POKEMON PROJECT

Yuran Wu
Jun Hao
Yi-hsin (Chris) Jong

MARCH 11, 2016

Department of Electrical Engineering, University of Washington, Seattle, 98105

Table of Contents

Abstract

In this project, a game is designed mimicking the game Pokemon. It aims to have the same turn based battle system between two players with an FPGA and microprocessor handling everything from inter-board communication to graphical display.

Introduction

In Pokemon, there are battles where two opponents will face off taking turns attacking. Our system mimics this by using two FPGA boards which communicate with one another. A database is necessary to store status information about the pokemon and so an SRAM module is designed for that purpose. Inputs are handled by buttons and outputs come in the form of a VGA monitor where drawn sprites visually represent the information stored in the boards. This report will introduce the design in detail, and also the test procedure.

Discussion of the Design

Design Specification

In this design, the objective is to develop an interactive battle game based off of Nintendo's "Pokemon". The whole system of the game including front end and back end will be developed on the FPGA. The user interface of the game will be displayed on a computer monitor driven by the system. The console control will be the keys and switches on the FPGA board. The high level introduction of the system modules will be discussed below separately from the front end to the back end.

- SRAM

  An *8×2048* SRAM was developed in order to store all of the data such as the pokemons' skill set damage, health points, and the mana points. The SRAM allows users to write 8-bit data to the SRAM at specific addresses, and store the data at that address unless the user overwrites it. And also, the user is able to read the data at a specific address from the SRAM. The SRAM plays a crucial role in this system since the whole system database is stored there.

- Communication module

  Furthermore, since this game is a battle between two players who use two different consoles, a communication module was developed to pass the data between two systems. The communication module takes 8-bit data through an 8-bit wide bus and passes it out serially. Also, it can receive serial data and processes it as 8-bit wide data in parallel.

- Display driver module

  Moreover, the game is displayed on screen via VGA port, thus a display driver is developed to graphically represent all of the pokemon, pokemons' healing points, pokemons' mana points and all of the moves the pokemon has access to. Depending on the user input (via keys and switches), the data is passed into the module and the display will alter simultaneously.

- Game module

  As mentioned before, this game is similar to a well known game "Pokemon" which is a fighting game between two "Pokemon". It is a round-based battle where each pokemon takes turn attacking the other. In the attack round, the player can choose to either attack the enemy, or use items to heal his or her "Pokemon". If the player intends to attack the enemy, four skills are available which have different damages with different mana costs. In general, the higher the damage is, higher the cost will be. On the other hand, if the player opts to use items, two kinds of items can be used:"Health Potion" and "Mana Potion". The "Health Potion" will heal the pokemon up to 50 healing points or until the Pokemon is fully healed. The "Mana Potion" will recover the pokemon's mana points in similar fashion. Each player has three pokemon in total, if one of them has fainted, the system will automatically switch to the next one pokemon. All of the skills will then be updated to those specific to that next pokemon. However, the inventory of the "Health Potion" and the "Mana Potion" will stay the same. If the player loses all of his or her pokemon, he or she then loses the game.

Design Procedure

This system was developed module by module in sequence from the back end to the front end. The design started development with the SRAM which provides the back-up for database of the system, then the communication module which ensures that the data transfer between two systems, the graphic driver module which provides the user interface of the system, to the game module that controls all of the other modules and manipulates the data flow throughout. In this section, the design procedure of each module will be introduced following the same sequence as the previous section:

- SRAM

  The system requires a memory module (SRAM) to maintain all of the data used in the system. The SRAM is an 8-bit wide read and write memory with a size of 2K bytes. The verilog below shows the main function of the SRAM:

```
reg [7:0] m[2047:0];

always@(posedge clk)
if(enable) begin
```

```
if(wr)
m[addr] <= data_I; // write
else
data_O <= m[addr]; // read
end
```

In order to store the data on the hardware, the subsystem requires sequential logic, in other words, a D-flipflop. Since the SRAM is an 8-bit wide memory, 8 registers are needed for 8 D-flipflops. To write the data into the SRAM, the user needs to set the address line which is an 8-bit bus first. The read_write line is then required to be pulled high in order to allow the register to load the input 8-bit data. The data line can then be set to the desired value. After the data is set, the read_write line should then be pulled low to set the SRAM into read mode which is the default mode. If the read_write line is set to low which is under the read mode, the output bus will be assigned to the data at the location which is specified by the address line.

- Communication module

The communication module allows the system to transfer an 8-bit data to another system which can resolve the data properly. The communication module has two parts: the transmitter and receiver. The transmitting part will take an 8-bit data from the microprocessor and store it in the shift register in parallel. The shift register will then pass the data out in serial bit by bit. The receiver will receive 10 data in serial if the starting bit is detected. After the data is fully received, the correct 8-bit data is output to the microprocessor. The communication module has few sub-modules in it. All of the sub-modules will be introduced as follow.

"P2S_SR" module:

"P2S_SR" module is the parallel to serial shift register. This shift register will take two clocks lines as inputs, one reset line, one load line, one end_pass line and an 8-bit data line. The output will be a 1-bit data line. The verilog code below shows the main functions of the "P2S_SR" module:

```
initial S_data_out = 1'b1;
initial buffer = 10'b1111111111;
reg clk, sel;
reg [9:0] buffer;
always@(*)
if(1'b1 == load & 1'b0 == end_pass) begin
        clk = CLOCK_50;
end
else if(1'b0 == load & 1'b0 == end_pass) begin
        clk = ic_clk_ctrl;
end
else begin
        clk = ic_clk_ctrl;
end

always@(negedge clk)
```

```
if(reset) begin
        if(1'b0 == load & 1'b0 == end_pass) begin

        buffer <= {buffer[8:0], 1'b1};
        S_data_out <= buffer[9];
        end
        else if(1'b1 == load & 1'b0 == end_pass) begin
        buffer[0] <= 1'b1;
        buffer[8:1] <= P_data_in;
        buffer[9] <= 1'b0;
        S_data_out <= 1'b1;
        end
        else begin
        buffer <= buffer;
        S_data_out <= 1'b1;
        end
end
else begin
        buffer <= 10'b1111111111;
        S_data_out <= 1'b1;
end
```

The module has two clock lines. The “ic_clk_ctrl” is the inverse of the fourth bit of the output of the “bit_counter” module which will be discussed in the following sub-sections. The functionality of “ic_clk_ctrl” is to send out the data in serial. As the “load” line is pulled low which means the data is loaded successfully, and the “end_pass” line is low as well, which means the data is not fully passed. The “ic_clk_ctrl” clock will toggle for 10 clock cycle at a baud of 9600 bit/sec. If the “load” line is high, the “ic_clk_ctrl” clock will not work which is determined by the “bit_counter” module. In order to load the data into shift register, another clock line must be introduced to the shift register which is the “CLOCK_50”. When “load” line is high, and the “end_pass” line is low so that the data is not fully passed (in this case, it is not passed at all). Under the clock of “CLOCK_50”, the 8-bit data will be stored to the shift register with a starting bit (1’b0) at the last bit of the shift register, and an end bit (1’b1) at the first bit of the shift register. After the data is loaded successfully, the “ic_clk_ctrl” starts working, and the last bit of the shift register will be assigned to the output, and all of the data will be shifted towards to the end bit of the shift register. A 1’b1 will be assigned to the first bit of the shift register. Since the output data line should be high by default. On reset, the shift register should be set to all 1’b1s in case that the “ic_clk_ctrl” is occasionally enabled which passes a “0” to the output.

“S2P_SR” module:

Similar to “P2S_SR”, “S2P_SR” module is another shift register but in an reversed direction. The “S2P_SR” will take a 1-bit input “S_data_in”, one clock line “ic_clk_ctrl”, one “read” line, and one “receive_enable” line. It has an 8-bit output data line “P_data_out”. The verilog code below shows the main functions of this module:

```
always@(*)
if(receive_enable & !read) begin
```

```
P_data_out <= P_data_out;
end
else if(receive_enable & read) begin
P_data_out <= buffer[8:1];
end
else begin
P_data_out <= 8'b11111111;
end

always@(negedge ic_clk_ctrl)
if(reset) begin
        if(receive_enable) begin
        buffer = buffer << 1;
        buffer[0] = S_data_in;
        end
        else begin
        buffer <= buffer;
        end
end
else
buffer <= 9'b111111111;
```

The starting bit is 0. If the "sb_detect" module, which is the starting bit detector, detects a 0 passed in, it will pull the "receive_enable" line of the "S2P_SR" module to high. As the "receive_enable" is pulled high, and the "read" line stays low, the shift register will start to receive the data from the "S_data_in", and shift the data from the 0th bit of the shift register to the 8th bit of the shift register. At the last clock cycle of the shifting phase, the read will be pulled high by the "bit_counter" module. The data in the shift register will be assigned to the output "P_data_out". By default, the output will be assigned to all 1s.

