For one character motion only takes place when the character is touching the ground or the platform. Character X motion is based on keypress holds, and these holds are not read in the character.sv module itself, but in the keycodeReader.sv module. And it is important to note that, our keycode is 32 bits in our game, meaning we can take in 4 keystrokes. For specifics on how we do this check the main.c description, which is the comprehensive module description section. The way we parse the 32 bit keycode for specific key strokes is by breaking the keycode into 4,8 bits sections, and checking if the hex values match "w", "a", "s", "d" keys on the keyboard. Here is how we determined if "w" was being pressed.

```
assign w_on = (keycode[31:24] == 8'h1A |
                      keycode[23:16] == 8'h1A |
                      keycode[15: 8] == 8'h1A |
                      keycode[ 7: 0] == 8'h1A |
          (~KEY[1]));
```

The game movement can also be controlled by using KEY[3] as "a", KEY[2] as "w", KEY[1] as "d", in case the keyboard stops working.

Y movement deals with gravity, so we'll talk about that in the next section.

Gravity:

Gravity was the first game aspect we implemented after simple movement, and is included in the character.sv and enemy.sv modules. Any vertical movement in the game is affected by gravity, and gravity is applied on the character and the enemy unless they are on the platform or the ground. These statements take care of deciding whether to turn gravity on:

```
else if(Ball_Y_Pos + Ball_Height != Ball_Y_Max)
                                        Ball_Y_Motion_in = Ball_Y_Motion + 1;
else
                                        Ball_Y_Motion_in = Ball_Y_Motion;
```

The conditional statements before this else if statement handle collision handling with the platform and thus make sure gravity doesn't pull the character of the platform. The reason the else statement exists is so that gravity will be turned off when the character is touching the ground. Without it, there is collision handling with the ground, but the objects will pulsate, because they will constantly be reset to be above the ground and have their Ball_Y_Motion = 0, in every frame. Since 1 is added for every frame that the character is not on the platform or the ground, gravity will seem realistic, since the character will slow as they reach the peak of their jump and then fall back to the ground at increasing speed.

Sprites:

The sprites we used for our game pertained to a background, a platform, a character, and an enemy. We found all of these images from free sources online. We used On-Chip memory to hold the data, and we used color palettes, instead of storing the RGB data directly. To convert our images to hex we used **Rishi's ECE 385 Helper Tools**, which we found on the course website. The way we knew to draw a certain object was by sending a signal similar to how is_ball was sent in lab8. The main difference is that all our objects are rectangles in the game, so our condition for all objects was:

```
(DrawX_plus_progress >= Ball_X_Pos && DrawX_plus_progress < Ball_X_Pos + SizeW && DrawY >=
Ball_Y_Pos && DrawY < Ball_Y_Pos + SizeH )
```

More on DrawX_plus_progress in the scrolling section, but if the game had no scrolling it could have been replaced with DrawX. If this condition was true we would send a is_character, is_enemy, or is_platform signal.

The way we knew where to index into the ROM of each object was, we created was by using the general format of (DrawX - objectrXposition) + ((DrawY - objectYposition) * (spritesheetWidth)). Object X and Y position were sent to Color_Mapper.sv from every object instantiation. The background ROM was a bit different in that it indexed by DrawX + progress + (DrawY * 10'd640), where 640 was the width of its spritesheet. Progress was added to the index of the background ROM, to give the illusion that the background was scrolling with the character.

The way we handled if multiple objects intersected the same pixels on the screen was a chain of if and else if statements. If the else statement was reached, the background would be shown.

The way we handles transparent portions of a sprite was to color it with some garbage color, and assign it the palette index 0. That way even if it is_object condition was true it would fail the condition of whether to display that pixel.

Animations:

We had walking animations for the character and the enemy. Based on the direction of their movement we 4 sprites, so 8 sprites in total for each. The way we would determine which frame of walking to show was based on if the object was moving and a counter. If the object is moving a 8 bit counter increments on the frame clock, the counters are split into 4 sections, and based on which section the counter is in, that frame will be shown.

Since we have to account for direction and object frame when doing the animation their ROM access doesn't follow the simple format. And instead follows:

```
(DrawX - characterX) + ((DrawY - characterY) * 7'd64)+ offset + (characterFrame * 10'd16);
```

This is an example based on the character ROM's index. Offset accounts for direction.

