Janish Yerra- jyerra2
Akhil Kandimalla-kandmll2

May 2nd 2017

ECE 385- Final Project Report

FPGA Pokemon Game

Introduction

For the final project, we have implemented the popular Nintendo game, “Pokemon”, in hardware-SystemVerilog. At the start of the game, the player is asked to choose between three starter pokemon, and the user would have to continue throughout the game with the same pokemon. Once the player selects the starter pokemon, the display changes to the first map where the user gets to control the pokemon trainer’s motion. While the pokemon trainer is in the dark green grass, he may find a wild pokemon that he can chose to fight. If the trainer chooses to fight the wild pokemon, he gets to attack the wild pokemon and gain experience points, which eventually leads to an increase in the pokemon’s level. The user can choose from three different levels of difficulty(easy, medium and strong) and three different types of Pokemon(mild, medium and legendary) using switches on the FPGA. During the fight, the user has the option of quitting the battle and going back to the map. The user can also heal his pokemon’s health by visiting the pokemon centre or a friendly rest center where the pokemon’s health would be replenished. The game ends when the user’s pokemon reaches level 15.

Summary of Operation

Almost the entire game was implemented in hardware alone(with the exception of the USB Host Controller for the Keyboard) which was quite a challenge, given that we needed to perform various arithmetic operations, random number generation for the wild pokemon and their frequency of appearance and memory registers which stored or reloaded the unique ID of User Pokemon or Wild Pokemon.

We implemented a module called ‘poke_battle’ in which we defined various other modules which enabled us to properly carry out both the attacks of the User and the Wild Pokemon respectively and output the Health Points onto the screen. While each Pokemon could only attack with a single attack, each Pokemon had a definite type(Leaf, Fire, Water, Thunder, Flying, Rock, Fighting or Psychic) and a

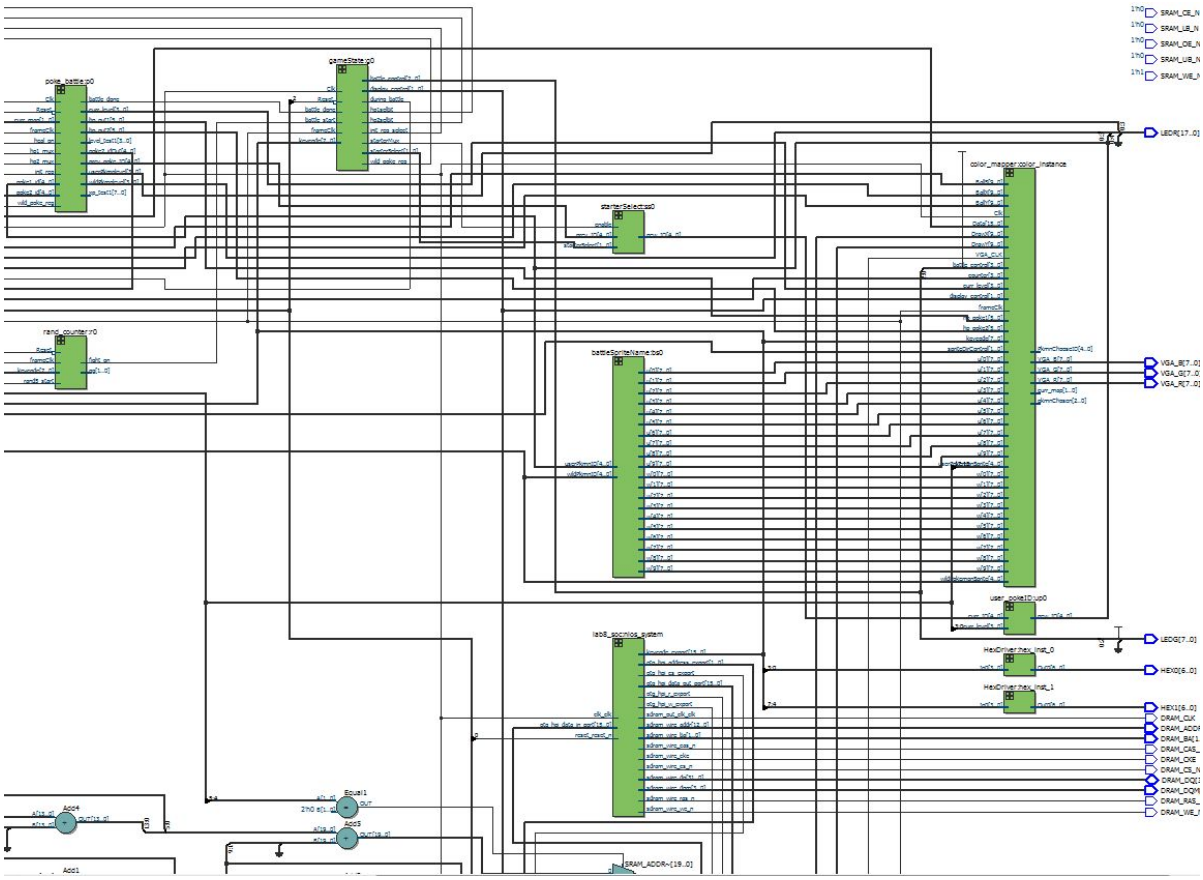
level which uniquely defined the damage that they would affect to the other Pokemon or be subject to.