"bit_counter" module:

The "bit_counter" module is an 8-bit counter that will take four inputs and has two outputs. Inputs include a reset "reset" line, a clock "clk" line, a enable "transmit_enable" line, and a "load" line. All of the input lines are 1-bit lines. The outputs are a 1-bit "char_sent" line, and an 8-bit "bit_count" line. The following Verilog code shows the main function of the "bit_counter" module.

```
 reg [7:0] ps, ns;

initial ps = 8'b00000000;
initial ns = 8'b00000000;

always@(*)
if(transmit_enable & !load & ps[7:4] != 4'b1010) ns <= ps + 8'b00000001;
else if(transmit_enable & !load & ps[7:4] == 4'b1010) ns <= 8'b00000000;
else ns <= 8'b00000000;

always@(negedge clk)
```

```
if(reset)
ps <= ns;
else
ps <= 8'b00000000;

assign bit_count = ps;
assign char_sent = (4'b1010 == ps[7:4]);
```

The counter will start with 0. The “load” line in this module is the same “load” line in the “P2S_SR” module. If the “load” is high, the counter should not be enabled because the shift register is not ready for transmitting the data yet. As the “load” is pulled low which means the data is ready to be sent, the “transmit_enable” line will enable the counter to count up to 8’b10100000, and assign the current value to the output. Both of the parallel to serial shift register and the serial to parallel shift register are using the inverse of the fourth bit of the output as the clock. The reason why the shift registers use the inverse of the fourth bit of the output as the clock is that the last four bit of the counter will count up from 4’b0000 (0) to 4’b1010 (10) which represents that ten bits of data are transferred to the output. Every time the first bit of the last four bits of the output count up 1, the clock should have been through 1 whole clock cycle. The fourth bit of the counter will exactly do this job as the last four bit counts up. As the counter counts to 8’b10100000, the data is then fully sent, so that the “char_sent” should be pulled high to notify the other modules the data is fully transferred.

“sampling_mod” module:

The “sampling_mod” module will sample the data while the data is transferred out of the parallel to serial shift register and the data that is passed into the serial to parallel shift register. The “sampling_mod” will take the middle bit of the input data as the output data. The module has four inputs including a “reset” line, a clock “sc_clk_ctrl” line, a input data “data” line, and a “sample_phase” line. The following verilog code shows the main function of the “sampling_mod”.

```
reg [2:0] n_count, p_count;

always@(*)
if(3'b111 != p_count & 1'b1 == sample_phase) begin
n_count = p_count + 3'b001;
S_data_out = S_data_out;
end
else begin
n_count = 3'b000;
S_data_out = data;
end

always@(negedge sc_clk_ctrl)
if(reset)
p_count <= n_count;
else
p_count <= 3'b000;
```

The “sample_phase” line takes the fourth bit of the output of the “bit_counter”. As the “sample_phase” is high, the internal 3-bit counter should count up by 1. As the counter counts up to 3’b111, the current data sampled should be assigned to the output, and the counter will be reset to 0. The clock of this module will use the last bit of the counter, so that it can retrieve the data at the center of each clock cycle of the clock of the shift register.

“sb_detect” module:

The “sb_detect” module is the starting bit detector of the receiving part. If a starting bit is detected, the module will enable the bit counter of the receiving part to initialize the receiving procedure of the shift register. This module will take four inputs including a clock “clk” line, a reset “reset” line, an input data “data” line, and a “char_received” line. The module will produce an output of “receive_enable”. The following Verilog code shows the main function of the “sb_detect” module.

```
initial receive_enable = 1'b0;
always@(negedge clk)
if(reset) begin
case(char_received)
        1'b0: if(!data_in & 1'b0 == receive_enable) receive_enable = 1'b1;
                        else if(1'b1 == receive_enable) receive_enable = 1'b1;
                        else receive_enable = 1'b0;
        1'b1: receive_enable = 1'b0;
        default receive_enable = 1'b0;
endcase
end
else
receive_enable <= 1'b0;
```

The “char_received” represents if the data is fully received by the shift register. If it is, the shift register shall be disabled so that the output will be assigned to 0. If the “char_receive” is low, the data is then not fully received or not received at all. So that if the input data is 0 while the “receive_enable” is low, this input data must be the starting bit of the data, so that the shift register should be enabled. If the “receive_enable” is high which means the shift register is enabled, the shift register should be kept enabled until the data is fully passed to the shift register (char_received == 1’b1). By default, the output should stay at 0 so that it will not necessarily enable the shift register.

- Graphical driver module

To utilize the FPGA’s VGA port and display graphics to the screen, the board needed a driver to translate digital signals into the analog graphic signal output. The optimal resolution 640 pixels by 480 pixels. The driver draws the image by selecting an x-y coordinate giving it a color. The colors are represented by three 8 bit unsigned integers which represent the intensity of each of the color channels (red, green, blue). The refresh rate of the graphics was designed at 60hz for a stable viewing experience. The driver

concept is shown below.

```
always @(posedge CLOCK_50)
CLOCK_25 <= ~CLOCK_25;
always @(posedge CLOCK_25) begin
if(reset) begin
xt <= 0;
yt <= 0;
xd <= 0;
yd <= 0;
x <= 0;
y <= 0;
end else begin
read_enable_last <= read_enable;
if(read_enable) begin
xt <= xt + 1'b1;
if(xt >= X_START && xt < X_STOP) begin
if(xd == BLOCK_STOP) begin
xd <= 10'b0;
x <= x + 1'b1;
end else begin
xd <= xd + 1'b1;
end
end else begin
xd <= 10'b0;
x <= 10'b0;
end
end else begin
xt <= 10'b0;
xd <= 10'b0;
x <= 10'b0;
end
if(end_of_active_frame) begin
yt <= 9'b111111111;
yd <= 9'b0;
y <= 9'b0;
end else begin
if(read_enable_last & ~read_enable) begin
yt <= yt + 1'b1;
if(yt >= Y_START && yt < Y_STOP)
          begin
if(yd == BLOCK_STOP) begin
yd <= 9'b0;
y <= y + 1'b1;
end else begin
yd <= yd + 1'b1;
end
end else begin
yd <= 9'b0;
y <= 9'b0;
end
```

```
end
end
end
end
assign VGA_BLANK_N = vga_blank;
assign VGA_CLK = CLOCK_25;
assign VGA_HS = vga_h_sync;
assign VGA_SYNC_N = 1'b0;
assign VGA_VS = vga_v_sync;
reg [7:0] rout, gout, bout;
always @(posedge CLOCK_25) begin
if(xt >= X_START && xt < X_STOP && yt >= Y_START && yt
          < Y_STOP) begin
rout <= r;
gout <= g;
bout <= b;
end else begin
rout <= 8'b0;
gout <= 8'b0;
bout <= 8'b0;
end
end
```

The Altera VGA module which was built into the board was used to translate the digital information into the output VGA signal. Code is shown below.

```
altera_up_avalon_video_vga_timing video (
// inputs
.clk(CLOCK_25),
.reset,
.red_to_vga_display({rout,2'b00}),
.green_to_vga_display({gout,2'b00}),
.blue_to_vga_display({bout,2'b00}),
.color_select(4'b1111),
// outputs
.read_enable,
.end_of_active_frame,
.end_of_frame,
// dac pins
.vga_blank, // VGA BLANK
.vga_c_sync, // VGA COMPOSITE SYNC
.vga_h_sync, // VGA H_SYNC
.vga_v_sync, // VGA V_SYNC
.vga_data_enable, // VGA DEN
.vga_red(VGA_R), // VGA Red[9:0]
.vga_green(VGA_G), // VGA Green[9:0]
.vga_blue(VGA_B),   // VGA Blue[9:0]
.vga_color_data()   // VGA Color[9:0] for TRDB_LCM
);
```

Each frame contains 2 Pokémon, one for each player, as well as various bars to provide information about the status of your Pokémon, etc. Each sprite is drawn pixel by pixel in Verilog. Using a scaling factor, it is also possible to display the graphics in different sizes and even move them around on the screen. The health and mana bars (red and blue) reflect the health and mana status of the Pokémon as stored within the SRAM. The 3 pokeballs represent the number of Pokémon the player has left, removing one each time a Pokémon faints. The 6 squares found on screen represent the skills available as well as the health and mana potion options. Whenever, a pokemon attacks, its sprite will move up and down. Upon taking damage, the receiver's half of the screen will flash red. When your pokemon has defeated all of the enemies, it will move to the center of the screen in victory.

With this system, the users will be able to play the game relying only on the information displayed on screen.

- Game module

The game module was built based on the NIOS II microprocessor. The game module controls and manipulates the SRAM, the communication module, and the graphical driver module to process the whole pokemon battle game through the NIOS II microprocessor. In this system, all of the hardware module was implemented with the "Qsys" tool to wire up each component to the microprocessor. The connection is shown in Figure 1, 2 below.

