ECE 385
Spring 2019
Experiment #10

The Final Project

Arkin Shah (arkinas2)
Brian Zheng (brianjz2)
Thursday 8am (ABL)
TA: Vikram Anjur
Xinying Wang

Introduction:

For our final project, inspired by the classic Mummy Maze game developed by PopCap Games, we decided to replicate a singular level that is in the game. The game utilises the “WSAD” keys as user input through the keyboard to move the character in the four cardinal directions. We chose this game as it alludes back to our childhood and is quite nostalgic. It made us feel proud to now create something we spent so much of our childhood over.

Written Description:

For this final assignment, the game we created drew inspiration from plenty of movies (such as Indiana Jones) combined with strategy based games. At its core, our game can be thought of as one piece chess. Implementing an entire chess game would’ve been ideal in our opinion, but we have a knowledge gap when it comes to implementing the artificial intelligence required, and hence it is beyond our scope. However, this game is an implementation of one-piece-chess where the opponent makes two moves for every move the player is allowed to make, this makes the game challenging, or else it would have simply been a game of one piece checkers. The background for the game looks exactly like a chessboard, although more vintage; the hero (you) begins at the bottom right square, whereas the mummy (opponent) begins at the golden goal escape tile, which is the desired final destination of the hero. The hero loses if the mummy catches (i.e both the hero and the mummy’s positions coincide), and the hero wins if the hero makes it to the golden goal. The game is purely strategy based and ideally the way to win is to trap the mummy at a “wall” which is a part of the chessboard playground, these can be thought of as obstructions. If the mummy encounters a wall and the shortest path to the hero is "through" the wall, the mummy loses that turn. Hence, for example now it gets one turn. If you trap it extremely strategically, you might cause it to lose both its turns. If the hero makes it out alive, if the hero wins.

The FPGA is used to implement almost everything besides the key-code handler through the USB port. Our on-chip memory is used to hold the color mapper for the entire background, which is created by setting the specific RGB values for the coordinates that were calculated by hand and implemented in hardware using multiple if-else cases depending on their coordinates.. The movement for the player was handled by accepting a keyboard press (handled using C code), which was then communicated through to the hardware using the NIOS II bus. The hardware then used this key press to move the hero in the desired direction, of it is possible, as this is also when it checks for boundary conditions. Depending on the hero's move, the artificial intelligence we implemented decides its next move through its algorithmic implementation in hardware. It makes coordinate based calculations and changes for each move the player makes.

It is worth noting that the USB code used in this final project is very much the same from the previous lab8. Refer to the lab8 report for more details about the implementation of the USB protocol.

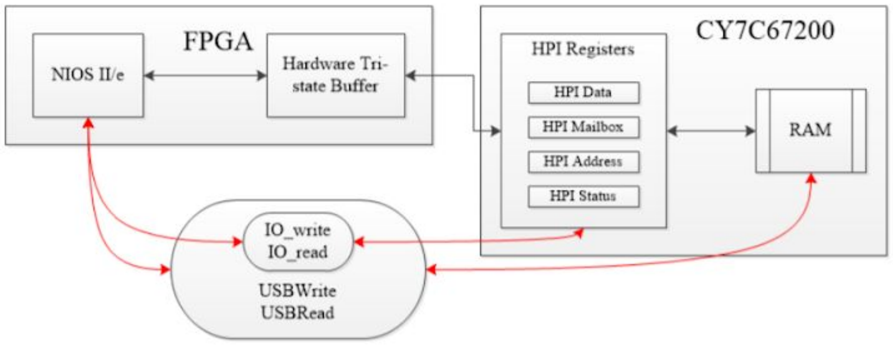


Figure 1: Block Diagram for the Interactions Between USB Functions, NIOS II, and the CY7C67200 chip