In order to save or reload the registers, we used combinational logic in conjunction with the Finite State Machine to set the control signals such that we could replicate the exact functioning of a handheld Pokemon game. The three maps the user could explore had different types of Pokemon with varying levels of power just like a normal Pokemon game. Collision handling was implemented by utilizing their coordinates on the map. After every successful battle, the Pokemon gained a level if it receives adequate experience points and might even evolve.

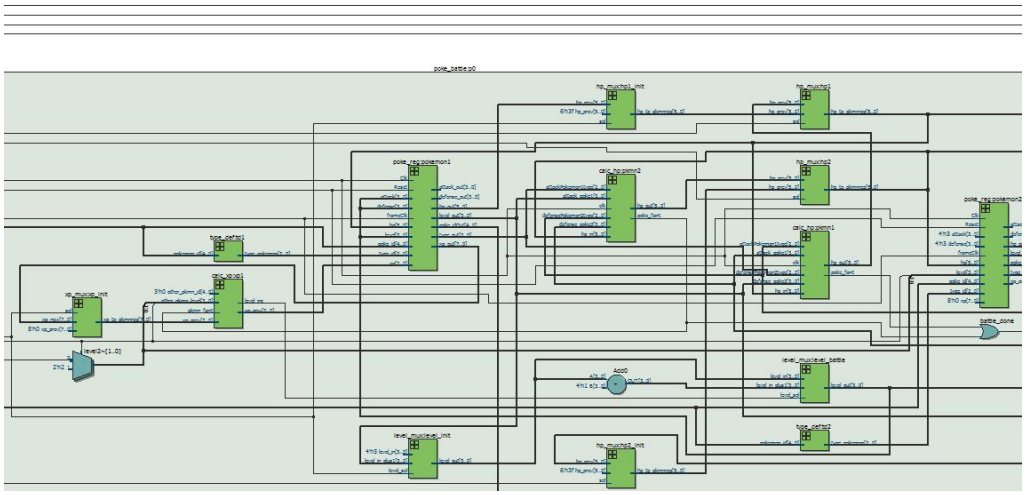
SRAM and On-Chip Memory Usage

Our implementation of the Pokemon game relied heavily on the on-board SRAM and the On-Chip Memory to provide fast memory access to draw images and whole maps onto the screen. We had four images - (The start screen, Map1, Map2 and Map3) on the SRAM which were addressed appropriately according to the control bits set by the user. The On-Chip Memory contained all the Pokemon Sprites which were uniquely identified by their Pokemon ID.

Block Diagrams



High level block diagram of the logic implemented.
NOTE: Please zoom in to see individual modules.



Poke_battle module designed completely from scratch. NOTE: Please zoom in to see individual modules.

SV module description:

Module: gameState
Input: Clk, Reset, frameClk, battle_done, battle_start, [1:0] display_control, [2:0] battle_control, [7:0] keycode, hp1selbit, hp2selbit, init_reg_select, during_battle, Wild_poke_reg
Output: [1:0] starterSelect, starterMux
Description: This module enables the game to switch between states and also controls the Battle FSM which is instantiated inside it.
Purpose: Switches the game between various states.