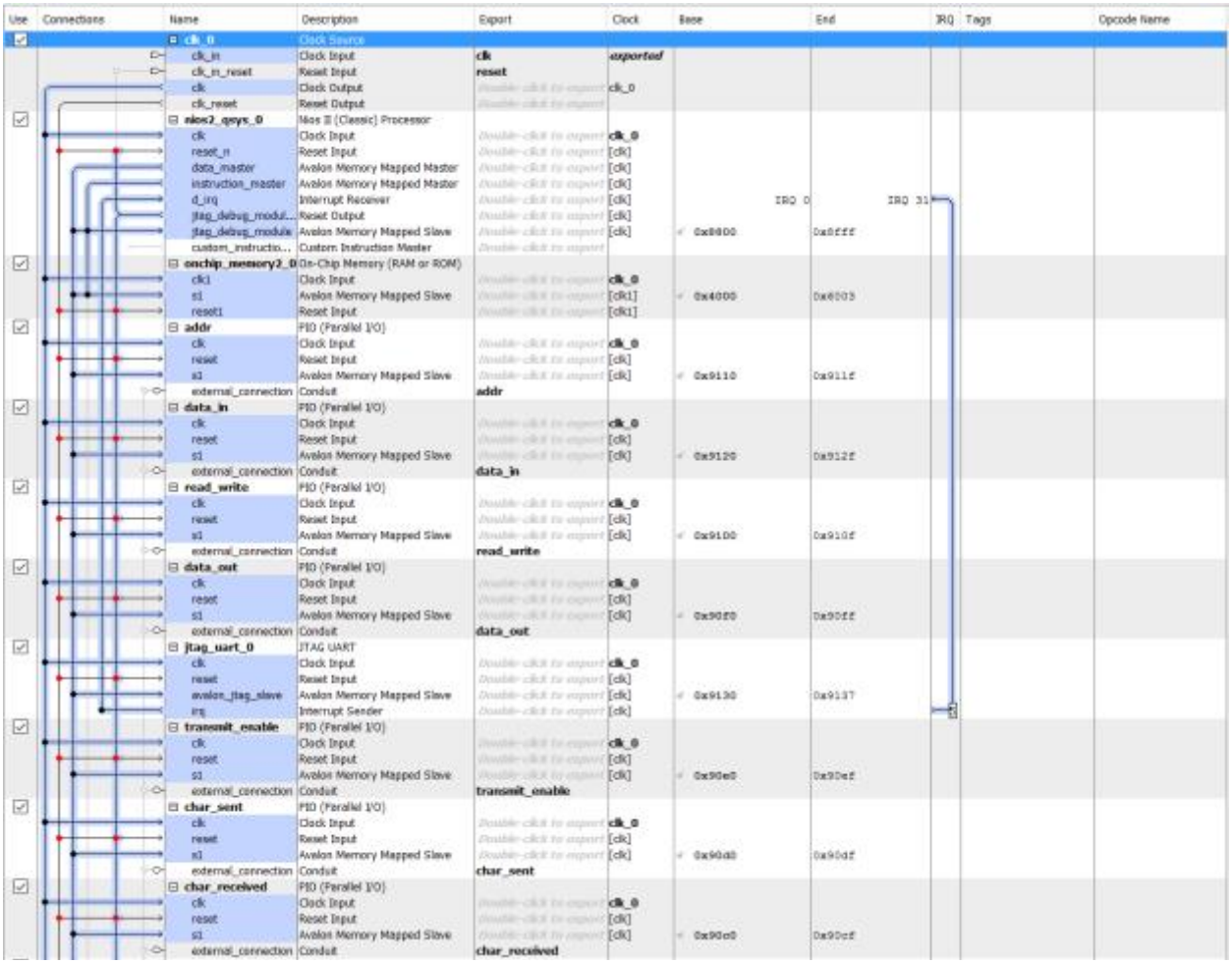

Figure 1-Qsys module connection schematic part 1

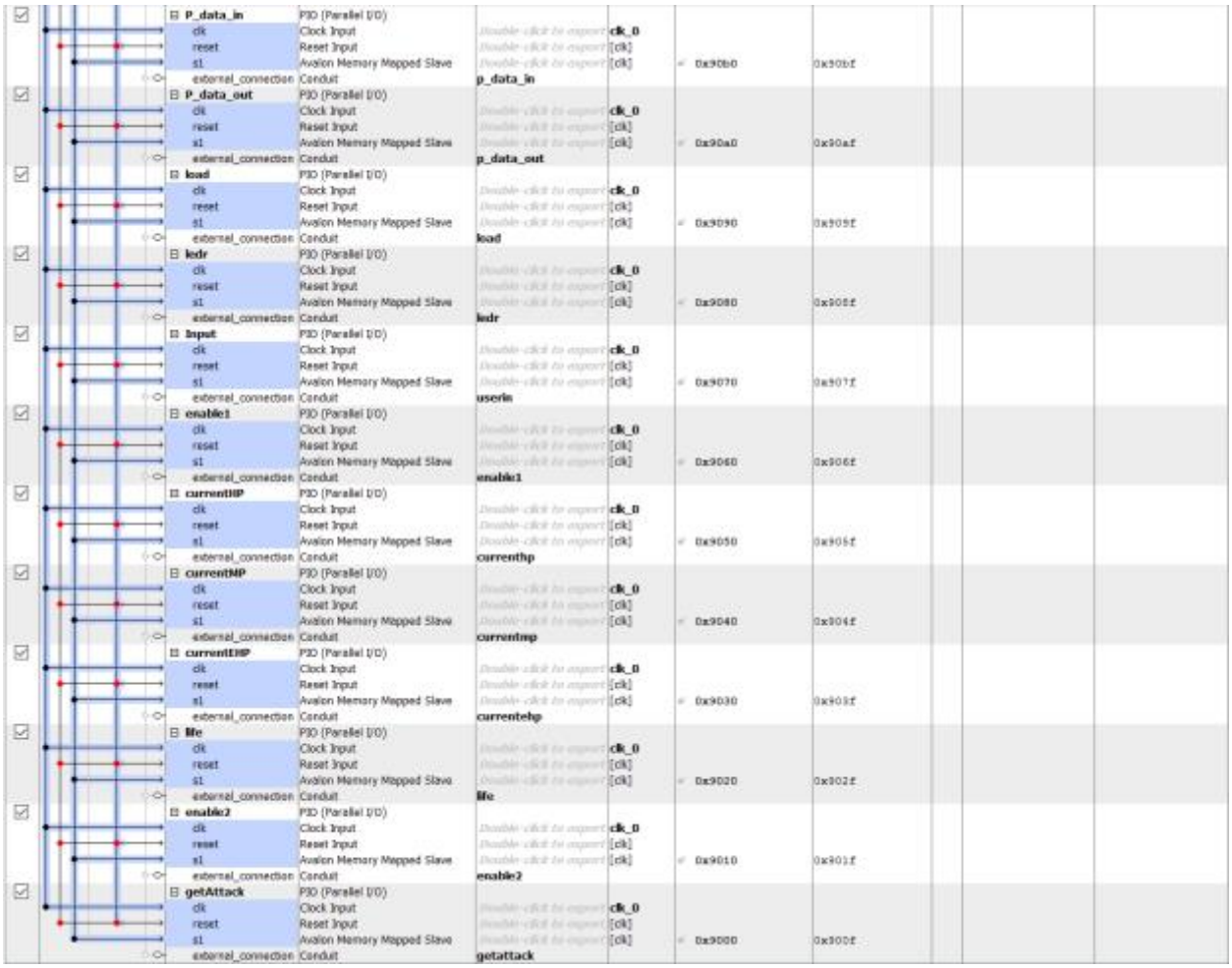

Figure 2-Qsys module connections schematic part 2

The microprocessor module consists of multiple components including a clock module, a NIOS II (classic) microprocessor, an on-chip memory, a JTAG-uart module, and multiple Parallel I/O modules. Each component is assigned a unique base address as shown in Figure 1, 2. All of the component will be discussed below separately:

Clock module:

The clock module uses a system clock with a frequency of 50 MHz in order to synchronize the system on the microprocessor and on the FPGA. The clock of the microprocessor has the same frequency as the FPGA.

NIOS II microprocessor:

For the microprocessor, as shown in Figure 3, a NIOS II/e core is selected. The NIOS II/e core is a 32-bit microprocessor without instruction cache, branch prediction, hardware multiply, and hardware divide. It only contains the basic microprocessor's functionalities.