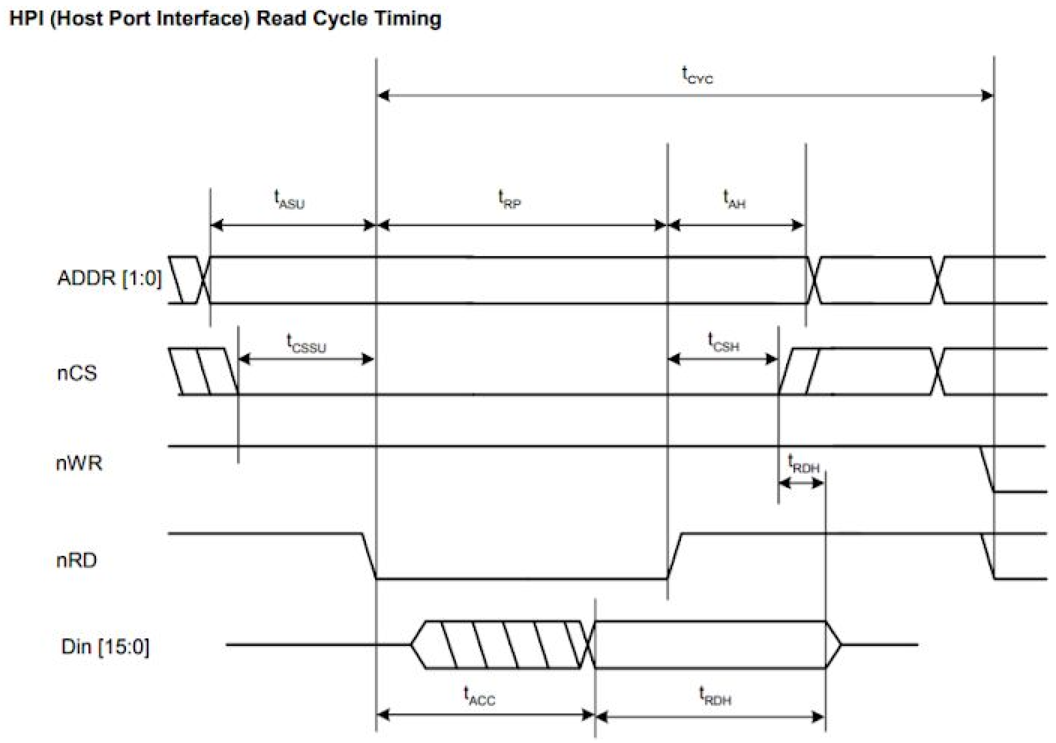


Figure 2: HPI read cycle timing diagram given by CY7C67200 Data Sheet

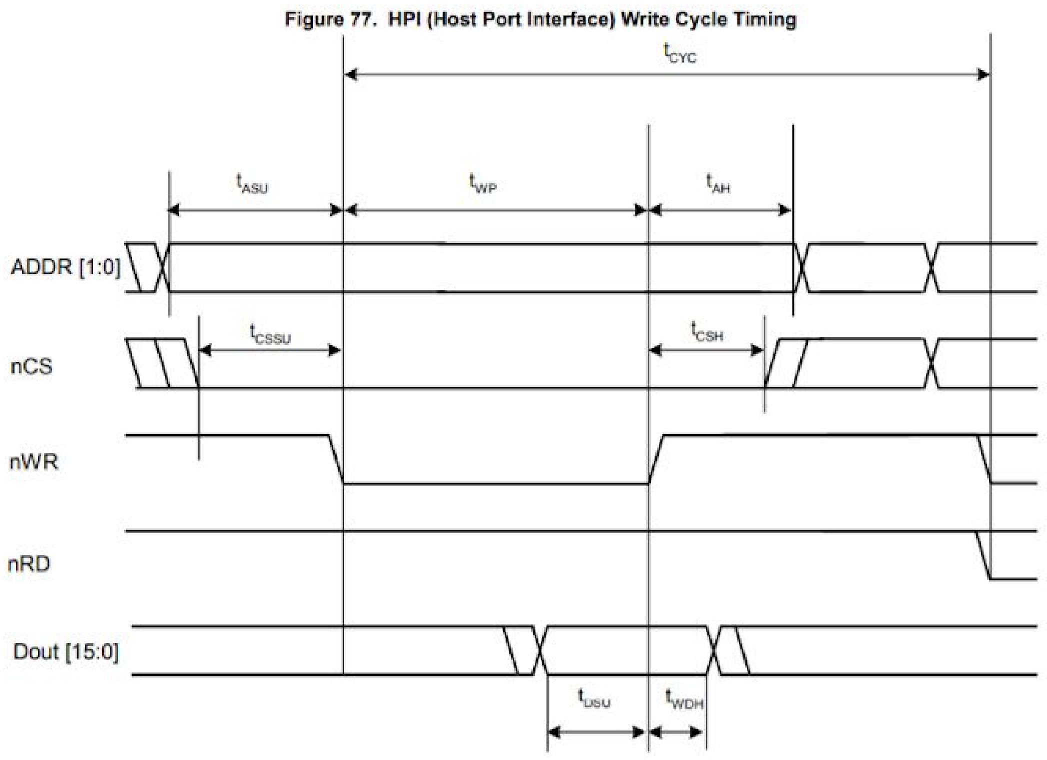


Figure 3: HPI write cycle timing diagram given by CY7C67200 Data Sheet

Ball:

Unlike the lab8 ball module, this ball does not bounce and always moves a set position per key press in a direction dictated by the key pressed. Say we want to prevent the ball from moving out of the stage. This would be determined by something like for a wall:

```
" if(Ball_X_Pos_in>=(10'd422-Ball_Size) && Ball_X_Pos_in<=(10'd523+Ball_Size) &&
Ball_Y_Pos_in>=(10'd138-Ball_Size) && Ball_Y_Pos_in<=(10'd188+Ball_Size))
        begin
            boundary = 1'b1;
          end"
```

and

```
"if( (Ball_Y_Pos_in + Ball_Size) >= Ball_Y_Max)
          begin
             boundary = 1'b1;
          End"
```

for the edge of the board.

Namely, what "boundary" does is prevent the position of the ball from changing and thus clipping through a wall or boundary.

Once the ball's keyboard press has been detected, the ball will send a "turndone" signal to the FSM to signify that the player's turn has ended, allowing the states to continue.

Mummy:

The mummy module is very similar to the ball module, except it sends out two turndone signals for the FSM, since the mummy must make two moves. The algorithm, which will talked about more in detail below, determines the value of button in for the mummy module. After the first "turndone" signal is sent by the mummy, the FSM will send another signal to have the mummy do its second move (done through mummyAI module). Once the second turn is done and the mummy module sends its second "turndone" signal to the FSM, it will move on to the next state.

Color Mapper:

The Color_Mapper module, which is responsible for handling the output of the FPGA to the VGA, is responsible for displaying the various screens, and the mummy. Extensively modified from lab8, the color mapper, based on what the FSM tells it, will either display the game or one of three screens. During the game, because the FSM will always display a value of 1 for the game control signal, the Color Mapper will display the game as is. Meanwhile, should an input signal such as winner or gameover be tripped from the FSM, the color-mapper will subsequently change the screen to reflect what the if statements appropriately state.

In regards to the calculations, of the letters from the font rom, we determined that our implementation of the letters required this equation. Say we wanted to acquire the letter R at x052 in the fontRom, our sprite_addr would be:

sprite_addr = (DrawY-R1y + 16*'h52);

The two 11-bit long logics for each individual letter determines the position we want on the x and y coordinates. We can use the control signals from the FSM to determine whether to display the letter or not.

It is worth noting that our implementation required that we flipped the order of the respective 8 bit register within font rom. Due to the manner of calculating, our module would keep reading the font_rom from right to left instead of left to right.