Scrolling:

Scrolling is done by setting walls within the frame of the screen, and checking for if the character is colliding into one of those walls. If the character is colliding in the right wall, we have a logic called "progress" increment, and if the character is colliding with the left wall "progress" will decrement. Progress has a limit of one screen size 640 pixels, so there is a limit on the gamespace. The DrawX_plus_progress, which was mentioned earlier is what allows the game to scroll. Since without it, there is no way for our projects drawing system to know where the character is in the gamespace.

Collision Detection:

There are four kinds of collision detection in our game and all of them have to do with the character. Collision with the ground due to gravity, collision with the bottom of the platform, collision with the top of the platform, and collision with the enemy.

Collision with the ground is similar to lab8, except in the project we change Y max the ground indicated in our background sprite, and we take out the bounce of the bottom code. For a more detailed description of collision with the ground check the Gravity section or Comprehensive Module Description of character.sv.

Collision with the bottom of the platform and above the platform is done similarly. The X,Y position of the character is sent to the platform module and based on some arbitrary offsets and the position and dimension of the platform, a box is mapped out bellow and above the platform. If the character enters one of those boxes, then a bellow platform or above platform signal will be sent to character.sv. And the character will bounce of the bottom, or land on the top respectively.

Collision with the enemy is done by sending the X,Y position and the X motion of the enemy to the character.sv module. Here a collision is detected by mapping out a box based on the dimensions of the character and the enemy and their movement. And if the character entered that box, the game will end, and all characters will stop moving. Below is an example of how a collision box is set:

```
if( ((Ball_X_Pos + Ball_X_Motion) <= (4'd10 + enemyX + progress + enemyXmotion)) && ((Ball_X_Pos + Ball_X_Motion+4'd10) >= (enemyX + progress + enemyXmotion)) && ((Ball_Y_Pos + Ball_Y_Motion) <= (4'd10 + enemyY)) && ((Ball_Y_Pos + Ball_Y_Motion+4'd10) >= enemyY))
                                        dead_ = 1'b1;
else
                                        dead_ = 1'b0;
```

Comprehensive Module Description:

Module: lab8.sv
Input: CLOCK_50, OTG_INT, [3:0] Key
Output: DRAM_RAS_N, DRAM_CAS_N,DRAM_CKE, DRAM_WE_N, DRAM_CS_N, DRAM_CLK, VGA_CLK, VGA_SYNC_N, VGA_BLANK_N, VGA_VS, VGA_HS, OTG_CS_N, OTG_RD_N, OTG_WR_N, OTG_RST_N, [1:0] OTG_ADDR, [1:0] DRAM_BA, [3:0] DRAM_DQM, [6:0] HEX0, [6:0] HEX1, [7:0] VGA_R, [7:0] VGA_R, [7:0] VGA_B, [12:0] DRAM_ADDR
Inout: [15:0] OTG_DATA, [31:0] DRAM_DQ
Description: Top level of our final Project.
Purpose: It has two main purposes. It connects the hardware and the software, and it also connects modules that needed to be connected.

Module: character.sv
Input: Clk, Reset, frame_clk, w_on, s_on, a_on, d_on, above_platform1, bellow_platform1, [9:0] DrawX, [9:0] DrawY, [10:0] enemyX, [10:0] enemyY, [10:0] enemyXmotion
Output: is_character, dead, [10:0] characterX, [10:0] characterY, [10:0] progress
Description: This module contains logics for our main character of the game.
Purpose: The module has logics for basic movements, movements on platform, gravity, scrolling, and death of the character. For basic movement we made the character move in X coordinate when keycode A or keycode D is pressed, but not while it is jumping. We only allowed our character to jump when it is above the platform, or on the ground, basically not allowing jumping in midair. We applied gravity when our character is not on ground or when it is not on platform. By adding one to the “Ball_Y_Motion” took care of it. In this module we also have logic for scrolling. We created a variable named “progress”, which act as a counter for the X coordinate movement of the character. The “progress” is incremented by size

of the step of the character, when character is at X = 550 and moving to the right. We decremented the "progress" when "progress" is positive, X = 50 and moving to the left. Adding this variable to "DrawX" allows us to draw at the character at the right position, and subtracting the variable from "Ball_X_Pos" allows to keep where our character is in the screen. We determine the death of the character in this module as well, by saying that our character dies when we enemy and character have collision in the next cycle. When the dead signal is on, we stop the motion so it will stop the motions when character and enemy collides.