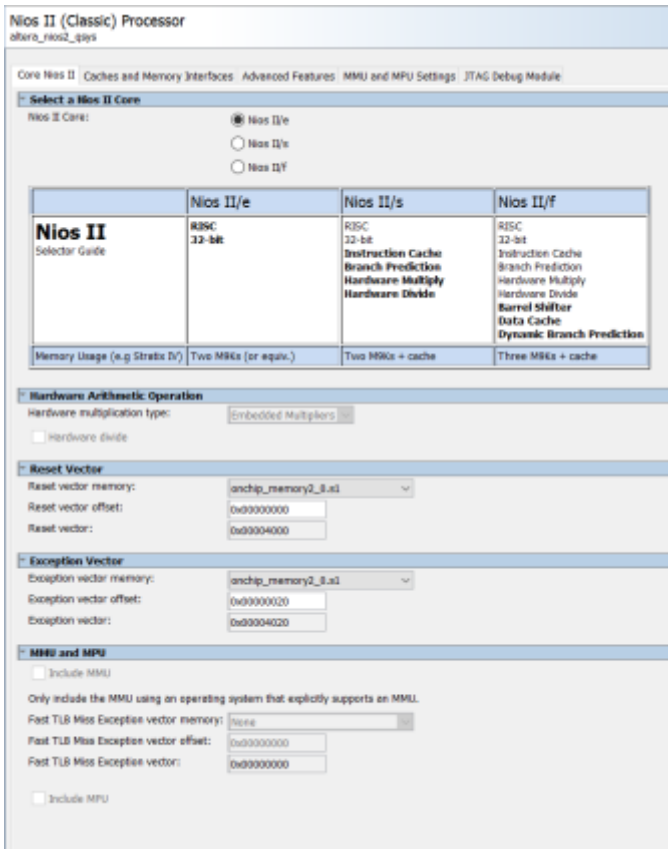

Figure 3-NIOS II/e microprocessor and its characteristics

On-chip memory:

In this design, the system utilizes a RAM (readable and writable memory) as the microprocessor module's memory module. The data width is the same as the data width of the NIOS II microprocessor which is 32-bit. Considering the size of the database and the weight of calculation in the game module, the memory size is chosen to be 8196 bytes which is 8K bytes. The details of the on-chip memory is shown in Figure 4 below.

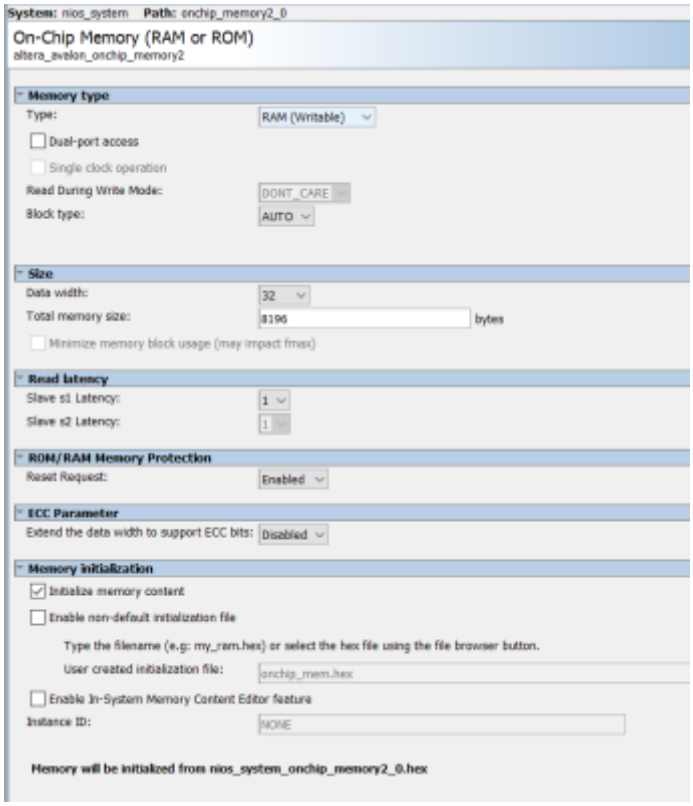

Figure 4-On-chip memory module and its characteristics

Parallel I/O peripherals:

Parallel I/O (PIO) peripherals are the actual connections between FPGA modules and the microprocessor. The output of the microprocessor will be the input of the FPGA modules, and the input of the microprocessor will be the output of the FPGA modules. For this system, 19 PIOs are allocated for different purposes. This sub-section will introduce each of the PIO follow the sequence in Figure 1, 2.

1. “addr”:

   The “addr” PIO is an 11-bit wide output bus. It allows the microprocessor to pass the address of the SRAM to the SRAM module, so that the user can then either read from or write to the correct memory cell.

2. “data_in”:

   The “data_in” PIO is an 8-bit wide output bus. It allows the microprocessor to assigned the data to the SRAM when the user wants to write data into the SRAM to a certain memory cell whose location is specified by “addr” bus.

3. “read_write”:

The “read_write” bus is a 1-bit output bus that is an input of the SRAM module. The “read_write” bus notifies the SRAM module if the user will write the data into the SRAM, or he or she will read the data from the SRAM.

4. “data_out”:

   The “data_out” PIO is an 8-bit wide input bus. It will deliver the data in the SRAM at location which is assigned by “addr” to the microprocessor when the user intends to read the data from the SRAM.

5. “transmit_enable”:

   The “transmit_enable” PIO is an 1-bit output bus. It will enable the “bit_counter” module to start the clock, so that the shift register (either parallel to serial or serial to parallel shift register) will be enabled.

6. “char_sent”:

   The “char_sent” is an 1-bit input bus. If the data is sent to the parallel to serial shift register and the “transmit_enable” is pulled high, the shift register then starts to work. After the data is fully sent, the “char_sent” will be high to notify the microprocessor the data is sent.

7. “char_received”:

   Similar to “char_sent”, the “char_received” is an 1-bit input bus as well. If the starting bit detector detects the starting bit, it enables the shift register to receive the data. If the data is fully received, the “char_received” will be pulled high to notify the microprocessor the data is ready to be read.

8. “P_data_in”:

   The “P_data_in” is an 8-bit output bus. The bus allows the user to input the 8-bit data into the parallel to serial shift register. The “P_data_in”is only functional when the “load” line is high.

9. “P_data_out”:

   The “P_data_out” is an 8-bit input bus. The bus allows the user to output the data from the serial to parallel shift register in parallel. The “P_data_out” can only read the data in the shift register when the serial to parallel shift register has received the data successfully so that the “char_received” line is high. Otherwise, it will return 0b11111111.

10. “load”:

The “load” is a 1-bit output bus. It allows the user to enable the “P_data_in” bus to input the data into the parallel to serial shift register. If the “load” line is high, the data passed into the “P_data_in” bus will be stored into the parallel to serial shift register, and the data will be sent out in serial if the “load” bus is pulled to low afterward.

11. “ledr”:

    The “ledr” bus is an 8-bit output bus. It is a test bus for testing the data in the data bus.

12. “Input”:

    The “Input” bus is a 6-bit input bus. It will input different data to the microprocessor depending on the different user input. Since there are four moves and two items for the game module, there are six different input data points to represnt each move or item.

13. “enable1”:

    The “enable1” bus is a 6-bit output bus. It will enable the current pokemon of the player and display it on the screen.

14. “enable2”:

    The “enable2” bus is also a 6-bit output bus. Similar to the “enable1” bus, it will enable the pokemon and display on the screen. However, the pokemon displayed on the screen is the pokemon of the enemy instead of the player.

15. “currentHP”:

    The “currentHP” bus is an 8-bit output bus. It will pass the data which represents the current healing points of the player’s current pokemon to the display driver module. And display the current healing points of the player’s current pokemon to the screen.

16. “currentMP”:

    The “currentMP” bus is also an 8-bit output bus. It will pass the data which represents the current mana points of the player’s current pokemon to the display driver module. And display the current mana points of the player’s current pokemon to the screen.

17. “currentEHP”:

    The “currentEHP” bus is an 8-bit output bus as well. Similar to the “currentHP” bus that will pass the data of current healing points of the player’s current pokemon. The “currentEHP” will pass the data which represents the enemy’s current pokemon’s health points to the display driver module. And display the current health points of the enemy’s pokemon to the screen.

18. “life”:

    The “life” bus is a 3-bit output bus which will pass the data to the display driver to enable a “pokemon ball” on the screen to represents the number of the pokemon left for the player.

19. “getAttack”:

    The “getAttack” bus is a 1-bit output bus which will enable a animation effect while the player’s current pokemon is being attacked.