FSM:

Our FSM has a total of nine states within. It controls six outputs, three of which are fed into the color_mapper to control the title screen, win screen, and death screen. Three other outputs are fed into the ball (one total) and mummy modules (two total). These inputs, depending on the state, will deliver allow the player or the mummy to make a turn. In addition, the FSM also receives three inputs from the ball (one total) and mummy (two total) modules that tell the FSM when the mummy or player has completed making a move and can go into the next state. In addition to taking in/out the inputs/outputs from/to the player and mummy modules, the FSM also monitors the x-y coordinates of both the player and mummy in-game. Specifically, the three positionCheck states that occur between after each individual player or mummy turn, detects whether the player and mummy are in the same position, or whether the player has reached the goal. The FSM, during those PositionCheck states, takes in two inputs to see whether to trigger the gameover or winner screens. These inputs come from the gameoverCheck and winnerCheck modules. The state diagram for this FSM can be found later in this lab report.

AI Algorithm:

Used by the mummyAI module, this is the brains of the program. Because the mummy module must accept a keycode in order to move, the mummyAI is responsible for sending in the keycode (button) in order to have the mummy make its move.

This algorithm is implemented in the algorithm module, it is used to determine which keys the mummy should press. It is entirely dependent on both the location of the player and the mummy, and the keyboard that is pressed by the player.

What's considered:
The current position of the mummy's pixel coordinates
The player's pixel coordinates (current position)

FIRSTTURN:

Check what key is pressed by the player:

1. If A or D is pressed, compare x coordinates
2. If W or S is pressed, compare y coordinates

What happens if - or + x change is detected from player:
(Enemy X < PlayerX, enemy makes a +x direction movement (ignores wall)
(Enemy X > PlayerX, enemy makes a -x direction movement (ignores wall)

What happens if - or + y change is detected from player
(Enemy Y < PlayerY,enemy makes a +y direction movement (ignores wall)
(Enemy Y > PlayerY, enemy makes a -y direction movement (ignores wall)

RUN POSITIONCHECK

SECONDTURN
Make a change in the other position (for example, if player makes a x change, the enemy's second turn will be a y change)

Player's press was a -or+ x change

(If the Player’s Y Coordinate after input is == enemy’s Y coordinate), the enemy’s second turn will be the same as the first turn (ignores wall)

Player’s press was a - or + y change
(If the Player’s X coordinate after input is == enemy’s X coordinate), the enemy’s second turn will be the same as the first turn (ignores wall)

OTHERWISE
Player press was a -or+x change
(Enemy Y < PlayerY, enemy makes a +y direction movement (ignores wall)
(Enemy Y> PlayerY, enemy makes a -y direction movement (ignores wall)

Player press was a -or+y change
(EnemyX < PlayerX, enemy makes a +x direction movement (ignores wall)
(EnemyX > PlayerX, enemy makes a -x direction movement (ignores wall)

RUN POSITIONCHECK AGAIN
SV Modules Description:

Module: lab8.sv

Inputs: CLOCK_50, [3:0] KEY, OTG_INT,

Outputs: [6:0] HEX0, [6:0] HEX1, [6:0] HEX2, [6:0] HEX3, [6:0] HEX4, [6:0] HEX5, [6:0] HEX6, [6:0] HEX7, [7:0] LEDG, [7:0] LEDR
[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

Inouts: [15:0] OTG_DATA, [31:0] DRAM_DQ,

Description: Top-level hardware module for the entire program (Modified from Lab8).

Purpose: Instantizes the ball, mummy, artificial intelligence, FSM, ColorMapper, hpi_io, hexdriver, VGA_Controller, gameover check, winner check, and the NIOS II SOC file for the keyboard and the game program to run.

Module: ball.sv

Inputs: Clk, Reset, frame_clk, Run, go, [3:0] button, [9:0] DrawX, [9:0] DrawY

Outputs: is_ball, turndone, [9:0] XPos, [9:0] YPos

Description: Provides the boundary, location, and movement conditions for the player, as well as detect when the player has made a move.

Purpose: This module defines how the ball can move (the one-tile movement), and also provides the boundary conditions for both the walls in the map and the edge of the stage. The is_ball is used by Color_Mapper to map out the ball's location and the XPos and YPos are used in various modules. The turndone is used to detect when a ball's position has moved (when a player inputs the keyboard press). Without this module, the player cannot play the game.