Module: battleSpriteName
Input: [4:0] userPkmnID,[4:0] wildPkmnID
Output: u, w
Description: This takes in the Pokemon IDs of the user Pokemon and the wild Pokemon and outputs their data to the font rom.
Purpose: This is used to print the name of the battling user and wild Pokemon.

Module: user_pokeID
Input: [4:0] curr_level, [5:0] curr_ID
Output: [4:0] new_ID
Description: Assigns a pokemon register a new value according to the previous ID and the current level.
Purpose: This is used to ensure that Pokemon evolve once they reach a particular level.

Module: starterSelect
Input: [1:0] starterSelect, [4:0] prev_ID, enable
Output: [4:0] new_ID
Description: Selects the starter pokemon the user has chosen.

Purpose: This unique ID is used throughout the game and is stored in the Pokemon register.

Module: rand_counter
Input: frameClk, Reset, rand5_start, [7:0]keycode
Output: fight_on, [1:0]gg
Description: This module has a 9 bit counter that increments between 0 and 509 at every clock cycle.
Purpose: We use this module in order to keep the control bit fight_on for about 20% of the time when the trainer is moving in the grass.

Module: HexDriver
Input: In0[3:0]
Output: Out0[6:0]
Description: The module helps display 4-bit binary number as a 1-bit hex number on the DE2 board. The module takes in the 4 bit binary number and converts it into a single bit hexadecimal.
Purpose: This module is needed in order to display a hex values of the keycodes on the FPGA

Module: wildpoke_select
Input: frameClk, Reset, [1:0] curr_map
Output: [4:0] wild_ID
Description: This module has a 5 bit counter that increments all the time between 0 and 31.
Purpose: Depending on the current map and the counter's value, the wild pokemon's ID is set.

Module: ID_MUX
Input: [4:0] poke2_wild_id, wild_ID, battle_start, wild_ID_final

Output: wild_ID_final
Description: This module is a 2:1 MUX.
Purpose: This MUX is used in order to hold the Pokemon ID while battle_start is high.

Module: poke_battle
Input: Reset, heal_on, hp1_mux, hp2_mux,[4:0] poke1_id,[4:0] poke2_id, battle_done, hp_out1, hp_out2, init_reg, Clk, frameClk, curr_map, curr_level, [3:0] userPkmnlevel, [3:0] wildPkmnlevel, prev_poke_ID
Output: poke2_idOut, wild_poke_reg
Description: This is the comprehensive module which contains all other battle related modules.
Purpose: This provides underlying hardware logic into which control signals can be given to control the FSM.

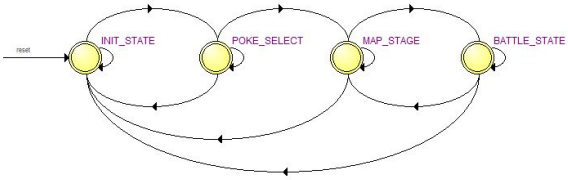
Module: ball.sv
Inputs: frame_clk, Reset, keycode[7:0]
Output: BallX[9:0], BallS[9:0], BallY[9:0], leds[3:0]
Description: This sv module helps us control the movement of the ball. The ball is initially at the center if the screen, and moves up, down, left or right depending on our keyboard input. It also dictates the size of the ball, and what should happen to the ball when it hits the end of the frame/wall.
Purpose: This module is needed in order to control the movement of the trainer, and to decide what must happen to the trainer when he hits any obstacles.

Module: color_mapper.sv
Inputs: BallX[9:0], Ball_size[9:0], BallY[9:0], DrawX[9:0], DrawY[9:0]
Output: Red[7:0], Blue[7:0], Green[7:0]
Description: This decides the shape of the ball, the colour of the ball and background by sending the RGB signals to the screen. The module gets BallX and BallY, which show the center of the ball. If the pixel is within the

radius of the ball, it colours it as the specified colour of the ball, otherwise it gives the pixel the specified background colour.
Purpose: We decide the sprites to be printed onto the the screen depending on the constraints that we set.

Written Description of the State Diagrams

There are essentially two Finite State Machines being employed in our design. On a higher level, there is an FSM with four states, which include: INIT_STAGE, POKE_SELECT, MAP_STAGE, BATTLE_STATE.