After the microprocessor was wired up to the FPGA modules. The software of the game itself was developed. The software manipulates the data flows inside the whole project. Initially, the program requires a data type to describe the player’s pokemon, player’s current pokemon, and player’s inventory to the user. The player is described as a struct in C called “player”. The C code below shows the “player” struct in this project.

```
typedef struct {
        char* name;
        int currentMon;
        int mon[3];
        int tg;
        int mg;
} player;
```

In the system, each player needs to be initialized by assigning values to the data in the struct via an initialization function. In the function, all of the pokemons’ numbers must be given. The current pokemon initially is set to 0. And all of the data that describes the pokemon and the items will be stored to the SRAM. The functions that allocates the SRAM memory and stores the data will be introduced in the following paragraphs. The C code of the initialization function below.

```
player initPlayer(player plr, int mon0, int mon1, int mon2) {
        plr.mon[0] = mon0;
        plr.mon[1] = mon1;
        plr.mon[2] = mon2;
        plr.currentMon = 0;
        memAlloc(plr);
        return plr;
}
```

In order to store the data to the SRAM, a function is created for writing the data to the SRAM. As mentioned before, the data can be written to the SRAM by pulling the “read_write” bus to high, and putting the data into a memory cell whose location is specified by the “addr” bus. The C code below shows the function that write the data to the SRAM.

```
void setValue(char add, char data) {
        *addr = add;
        *read_write = 1;
        *data_in = data;
```

```
        *read_write = 0;

}
```

The system is also able to read the data from the SRAM at specified location. Getting the value from the SRAM is simpler than writing the value to the SRAM since by default, the "read_write" line is set to 0 which is the read mode of the SRAM. The user only needs to pass the address to the SRAM, the SRAM will then return the value at that address to the microprocessor. The C code below shows the function that reads the value from the SRAM.

```
char getValue(char add) {
        usleep(500);
        *addr = add;
   return *data_out;
}
```

After creating a struct to describe the player, the game then needs a database to store all of the data such as damage of the pokemon, the cost of each move, the healing of the items, or the current health points of the pokemon. Instead of creating different structs to describe different objects like pokemon or moves, the system chooses to store all of the data into SRAM because the size of on-chip memory is limited. And it will increase the workload of the microprocessor if the system store all of the data in the on-chip memory.

For the pokemon, each pokemon has four moves. Each move has a damage, a mana cost, and a critical rate (designed for future use). In this system, 6 pokemon are created in total, so that the design uses a $6 \times 4 \times 3$ char array to store all of the data. The C function will return the certain data if the index of the data is properly given as the argument of the function. The C code below shows the function that store those data.

```
char pokemonSkill(int pokemonNum, int skillNum, int type) {
        char skill[6][4][3] = {
                        {{5, 0, 1}, {25, 25, 5}, {15, 10, 3}, {30, 25, 3}}, // pokemon 0
                        {{5, 0, 1}, {40, 50, 4}, {30, 25, 5}, {20, 15, 5}}, // pokemon 1
                        {{5, 0, 1}, {20, 15, 5}, {20, 20, 3}, {20, 25, 4}},  // pokemon 2
                        {{5, 0, 1}, {30, 30, 3}, {25, 25, 3}, {15, 15, 2}}, // pokemon 3
                        {{10, 0, 2}, {35, 35, 3}, {20, 20, 2}, {25, 30, 4}}, // pokemon 4
                        {{10, 0, 2}, {35, 30, 4}, {25, 30, 2}, {30, 35, 3}} // pokemon 5
        };
        return skill[pokemonNum][skillNum][type];
}
```

Pokemon has two properties, health points and mana points (hp and mp). The hp and mp share the same equation that $hp/mp = 60 + factor \times 10$. The system has a function to return the value of hp and mp by inputting the factor value to the function. The C code below shows the function that calculates the hp and mp.

```
int hpmp (int factor) {
        return 60 + factor * 10;
}
```

After the hp and mp is calculated, the hp and mp are stored to a two dimensional array. The function is created to return the value of hp and mp if the index of the array is properly given to the function. The C code below shows the function in the system that will return such value.

```
char pokemonProp (int pokemonNum, int prop) {
	char pokemonProperty[6][2] = {
			{hpmp(4), hpmp(3)},
			{hpmp(3), hpmp(4)},
			{hpmp(6), hpmp(1)},
			{hpmp(5), hpmp(2)},
			{hpmp(2), hpmp(5)},
			{hpmp(2), hpmp(6)},
	};
	return pokemonProperty[pokemonNum][prop];
}
```

With all of the functions that describe the objects in the system, the data can then be stored to the SRAM in a regulated pattern. For each player, 60 memory cells will be pre-allocated to store all of the data. The 60 memory cells will be grouped into four parts. First three parts will have 18 memory cells for each part, and the last one will have 5 cells. The first three groups of memory cells are used to store the data for moves' damage, cost, critical rate, hp, and mp of the pokemon. By calling the function that will return the value required, this function can then set the value to the SRAM in sequence. In each group of 18 memory cells, the properties of moves, hp, and mp value will be put in sequence. The last group of memory cells stores the current pokemon number, the amount of the health potion, the healing effect of the health potion, the amount of the mana potion, and the healing effect of the mana potion. At the end, there is a global variable that increments the busy flag for the memory allocation function that will prevent the next memory allocation function overwrites the previous data. The C code below shows the main function of the memory allocation function in the system.

```
void memAlloc(player plr) {
	int i, a, b, c;
	// pokemon attributes memory allocation
	for(i = 0; i < 3; i++) {
		// skill memory allocation
		for(a = 0; a < 4; a++) {
			setValue(busyAddr + i * 18 + a * 4, 0); // pokemon No.
			setValue(busyAddr + i * 18 + a * 4 + 1, pokemonSkill(plr.mon[i], a, 0)); //
damage
			setValue(busyAddr + i * 18 + a * 4 + 2, pokemonSkill(plr.mon[i], a, 1)); //
cost
			setValue(busyAddr + i * 18 + a * 4 + 3, pokemonSkill(plr.mon[i], a, 2)); //
critical
		}
		// tango memory allocation
		setValue(busyAddr + i * 18 + 16, pokemonProp(plr.mon[i], 0)); // hp
		setValue(busyAddr + i * 18 + 17, pokemonProp(plr.mon[i], 1)); // mp
	}
```

```
        setValue(busyAddr + 54, plr.currentMon); // current pokemon memory

        // tango
        setValue(busyAddr + 55, items(0, 0)); // amount
        setValue(busyAddr + 56, items(0, 1)); // heal

        // mango
        setValue(busyAddr + 57, items(1, 0)); // amount
        setValue(busyAddr + 58, items(1, 1)); // heal
        busyAddr = busyAddr + 60;
}
```

Since all of the data are sent to the SRAM, the user need to get the value by passing an address to the the "getValue" function to retrieve the data from the SRAM. The address of each data can be calculated mathematically. That data is put in the first three groups of memory cells are calculated by a function that $player\ number \times 60 + current\ pokemon\ number \times 18 + if\ using\ move \times move\ number \times 4 + offset$. For "if using move", if the user wants to retrieve the data such as damage or cost, the "if using move" variable should be 1, otherwise be 0. The offset varies depending on the type of data the user wishes to retrieve. The data that is put into the last group of memory cells is stored in relatively static position. The address can be returned without complicated calculation. The function is $player\ number \times 60 + offset$. The C code below shows the function that will return the address of the data.

```
int getAddress(int currentMon, int skill, char type, int playerNo) {
        int offset = 0;
        int isSkill= 0;
        skill = skill - 1;
        if('d' == type) { // damage
                offset = 1;
                isSkill = 1;
        } else if('c' == type) { // cost
                offset = 2;
                isSkill = 1;
        } else if('a' == type) { // critical
                offset = 3;
                isSkill = 1;
        } else if('h' == type) { // hp
                offset = 16;
                isSkill = 0;
        } else if('m' == type) { //mp
                offset = 17;
                isSkill = 0;
        } else if('t' == type) { // tango's heal
                offset = playerNo * 60 + 56;
                isSkill = 0;
                return offset;
        } else if('l' == type) { // tango's amount
                offset = playerNo * 60 + 55;
                isSkill = 0;
                return offset;
```

```
	} else if('g' == type) { // mango's heal
		offset = playerNo * 60 + 58;
		isSkill = 0;
		return offset;
	} else if('k' == type) { // mango's amount
		offset = playerNo * 60 + 57;
		isSkill = 0;
		return offset;
	}

	return playerNo * 60 + currentMon * 18 + isSkill * skill * 4 + offset;
}
```

For the communication module, the read and write function are provided for the user to send the data to the other system, and read the data back from the other system. For the read function, as the “char_received” bus is pulled high, the shift register then successfully receives the data from the other system. The data in the “p_data_out” bus should then be the data that is passed into the system. The C code below shows the read function of the system.

```
char read(void) {
	while (1) {
	  if(1 == *char_received){
		return *p_data_out;
	  }
	}
	return 0;
}
```