Module: mummy.sv

Inputs: Clk, Reset, frame_clk, Run, go, go2, [3:0] button, [9:0] DrawX, [9:0] DrawY

Outputs: is_ball, turn1done, turn2done, [9:0] XPos, YPos

Description: Very similar to the ball.sv, but this time for the mummy or computer. Unlike the ball.sv, which only has one turndone signal, this has two, to help the FSM know when the mummy has made two moves.

Purpose: To implement the move and boundary conditions for the mummy. Without this, the mummy, and thus the game, will not function properly.

Module: FSM.sv

Inputs: Clk, Reset, Run, playdon, mumdon1, mumdon2, win, loss

Outputs: playerSignal, mummySignal, mummySignal2, opening, gameover, winner, game

Description: State machine for the game logic.

Purpose: The brain that connects all the various modules in this program. It regulates both the player and mummy turns through the done inputs and win and loss inputs, as well as all three player/mummysignals, and using the opening, gameover, game, and winner outputs, tells what the color mapper should output. This is necessary to ensure that all the modules in this program can properly interact with each other

Module: color_mapper.sv

Inputs: is_ball, is_mum, [9:0] DrawX, [9:0] DrawY, [9:0] playerX, [9:0] playerY, [9:0] enemyX, [9:0] enemyY, start, win, death, game

Outputs: VGA_R, VGA_G, VGA_B

Description: Maps out the display for the screen. Used to display either the game, win screen, death screen, or start screen.

Purpose: Through the one bit input signals (start, win, death, game), the color_mapper will determine whether it needs to display the game board, or either screens. When it displays the game board, color_mapper is also responsible for mapping out the current location of the ball and mummy on the board, as well as the various colored walls.

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: Module for the Nios II and EZ_OTG chips (Copied from Lab 8)

Purpose: Bridges the gap between the NIOS2 and CY7C67200. The HPI registers within the file have to communicate with the NIOS II. The occurrence of read and write from the chip memory occurs and is implemented here

Module: VGA_controller

Inputs: Clk, Reset, VGA_CLK,

Outputs: VGA_BLANK_N, VGA_SYNC_N, VGA_HS,VGA_VS,
[9:0] DrawX, [9:0] DrawY

Description: Renderer for the game (Copied from Lab 8)

Purpose: This module is used in conjunction with the Color_Mapper, ball, and mummy to create the overall game. It primarily is used to implement the FPGA's ability to display an image on an external monitor.

Module: font_rom.sv

Inputs: [10:0] addr

Outputs: [7:0] data

Description: Font Rom used by the color mapper to map out letters for the title screen (copied from the font_rom file given by the final project page on the ECE 385 site. (Albeit slightly modified).

Purpose: Used by color mapper, it maps out the letters and respective addresses to each symbol. Since we want to display letters on the title screen, the color_mapper, with the appropriate input being fed into it from the FSM, will display the desired letter by reading from the appropriate address.

Module: gameoverCheck

Inputs: [9:0] playerX, [9:0] playerY, [9:0] enemyX, [9:0] enemyY**,**

Outputs: endgame

Description: Module to check whether the player and mummy coordinates match (the player died)

Purpose: If the player X-Y coordinates match with the enemyX-Y coordinates, that means the mummy has caught the player and endgame is set to one. This tells the FSM to end the game and trigger the game over screen.

Module: winnerCheck

Inputs: [9:0] playerX, [9:0] playerY, [7:0] keycode (not used)

Outputs: winner

Description: Module to check whether the player has reached the winner square and to end the game.

Purpose: If the player X-Y coordinates is equal to (10'd141, 10'61), the center location of the golden square on the game board, winner is set to one, which is sent to the FSM to end the game and display the win screen.

Module: algorithm

Inputs: [7:0] playerKeycode, [9:0] playerX, [9:0] playerY, [9:0] enemyX, [9:0] enemyY,

Outputs: [7:0] enemyKeycode1, [7:0] enemyKeycode2

Description: Determines the keycodes for the mummy's two turns based upon what the player and mummy's locations and what the player pressed. Instantized in mummyAI.