Module: enemy.sv
Input: Clk, Reset, frame_clk, dead, [9:0] DrawX, [9:0] DrawY, [10:0] progress
Output: is_enemy, enemyDir, [10:0] enemyX, [10:0] enemyY, [10:0] enemyXmotion
Description: This module contains logics for enemy of the game, and creates the enemy.
Purpose: The module is very similar to the character.sv, but instead of the input from keycode we have set the movement of the enemy in this module. As character.sv we added the "progress" created from character.sv to "DrawX" and subtracted the value from "Ball_X_Pos", to synchronize the scrolling to the enemy.

Module: platformTile.sv
Input: Clk, Reset, frame_clk, dead, [9:0] DrawX, [9:0] DrawY, [10:0] progress, [10:0] characterX, [10:0] characterY
Output: is_platform, above_platform, bellow_platform, [10:0] platformX, [10:0]platformY
Description: This module creates a platform object.
Purpose: Using the "progress" created from character.sv and using the same logic as all other objects that scrolls, we make the platform to perform synchronized scrolling. In this module we have variables that determines if the character is above the platform or below the platform. We made two rectangle shapes above the platform and one below the platform. If lower rectangle detects character in its parameter, we send a signal to character.sv so that it can bounce down, when character hits the bottom of platform. When upper rectangle detects the character we made the character to stop Y motion, so it can be on the platform. By having detecting parameters set up, we can easily create numerous platforms for the game.

Module: Color_Mapper.sv
Input: is_character, Clk, enemy1, frame_clk, dead, a_on, d_on, enemyDir1, platform1, [9:0] DrawX, [9:0] Draw Y, [10:0] characterX, [10:0] characterY, [10:0] enemy1X, [10:0] enemy1Y, [10:0] progress, [10:0] platform1X, [10:0] platform1Y
Output: [7:0] VGA_R, [7:0] VGA_G, [7:0] VGA_B
Description: This module allows us to set color for each pixel.
Purpose: We draw background, character, enemy, and tiles by using this module and sprite sheets that we have created. For X coordinate of each of them we use "Ball_X_Pos" subtracted by "progress", which is the of the object position on the screen. However, for the background we added the progress on the X coordinate so it can repeat itself without any calculations. This module also take cares of the animation. We created frames, counters, and offsets for objects that needed the animation. We used frames for one directional movement left or right. By using the counter that increases each rising "frame_clk" and we made it switch the frame of character or enemy every 8 frame_clk cycles, so that it can perform animation. The offset, which is one bit variable, changes when the direction of character changes. When

offset is on, it uses the lower sprites on the sprite sheet. Difference between character and enemy is that for character counter goes back to 0 when there are no movement input to the character, which make the character to be in the original sprite. However, offset of the character does not change to keep the character to look where it was looking at before it stopped.

Module: backgroundROM.sv
Input: [19:0] background_address, Clk
Output: [7:0] VGA_R, [7:0] VGA_G, [7:0] VGA_B, [3:0] data_Out
Description: Module for the background drawing (640 x 480).
Purpose: It reads a txt file that has information about the background we are trying to draw. Based on the data pulled out from the txt file, we set the color for that pixel using "VGA_R", "VGA_G", and "VGA_B".

Module: spritetile.sv
Input: [19:0] tile_address, Clk
Output: [7:0] VGA_R, [7:0] VGA_G, [7:0] VGA_B, [3:0] data_Out
Description: Module for the tile drawing (85 x 21).
Purpose: It reads a txt file that has information about the tile we are trying to draw. Based on the data pulled out from the txt file, we set the color for that pixel using "VGA_R", "VGA_G", and "VGA_B".

Module: spritesheet.sv
Input: [11:0] character_address, Clk
Output: [7:0] VGA_R, [7:0] VGA_G, [7:0] VGA_B, [3:0] data_Out
Description: Module for the character drawing (64 x 48).
Purpose: It reads a txt file that has information about the character we are trying to draw. Based on the data pulled out from the txt file, we set the color for that pixel using "VGA_R", "VGA_G", and "VGA_B". The "data_Out" is sent to Color_Mapper.sv, and if the variable is 0 we set it to be "is_transperent" and do not draw "VGA_R", "VGA_G", and "VGA_B" that came out from this module. Instead we fill those "VGA_R", "VGA_G", and "VGA_B" from another sprite.