Other than the read function, the system requires a write function to pass the data to the other system through the communication module. As introduced before, the write operation should first pull the “load” bus to high, and set the data into the “p_data_in” bus. The “load” line should then be set low and enable the “transmit_enable” bus to initialize the shift register to shift the data out in serial. If the communication return a “char_sent” signal. The data is then successfully sent. The “transmit_enable” shall then be pulled low. The C code of the write function is shown below.

```
void write(char userIn) {
	*load = 1;

	 *p_data_in = userIn;

	 if (1 == *load) {
		*load = 0;
	 }
	 *transmit_enable = 1;
	 do {
		if(1 == *char_sent){
			usleep(100);
			alt_printf("Data is sent successfully.\n");
		}
```

```
        } while (0 == *char_sent);

        *transmit_enable = 0;
}
```

In order to display the current pokemon of the player and the current pokemon of the enemy, the system needs to send two 6-bit data groups to the graphic driver to enable current pokemon's display. A function is created to decode the data in the player struct that represents player's current pokemon to the data that can enable the actual display on the screen. The C code below shows the function that will do this job.

```
char singlePokeDecoder(int currentMon) {
        char enableBus = 0;
        switch(currentMon) {
                case 0:
                        enableBus |= 0b000001;
                        return enableBus;
                case 1:
                        enableBus |= 0b000010;
                        return enableBus;
                case 2:
                        enableBus |= 0b000100;
                        return enableBus;
                case 3:
                        enableBus |= 0b001000;
                        return enableBus;
                case 4:
                        enableBus |= 0b010000;
                        return enableBus;
                case 5:
                        enableBus |= 0b100000;
                        return enableBus;
                default:
                        return enableBus;
        }

}
```

Moreover, the user inputs from the FPGA have also to be decoded. Another decoder function has been used in this system to decode the user input signals into integers that can be used in the algorithm of the game. The C code below shows the main function of the input decoder function.

```
int keyDecoder(int key) {
        switch(key) {
                case 0b00000001:
                        return 1;
                case 0b00000010:
                        return 2;
                case 0b00000100:
                        return 3;
                case 0b00001000:
```

```
                return 4;
            case 0b00010000:
                return 5; // tango
            case 0b00100000:
                return 6; // mango
            default :
                return 0;
        }
}
```

So far, all of the software functions above provide all of the tools and functionalities to construct the whole game. In the main function, the algorithm utilizes all of those software functions and interacts with the FPGA hardware modules to make the whole game. The main function consists of two major subsections: initialization section and game algorithm.

The initialization section defines the two players: one is the player him or herself, the other one is the enemy. The initialization function (as shown below) is used for creating the data type of both of the players and creating the database for the game in the SRAM.

```
volatile player JunJun;
volatile player JunE;

JunJun.name = "JUNJUN";
JunJun = initPlayer(JunJun, 0, 1, 2);

JunE.name = "JUNE";
JunE = initPlayer(JunE, 3, 4, 5);

usleep(1000);
```

A software delay was added after the initialization function to ensure that the database is successfully built. The system then needs to keep track of a few states for both players. Those states are the player's current pokemon's maximum hp, maximum mp, current hp, current mp, current pokemon's number, the enemy's current pokemon's maximum hp, current hp. In order to keep track of those states, the function that will retrieve the data from the SRAM can be used, and the address of that data can be found from the function that calculates the address of those data. The C code below is an example that shows how the system keeps track of the player's current pokemon's current hp.

```
currentHP = getValue(getAddress(JunJun.currentMon, 0, 'h', PLAYER_1));
```

The second subsection is the game's algorithm. The algorithm lies in a while loop in the main function. For the game, there are two rounds--attacking round and defending round. The attacking round allows player's pokemon to either deal damage to the enemy or use items. In the defending round, the player's pokemon will get attacked or not if the the enemy decides to use items to heal his or her own pokemon.

In the attacking round, the system requires the user's input by decoding the user inputs. If the user does not input any signal, the "key" which is the variable that represents the user input will

stay 0. At the moment the user input the signal to the system, the key decoder will decode the user input and returns an integer assigning to the “key” as shown below.

```
while(0 == key) {
        key |= *userin;
        key = keyDecoder(key);
 }
```

The “key” has 7 potential values: 0 for no user input, 1 to 4 for four pokemon’s moves, 5 for using healing potion, and 6 four using mana potion. If the “key” is either 1,2,3 or 4, the player then will do damage to the enemy. The algorithm will then calculate the cost of the move the player chose by getting the cost value from the database in the SRAM. If the cost is less than the current mp, the damage will be stored into a variable and the mana points in the database will be updated. Otherwise, the damage will not be made, all of the other data will keep the same. The C code below shows the logic in this section.

```
healH = 0;
healM = 0;
cst = getValue(getAddress(JunJun.currentMon, key, 'c', PLAYER_1));
if(cst <= currentMP) { // have enough mana
        dmg = getValue(getAddress(JunJun.currentMon, key, 'd', PLAYER_1));
        currentMP -= cst;
        setValue(getAddress(JunJun.currentMon, 0, 'm', PLAYER_1), currentMP); // Update mp
} else { // don't have enough mana
        alt_printf("don't have enough mana\n");
        dmg = 0;
        healH = 0;
        healM = 0;
}
```

If the “key” is either 5 or 6, the user is then using the items in this round. The logic for using items is the same except the data they use is distinct. The healing potion will be taken as an example to illustrate the logic for this subsection. If the “key” is assigned to 5, the system will be notified that the user wishes to use the healing potion to heal his or her pokemon. It will keep the variables that stand for damage, and mana healing to be 0, and retrieve the amount of healing potion left from the database by the command below.

```
tangoA = getValue(getAddress(JunJun.currentMon, 0, 'l', PLAYER_1));
```

If the amount of the healing potion is greater than 0, then there are enough healing potions to be used. The system will then decrement the amount of the healing potions by 1, and store the health healing points to a variable. Otherwise, the system will keep all of the variables like health healing, mana healing, and damage to be 0. The C code below shows the logic of this subsection.

```
dmg = 0;
healM = 0;
tangoA = getValue(getAddress(JunJun.currentMon, 0, 'l', PLAYER_1)); // get tango's amount
```

```
if(tangoA > 0) { // have enough tango
        setValue(getAddress(JunJun.currentMon, 0, 'l', PLAYER_1), tangoA - 1); // decrement
tango's amount
        healH = getValue(getAddress(JunJun.currentMon, 0, 't', PLAYER_1)); // get tango's heal
} else {
        alt_printf("Don't have enough tango\n");
        healH = 0;
        healM = 0;
        dmg = 0;
}
```

After the damage, health healing, or the mana healing are stored to the variable. The damage, and the health healing will be sent to the other system by the writing function. If all of them are 0, the logic will go back to the beginning of the while loop to restart this round. Otherwise, the health healing will be added to the current hp of the current pokemon. In case that the healing points will go over the maximum points, the health is capped and the actual health healing points is calculated. Similarly with mana points. The logic is shown below.

```
if(currentHP + healH > maxHP) { // reach max HP
         healH = maxHP - currentHP;
}
setValue(getAddress(JunJun.currentMon, 0, 'h', PLAYER_1), currentHP + healH); // set
currentHP

if(currentMP + healM > maxMP) { // reach max MP
        healM = maxMP - currentMP;
}
alt_printf("healM = %x\n", healM);
setValue(getAddress(JunJun.currentMon, 0, 'm', PLAYER_1), currentMP + healM);
```

The damage and the health healing points will be written to the other system by the write function the system provided. There is a software delay between two write function to ensure that the data sent will not overlap with each other. The C code below shows the write operation.

```
write(dmg);
usleep(1000);
write(healH);
```

After the damage and the health healing points are sent, the enemy's current state will be updated locally. The new enemy's current hp will be calculated depending on the damage the player's current pokemon does. If the enemy's current pokemon's hp reaches 0, the pokemon faints, and will be switched to the next one, so that the enemy's current pokemon number will increment by 1. In this case, the new maximum hp for enemy's current pokemon will be updated. If the enemy

does not have enough pokemon, the enemy's current pokemon's hp will be set to 0, and the game will be ended by terminator to escape the main loop. The logic can be seen below.

```
if(dmg != 0) {
        currentEHP -= dmg;
        if(currentEHP <= 0 && JunE.currentMon != 2) {
                currentEHP = 0;
                 setValue(getAddress(JunE.currentMon, 0, 'h', PLAYER_2), currentEHP);
                 JunE.currentMon++;
                enemyMaxHP = getValue(getAddress(JunE.currentMon, 0, 'h', PLAYER_2)); //
get new Enemy's max HP
                alt_printf("You killed enemy's pokemon.\n");
        } else if(currentEHP <= 0 && JunE.currentMon == 2) {
                *enable2 = 0;
                 setValue(getAddress(JunE.currentMon, 0, 'h', PLAYER_2), 0);
                *currentehp = 0;
                 alt_printf("You win\n");
                 endgame = 1;
        } else {
        setValue(getAddress(JunE.currentMon, 0, 'h', PLAYER_2), currentEHP);
        }
}
```

Finally, the game needs to switch phase by changing the 'round' variable to 1 in order to loop into the defending round.
In the defending round, the system will wait for the data to be passed from the enemy by two read functions. The first data will be the damage, and the second one will be enemy's health healing points. Between the two read functions, there is a short software delay to ensure that the data will not overlap with each other. However, the delay cannot be greater than what was added between two write function in attacking stage, or the second read function will miss the data. The C code below shows the reading operation in the defending round.

```
inDmg = read(); // get damage
usleep(500);
eHeal = read(); // get enemy's heal
```

Then, depending on the data the system reads, there are three cases:

1. The damage is non-zero, the health healing is zero.
2. The damage is zero, the health healing is non-zero.
3. Both damage and health healing are zero.