Purpose: This algorithm is implemented so the mummy knows what moves it needs to make based upon what the player does in a certain location. Without this, the mummy would not move.

Module: mummyAI

Inputs: [7:0] playerKeycode, [9:0] playerX, [9:0] playerY, [9:0] enemyX, [9:0] enemyY, go, go2,

Outputs: [7:0] keyCode, [7:0] fort, [7:0] fort1

Description: Top level module to send the first and second keyboards for the mummy to move.

Purpose: Because the FSM has two turn signals for the mummy, which is fed into this module as go and go2, depending on what the signal is sent by the FSM, the output keyCode will be equal to either firstTurn or secondTurn from the algorithm module. Fort and Fort1 were used in the HexDriver for debugging purposes and is ultimately not used in the final product.

Module: lab8_soc.v

Inputs: accumulate_enable_export, accumulate_reset_export, clk_clk, [15:0] otg_hpi_data_in_port, reset_reset_n,

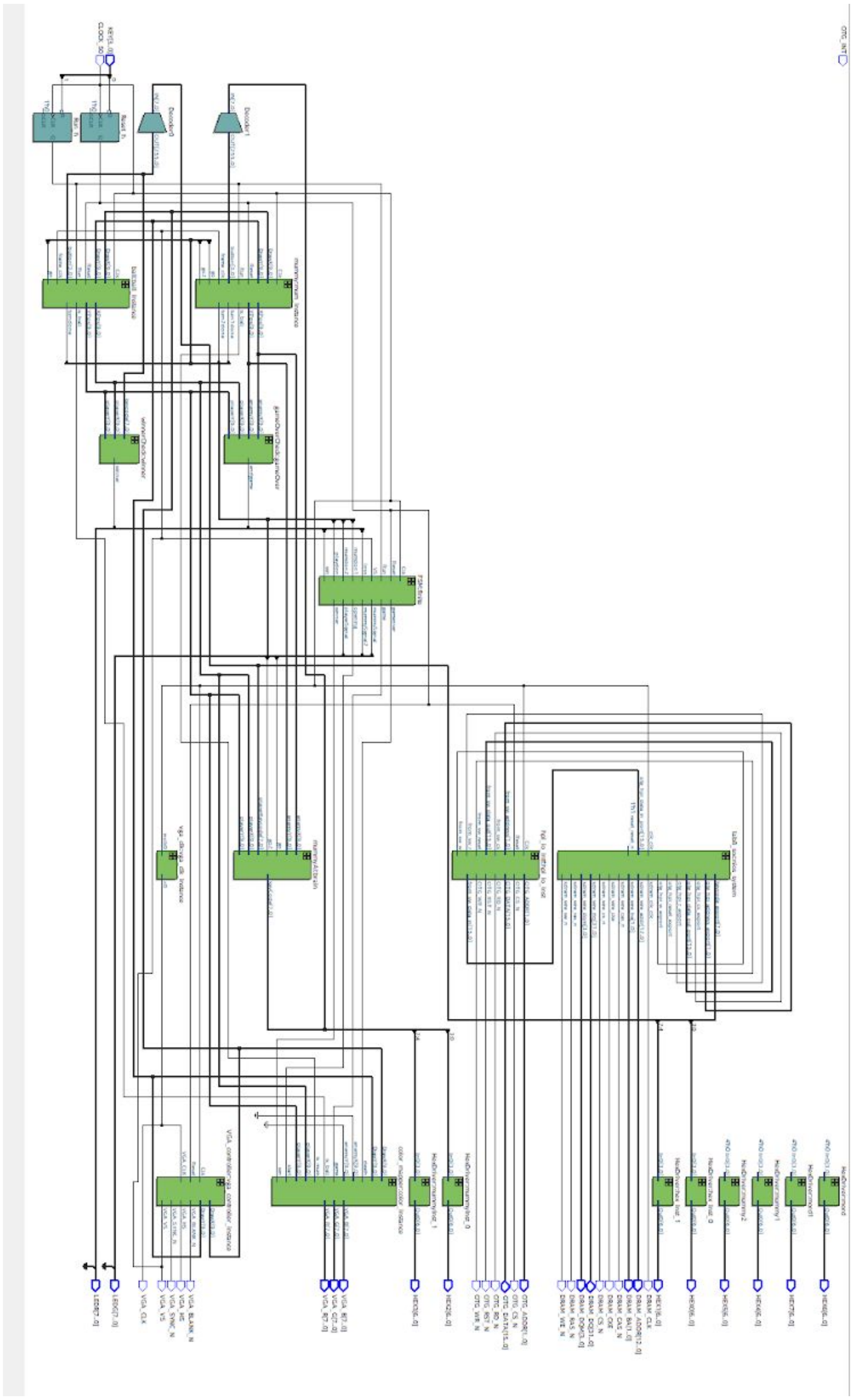
Inout: [31:0] sdram_wire_dq,

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

Description: Quartus implementation of the NIOS II SOC

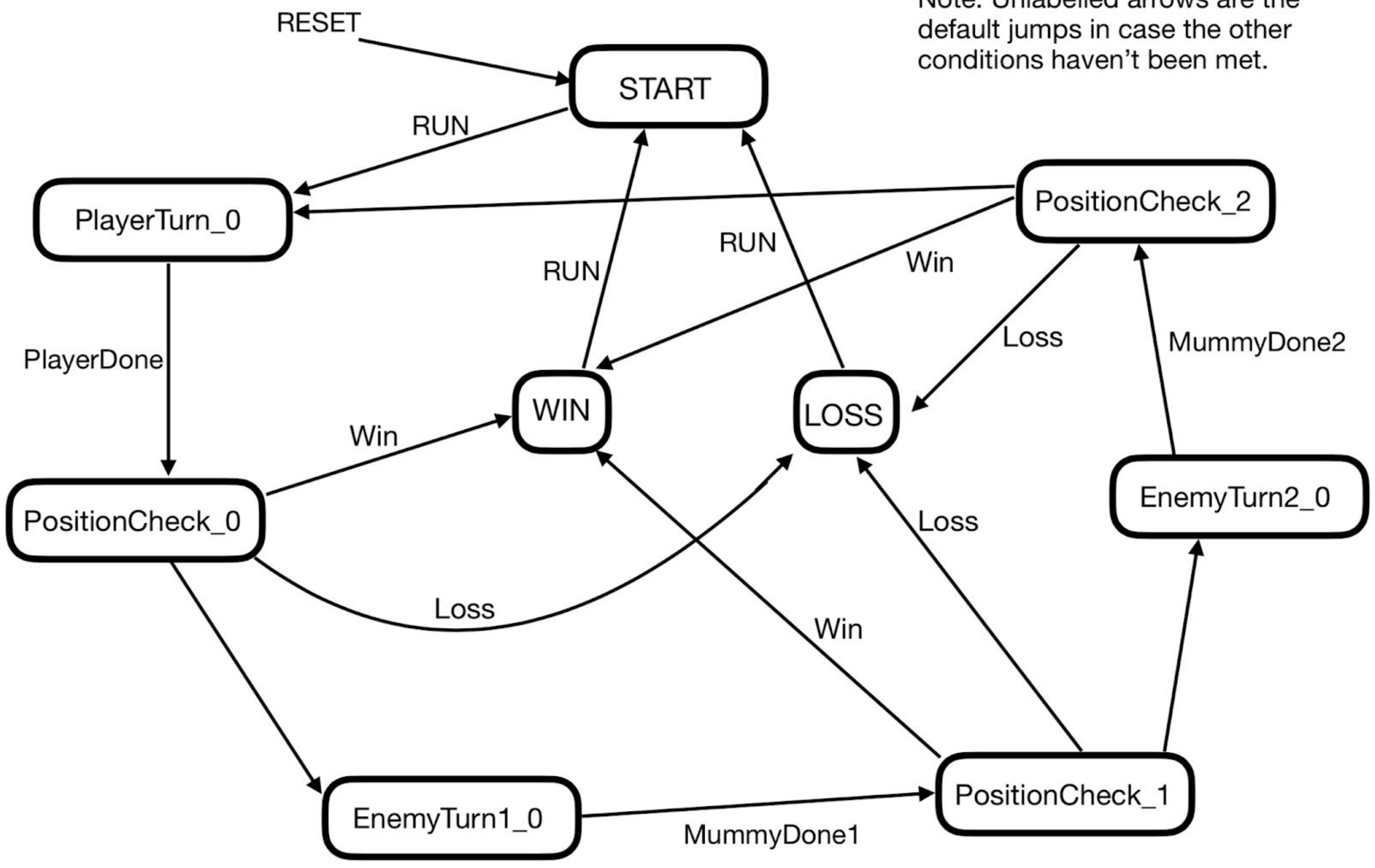
Purpose: The principal hardware code of the project. It allows for the software to be processed. Without it, the functions will not perform properly.