	Source State	Destination State	
1	BATTLE_STATE	MAP_STAGE	(!battle:b0).(!keycode[0]).(!keycode[1]).(!keycode[2]).(keycode[3]).(!keycode[4]).(keycode[5]).(!keycode[6]).(!keycode[7]).(!Reset) + (battle:b0).(!Reset)
2	BATTLE_STATE	INIT_STATE	(Reset)
3	BATTLE_STATE	BATTLE_STATE	(!battle:b0).(!keycode[0]).(!keycode[1]).(!keycode[2]).(!keycode[3]).(!Reset) + (!battle:b0).(!keycode[0]).(!keycode[1]).(!keycode[2]).(keycode[3]).(!keycode[4]).(!keycode[5]).(!Reset) + (!battle:b0).(!keyco
4	INIT_STATE	POKE_SELECT	(!keycode[0]).(!keycode[1]).(!keycode[2]).(keycode[3]).(!keycode[4]).(keycode[5]).(!keycode[6]).(!keycode[7]).(!Reset)
5	INIT_STATE	INIT_STATE	(!keycode[0]).(!keycode[1]).(!keycode[2]).(!keycode[3]) + (!keycode[0]).(!keycode[1]).(!keycode[2]).(keycode[3]).(!keycode[4]).(!keycode[5]) + (!keycode[0]).(!keycode[1]).(!keycode[2]).(keycode[3]).(!keyc
6	MAP_STAGE	MAP_STAGE	(!keycode[0]).(!battle_start).(!Reset) + (keycode[0]).(!keycode[1]).(!keycode[2]).(!battle_start).(!Reset) + (keycode[0]).(!keycode[1]).(keycode[2]).(!keycode[3]).(!keycode[4]).(keycode[5]).(!keycode[6]).(k
7	MAP_STAGE	INIT_STATE	(Reset)

MAIN FSM

MAIN FSM:

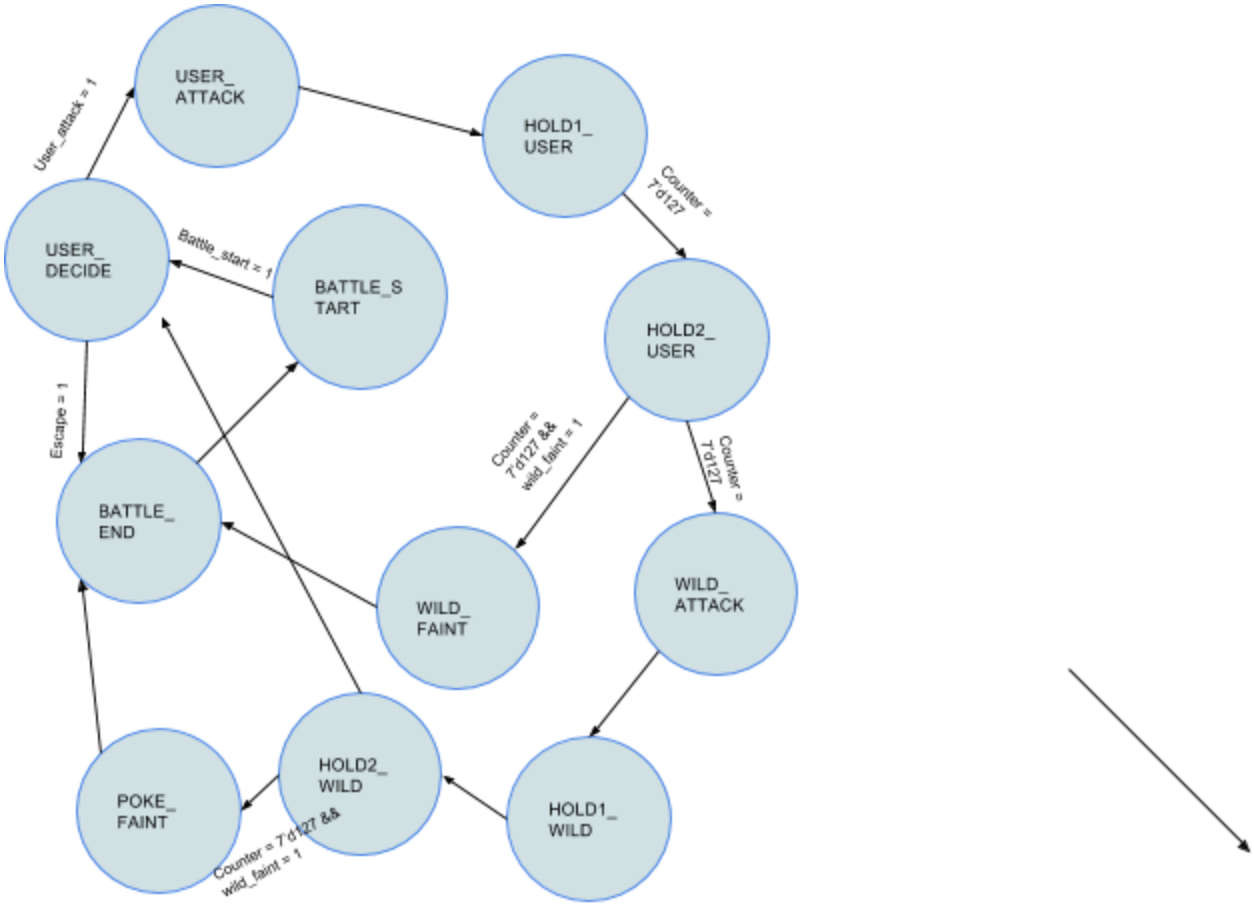
INIT_STAGE : Initializes the registers and the hardware required and sets the control signals required. This loads the start screen and required the User to press “Enter” on the Keyboard to go on to select the Pokemon.