For case 1, the player's pokemon get attacked by the enemy's pokemon. There will be a animation effect that will flash the screen for a half second. In order to do that, the system will

send a signal to the graphic driver for a half second. The C code below shows the commands that do this job.

```
*getattack = 1;
usleep(500000);
*getattack = 0;
```

Then, the current state will be calculated for player's pokemon. The current hp will be decremented by amount of the damage read. If the current hp reaches 0 or lower than 0, the system will set the current hp to 0 in the database, and switch to the next pokemon by incrementing the current pokemon number by 1. It will also update the new maximum hp and maximum mp. However, if the player does not have more pokemon, the current hp will be set to 0, and the end game terminator will be trigger to 1 and escape the main loop. Otherwise, it will set the current hp to the database. Finally, the round will be switched by flipping the round variable. The logic can be seen from the code below.

```
currentHP -= inDmg;
if(currentHP <= 0 && JunJun.currentMon != 2){ // my pokemon is dead
	setValue(getAddress(JunJun.currentMon, 0, 'h', PLAYER_1), 0); // set current mon's hp
to 0
	JunJun.currentMon++; // increment currentMon
	maxHP = getValue(getAddress(JunJun.currentMon, 0, 'h', PLAYER_1));
	maxMP = getValue(getAddress(JunJun.currentMon, 0, 'm', PLAYER_1));
	alt_printf("Your pokemon is killed.\n");
} else if(currentHP <= 0 && JunJun.currentMon == 2){
	*enable1 = 0;
	*life = 0;
	*currenthp = 0;
	alt_printf("You lost\n");
	endgame = 1;
} else {
	setValue(getAddress(JunJun.currentMon, 0, 'h', PLAYER_1), currentHP);
}
```

Case 2 is the case that the enemy is healing his or her pokemon. The system will then increment the current pokemon's health points by the amount of the health healing points read from the other system, and set it to the database. After that, the system will then switch the round. The code below shows the logic for case 2.

```
setValue(getAddress(JunE.currentMon, 0, 'h', PLAYER_2), currentEHP + eHeal); // enemy uses
tango
round = 0;
```

Case 3 is when the enemy uses a mana potion to heal his or her pokemon. There will be no change for local current states other than the round switching. Thus, the round will be switched but any other variables will not be changed as shown below.

round = 0;

System Description

As mentioned before, this design is to develop an interactive battle game based off of Nintendo's "Pokemon". The whole system consists of four main modules: SRAM module, communication module, graphic driver module, and the game module. SRAM module, communication module, and graphic driver module are hardware module that are built on FPGA board. However, the game module is a software module built on the NIOS II microprocessor. In this section, each module will be discussed separately to specify each module's inputs, outputs, and side effects.

- SRAM

  An *8×2048* SRAM was developed in order to store the database of the game module. The SRAM allows users to write 8-bit data to the SRAM at specific addresses, and store the data at that address unless the user overwrites it. And also, the user is able to read the data at a specific address from the SRAM.

  Inputs:

  As discussed in design procedure, the inputs of the SRAM are an 8-bit data "data_I", an 11-bit address input "addr", a 1-bit read and write input "wr", a 1-bit enable input, and a clock line.

  Outputs:

  The output will be an 8-bit data "data_O". The data will be output while the user input a 0 to the read and write line.

  Side Effects:

  To properly write the data to the SRAM, the address must be assigned before the read and write line is set to high. Otherwise, the address may stay the previous address, so that the data will be written to an incorrect location.

- Communication module

  A communication module is developed to pass the data between two systems. The communication module takes 8-bit data through an 8-bit wide bus and passes it out

serially. Also, it can receive serial data and processes it as 8-bit wide data in parallel.

Inputs:

The input of the communication module has a clock input with a frequency of 50 MHz, a reset input, a transmit enable input that will enable the shift register to shift the parallel data out in serial, a load input that will load the parallel data into the shift register, a serial data input line which takes the serial input data from the other system, and an 8-bit parallel data which takes the 8-bit data from the microprocessor and load it to the shift register.

Outputs:

This module has four outputs: a 1-bit output data that is the data shifted out from parallel to serial shift register, a 1-bit "char_sent" output that will be pulled high when a data is fully sent, a 1-bit "char_received" output that will be pulled high when a data is fully received from the other system, and an 8-bit data that will output parallel data to the microprocessor when an 8-bit data is fully received and stored in the shift register.

Side Effect:

The timing constraint in the communication module is extremely strict. The clock of the shift register is generated by an 8-bit counter. In order to set the baud rate the same for both parallel to serial shift register and the serial to parallel shift register, the clocks for both shift registers are set to the inverse of the fourth bit of the bit counter in the communication module. For the parallel to serial shift register, the register will not store any data while loading data since the clock will not work while loading data. As a result, the parallel to serial shift register requires another clock when the user wants to load the data into the shift register. In this system, the second clock uses a clock with a frequency of 50 MHz. Due to the fact that the data will be reset to all 1s after the clock stops working, the moment the shift register outputs the parallel data to the microprocessor should be right before the the counter's first four bit counts to 1010. As a result, in this design, the moment of reading the data is set to the time when the counter counts to 10011000 in binary so that the data in the serial to parallel shift register can be properly read.

- Display driver module

  Moreover, the game is displayed on screen via VGA port, thus a display driver is developed to graphically represent all of the pokemon, pokemons' health points, pokemons' mana points and all of the moves the pokemon has access to. Depending on the user input (via keys and switches), the data is passed into the module and the display will alter simultaneously.

- Game module

As mentioned before, this game is similar to a well known game “Pokemon” which is a fighting game between two “Pokemon”. It is a turn based battle where each pokemon will attack its enemy at the attacking round and get attacked in the defending round, and switch round over and over. In the attack round, the player can choose either attack the enemy, or use items to heal his or her “Pokemon”. However, if the player uses items, two kinds of items can be used for different purposes which are “Health potion” and “Mana potion”. The “Health potion” will heal the pokemon. If the pokemon reaches its maximum healing points, its healing points will stay at maximum healing points. Similarly, the “Mana potion” will recover the pokemon’s mana points. The mana points will stay maximum if the the recovering points make the mana greater than the maximum mana points. Each player has three pokemon in total, if one of them has fainted, the system will automatically switch to the next one. All of the moves will be replaced depending on what the next pokemon is. However, the inventory of “Health potions” and the “Mana potions” will stay the same. If the player loses all of his or her pokemon, he or she then loses the game.

Inputs:

The inputs of this module are all from output of the FPGA modules: an 8-bit data input from SRAM module that will pass the data from the SRAM to the microprocessor, another 8-bit data input that will pass the data received from the other system through the communication module to the microprocessor, a 1-bit input “char_sent” that represents the data the user wants to send to the other system is fully sent, a 1-bit “char_received” that represents the data the user received from the other system is fully received, and a 6-bit user input that is taken from the graphic driver module represents the user’s options while he or she is under attacking round.

Outputs:

The outputs of this module are the inputs of the FPGA modules. The outputs include a 11-bit address output that specifies the address on SRAM to store and read data, an 8-bit data output for SRAM that will pass the data to the SRAM, a 1-bit read and write output that will change the read/write mode of the SRAM, a 1-bit “transmit_enable” output that will enable the parallel to serial shift register in the communication module to send the data in the shift register out in serial, an 8-bit data output that will load the data to the parallel to serial shift register, a 1-bit load output that will allow the user to load data to the shift register, two 6-bit pokemon enable output to enable the graphical display for player and enemy’s current pokemon, three 8-bit state output that will update the current states of pokemon like healing points and mana points, and display them on the monitor, a 3-bit output that will pass to the graphic driver to display the number of pokemon left.

Side Effects:

In this module, the timing issue depends on hardware modules like SRAM module and the communication module. As mentioned before, for SRAM, the address must be assigned before the read and write output to be 1. In the game module, there are two data

groups to be passed in every round. In the attacking round, there is a software delay to insure FPGA module will not miss the second data to be loaded to the shift register. On the other hand, in the defending round, there is a software delay to ensure that the communication module will not miss the second data to come in.