Module: spriteenemy.sv
Input: [11:0] enemy_address, Clk
Output: [7:0] VGA_R, [7:0] VGA_G, [7:0] VGA_B, [3:0] data_Out
Description: Module for the enemy drawing (64 x 48).
Purpose: It reads a txt file that has information about the enemy we are trying to draw. Based on the data pulled out from the txt file, we set the color for that pixel using "VGA_R", "VGA_G", and "VGA_B". The "data_Out" is sent to Color_Mapper.sv, and if the variable is 0 we set it to be "enemy_transperent" and do not draw "VGA_R", "VGA_G", and "VGA_B" that came out from this module. Instead we fill those "VGA_R", "VGA_G", and "VGA_B" from another sprite.

Module: keycodeReader.sv
Inputs: [31:0] keycode, [2:0] Key

Outputs: w_on, a_on, s_on, d_on
Description: This module gets the values from the USB keyboard and creates an output that replaces those keycodes.
Purpose: Using this module allows us to read more than 2 key presses on the keyboard, and also made debugging faster by placing KEYs from DE-2 115 to output variables.

File name: main.c
Purpose: The main software function for this project. Calls functions in usb.c and io_handler.c. It connects NIOS II with the keyboard. It sets up configuration, get descriptor, and get keycode value.
IMPORTATNT NOTE: This file is largely unchanged from lab8, but we changed our keycode PIO to take 32 bits and can take 4 keystrokes. This was done by adding the lines:

```
IO_write(HPI_ADDR,0x0520); //the address of byte 2~3
```

```
keycode += (IO_read(HPI_DATA) << 16);
```

and editing the line:

```
printf("\nfirst two keycode values are %04x\n",keycode);
```

to:

```
printf("\nfirst four keycode values are %08x\n",keycode);
```

Note: We include general descriptions for the modules below since they are the same for lab 8

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: It uses counter for vertical and horizontal syncs to keep track of the beam.
Purpose: Produces vertical and horizontal syncs for VGA monitor. Same as lab8.

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, OTG_INT
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: It connects PIOs from NIOS to the CY7C67200 chip using tristate data bus.
Purpose: Interface between NIOS II and CY7C67200 chip. Same as lab8.

Module: final2_soc.v
Inputs: clk_clk, [15:0] otg_hpi_data_in_port, reset_reset_n,
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
Inouts: [31:0]sdram_wire_dq

Description: It connects the PIOs in NIOS to the hardware. It is generated from NIOS II system and the outputs are exported. It uses otg_hpi PIOs, and keycode to get data from the USB. It computes the data and exports it out to the hardware. JTAG UART implements a method to communicate serial character streams between PC and an SOPC Builder on FPGA.
Purpose: It connects the top level of the lab to NIOS system. Same as lab8.

Module: HexDriver
Inputs: [3:0] In0
Outputs: [6:0] Out0
Description: Changes 4 bit binary as 1 bit hex on DE-2 Board.
Purpose: Displays Hex values on DE-2 Board. It was used for debugging the USB keyboard. Same as lab8.

File name: usb.c
Purpose: This file includes the functions for usb to properly work. Important functions are UsbWrite and UsbRead, which calls on IO_write and IO_read. UsbWrite writes the address to HPI_ADDR and write the data into HPI_DATA. UsbRead writes the address to HPI_ADDR and read the data in the address from HPI_DATA. Same as lab8.

File name: io_handler.c
Purpose: It hold the helper function for UsbWrite and UsbRead, the IO_write and IO_read. By using otg_hpi values, it makes both reading data and writing data possible. IO_write receives the address and writes the data on otg_hpi_data. IO_read receives the address and reads the data from otg_hpi_data. Same as lab8.

Design Resources and Statistics:

LUT	3628
DSP	10
Memory (BRAM)	1,271,912 bits
Flip-Flop	2,329 registers
Frequency	78.57MHz
Static Power	102.26 mW
Dynamic Power	49.53 mW
Total Power	225.81 mW

Conclusion:

In the end the final project was not all that we hoped it would be, but it was still functioned within the baseline degree we set. And along the way we gained many valuable experiences. The fact we had to create a project with no set structure, was a first for both of us helped us improve as engineers and developers in general. Dealing with buggy tools and long compile times helped us grow in terms of our debugging skill set. For example, after a week of working on the project we had created multiple Quartus projects where we tested different parts of our entire game. One project had generic shapes and tested basic game logic, one had all working game logic with sprites and animations, but no keyboard, and the last project had everything included. At times this project was frustrating, but well worth it in the end.