POKE_SELECT : Initializes the User Pokemon register with the values and characteristics of the Chosen Pokemon. This includes, but is not limited to loading

the name of the Pokemon, the Sprite and its attack and defense stats according to its initial level(Level 5). This is the Pokemon controlled by the user throughout the game and can evolve upon reaching a higher level or selectively evolved through the switches.

MAP_STAGE : This is the stage where the user can freely roam on the map and enter wild grass in order to battle with wild pokemon to train his/her own Pokemon. This is necessary to advance through the game. The user can also set the level of difficulty in this stage by setting switches and can battle either mild, medium or legendary Pokemon.

BATTLE_STAGE : This is the stage in which the user can command his Pokemon to attack the wild pokemon or escape. Escape can be used when the player knows that his Pokemon's health is low or if his pokemon is of a disadvantaged type. If the user fights and wins, this leads to an increase in the XP and hence, the level of the Pokemon.



BATTLE FSM

BATTLE FSM:

The Battle FSM consists of:

BATTLE_START : Next state is USER_DECIDE. At the beginning of the battle, the FSM is in this stage.

USER_DECIDE: The user can make an informed decision in this stage whether to fight the Pokemon and gain experience points or whether he wants to escape and heal his Pokemon before advancing further.

USER_ATTACK: In this stage, the hpmux of the wild pokemon is set to 1 so that it can load the calculated value from the Calc_HP module.

HOLD1_USER and **HOLD2_USER**: These states are present to count up from 1 to 127 or for 2 seconds each so that the animations of the battle can be clearly seen. Their next state depends on the HP of the Wild Pokemon.

WILD_ATTACK : If the wild pokemon has still not fainted, it will attack the user's pokemon. The user's Pokemon's hpmux is set to 1 so that it can load the calculated HP from the Calc_HP module.

HOLD1_WILD and **HOLD2_WILD**: These states provide a two second delay and even transition to the next state based on the current HP of the User Pokemon.

WILD_FAINT : Game enters this state if the wild pokemon faints.

POKE_FAINT : Game enters this state if the user pokemon faints.

BATTLE_END : This is the final state which sets the battle_end bit to 1 which makes the MAIN FSM go back to the MAP_STAGE. It's arbitrary next state is BATTLE_START.

Debugging and Verification

Since a testbench would have involved setting many different control signals and dealing with a huge amount of test_memory, we decided that it would be in our best interest to debug and verify different modules by using the on-board switches and LEDs. This proved to be very useful and efficient as it quickly helped us deal with issues rather than trying to decipher the problem by looking at simulated waveforms.