RTL Diagram:



State Machine Diagram:

Note: Unlabelled arrows are the default jumps in case the other conditions haven't been met.



QSF Table and Specified Address (Copied from Lab 8):

NIOS2_Gen2	0x0000_1000
On chip Memory	0x0000_0000
LED	0x0000_0090
SDRAM	0x1000_0000
SDRAM_PLL	0x0000_00a0
SYSID_QSYS	0x0000_00b8
Switches	0x0000_0080
Accumulator	0x0000_0070
Accumulator Reset	0x0000_0060
JTAG_UART	0x0000_00c8
Keycode	0x0000_00a0
Hpi_address	0x0000_0090
Hpi_data	0x0000_0080
Hpi_read	0x0000_0030
Hpi_write	0x0000_0040
Hpi_cs	0x0000_0020
Hpi_reset	0x0000_0050

Table 1: Elements of the QSF and their Specified Addresses

Resources Table:

LUT	4737

DSP	N/A
BRAM	55296
FLIP-FLOP	1614
FREQUENCY	36.73 MHZ
STATIC POWER	105.9 mW
DYNAMIC POWER	114.93 mW
TOTAL POWER	324.4 mW

Conclusion:

We could not implement the sprites in time and that was a big let down to both of us because having better graphics would have taken the game to a whole new level, however, we prioritised functionality and we would make this decision every single time if we had to do it again. The main hurdle we faced was getting the motion set in accurately because unlike our peers, our game did not implement slow and continuous motion. Our hero moved like a chess piece, it had discrete movement. This would often hinder with both, boundary and wall conditions. The conditions for these had to be hard coded, and to be honest, there were way too many. Just when we thought we covered all the cases, somehow we would end up in a never before seen situation and the game ended up glitching. Another huge issue we faced was with the flag signals that our modules were sending out and how the finite state machine reacted to it by sending out control signals. Everything seemed right on paper after drawing it out, however, in its implementation, things would quickly fall apart and even after consulting with multiple TAs we couldn't figure out the issue. Even according to the TAs, everything seemed correct, including looking through line by line and starting from scratch. Another less talked about issue is the compilation time; the programmes take ridiculously long to compile due to all the complexities and during that time, making progress is impossible. Even if it takes two minutes to compile, drop by drop it becomes an ocean of time simply spent waiting. There's nothing that can be done about this, but hours have been wasted simply waiting, once again adding to the time crunch.

It was crucial for us to pick something that we actively enjoyed creating because we came across many speed bumps along the way. After spending so much time on this game, we definitely have a deeper sense of appreciation for game developers, no matter how simple it is. It was a pleasure getting this project done mostly because of the freedom it came with. It became very clear that all the 9 initial experiments were simply examples to get us familiar with the tools that were available for use. The final project was open ended and hence allowed us to maximise our creativity.