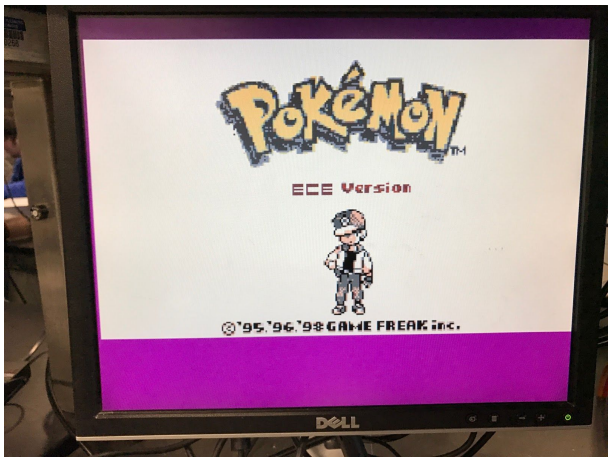
Features and Characteristics

For user interaction with the game, we have used a USB keyboard controlled by the Cypress EZ-OTG Host Controller. We have mostly reused the code from lab8 for the movement of the Sprite and included multiple sprites in order to make the sprite seem animated.

Random Pokemon Generator and Poke Registers:

In order to generate a random pokemon when we enter the grass, we made use of multiple counters. In the rand5_counter module, a 5 bit counter increments while the trainer is moving within the grass. We made sure that the control signal that is sent from this module is high only for twenty percent of the time. We then sent this control bit into the wild_poke_select module, where a 9 bit counter would be running between the values 0 and 511 at all times. As soon as the control signal from the rand5_counter is set to high, the wild wild pokemon is chosen depending on the value of the 9 bit counter in the wild_poke_select module at that particular clock cycle, and the current map. We have made sure that there are four wild pokemon available for each of the three maps, and the level of the wild pokemon keeps increasing as we progress from one map to another.

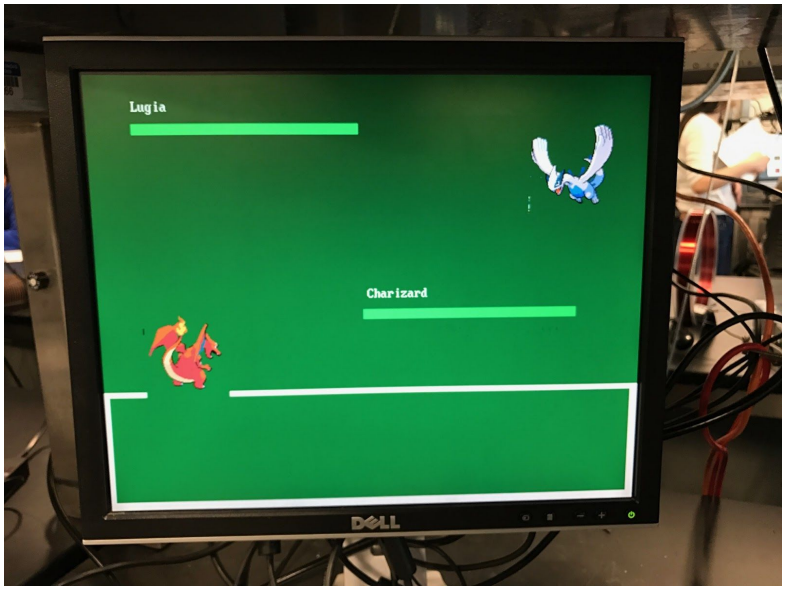
Pictures of the demo:



Start screen

Map1

Map2



Battle state

LUT	8249
DSP	-
Memory(BRAM)	248,832
Flip-Flop	2,317
Frequency	162.84 MHz
Static Power	102.43 mW
Dynamic Power	0.84 mW
Total Power	193.51 mW

Conclusion

In a nutshell, this final project provided us a great insight into hardware design since we were expected to design, document and implement a unique design for a game or a device of our own choice. This gave us a chance to implement various modules, gain experience by writing, testing and debugging SystemVerilog code.

Our project worked impeccably for the most part, only failing to provide constant outputs at times for Pokemon battles. One way to fix this issue would have been by using PIO blocks and defining this hardware in Qsys to implement the arithmetic there like we did for Lab7, however, we were not able to solve this minor glitch in time due to our lengthy compilation time. We were able to demonstrate other advanced features of our design such as random number generation, changing states and advanced graphics by controlling them with the on-board switches and keystrokes through the keyboard.