Software Implementation

The game basically has two rounds: attacking round and defending round. In the attacking round, the player can make move to attack the enemy or use item. In the defending round, the player will get attack or the enemy's pokemon will heal itself if the enemy uses the items. To fulfill this game rule, a software program is built based on NIOS II microprocessor. The top level software block diagram is shown below in Figure 5.

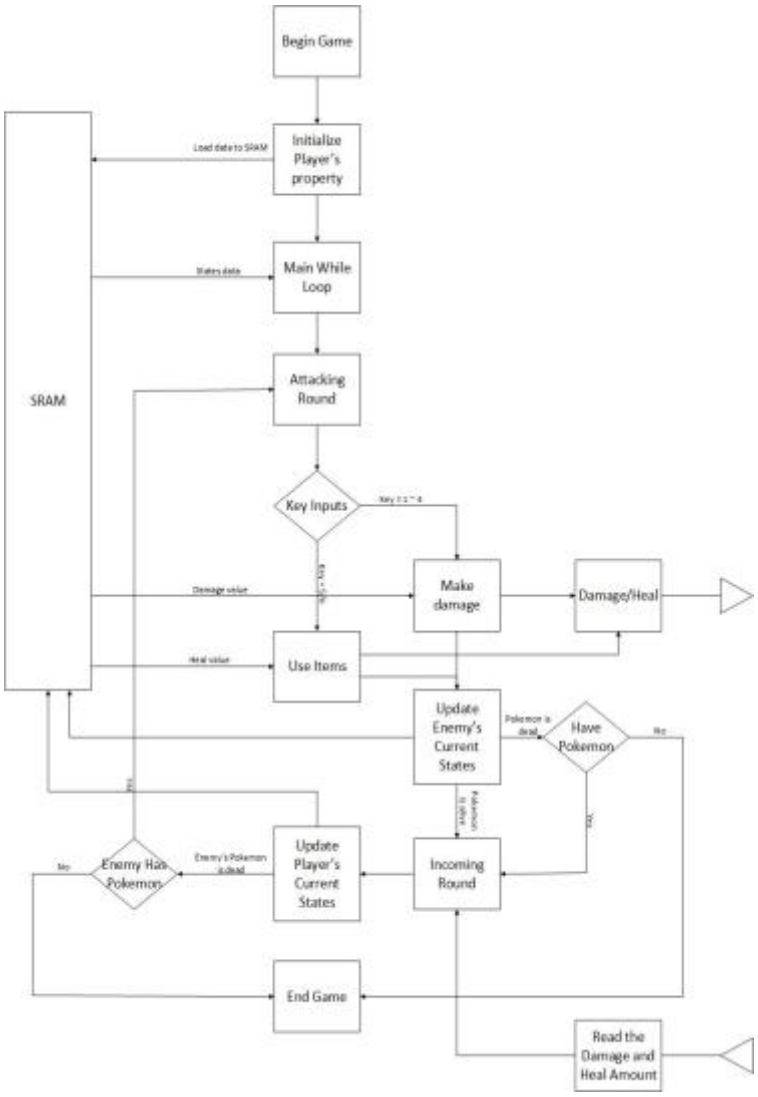

Figure 5-Software Program Flow Chart

Hardware Implementation

The system consists of four hardware modules including SRAM module, communication module, graphic driver module and system control module. The system control module is built based on the NIOS II microprocessor with 8K bytes an on-chip memory. The SRAM module and the NIOS II's on-chip memory are the memory bases of the whole system. The graphic driver will output the graphic display through the VGA port on the monitor. The communication will transfer the data to the other system in serial, and also it will receive the data in serial from another system. The system has an analog to digital converter to take the input from three analog keys on FPGA and transfer the analog voltage to digital input signals. The top level block diagram is shown in Figure 6 below.

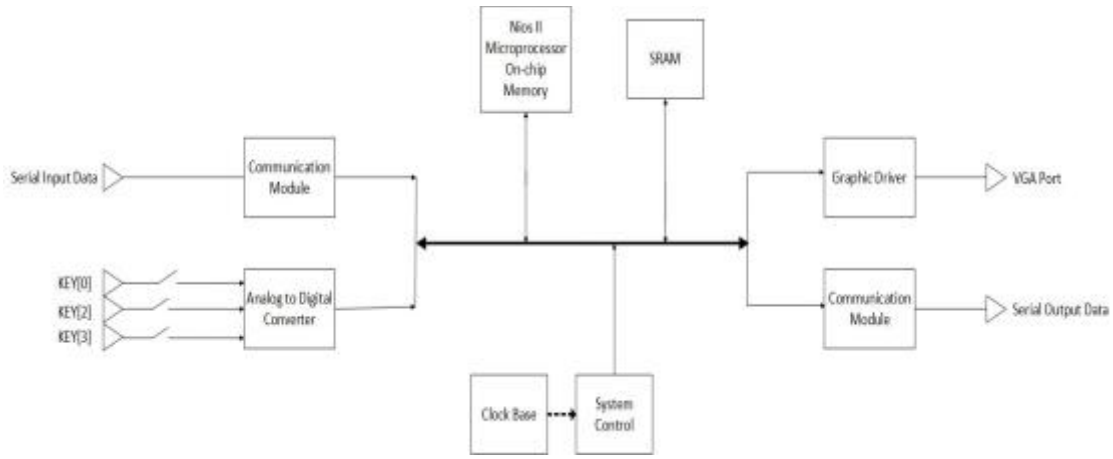

Figure 6-Hardware Top Level Block Diagram

The actual schematic of the system is shown in Figure 7. The graphic driver module has eight outputs through VGA port. For the communication module, it uses a GPIO pin as a serial output pin of the system. Also, it uses another GPIO pin as a serial input pin of the system.

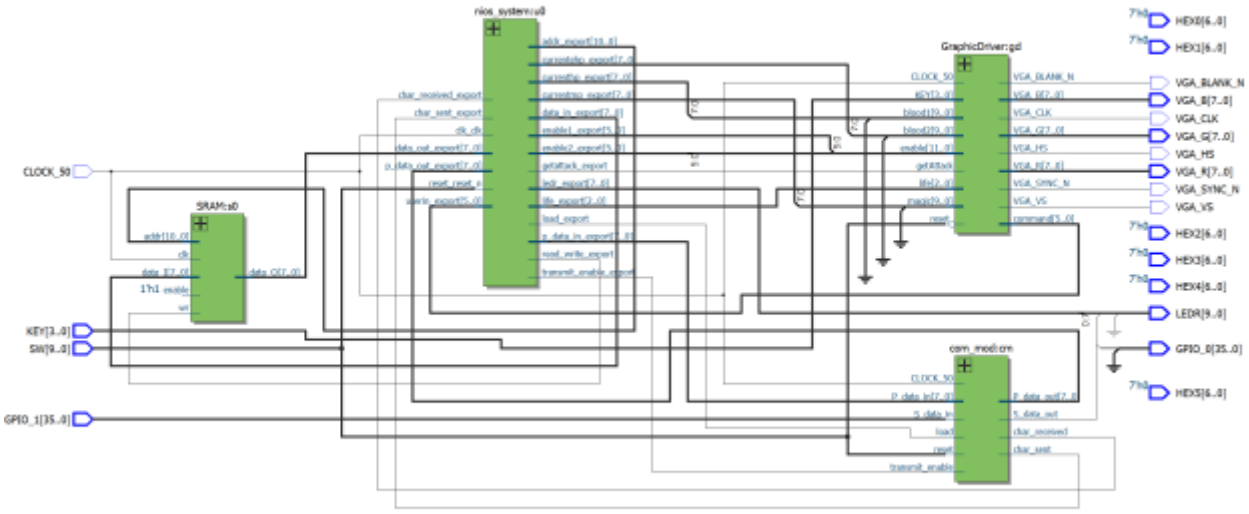

Figure 7-Circuit Schematic of the "Pokemon" Project

Test Plan

The team first tested the communication system. In the sending procedure, the clock signal for the parallel to serial shift register, the buffer data signal in the parallel to serial shift register, the serial data output signal, the load signal sent by the Nios II, and the char sent successfully signal had been tested.

For the receiving procedure, the clock signal for the serial to parallel shift register, the receive enable signal, the buffer data in the shift register, the serial input signal, the parallel output signal, and the char received successfully signal had been tested.

Then the team tested the Pokemon game that they designed. In this project, the team monitored the data that stored in the SRAM, the data that transferred between boards, and the graphic output that the team use VGA to display on the screen.

Test Specification

The parallel output signal should have 8 bits. The buffer in the serial to parallel shift register should have 10 bits, and the buffer in the parallel to serial shift register should have 9 bits. All the other signals should have 1 bit.

For the sending procedure, after the Nios II gave a load signal pulse, the clock signal for the shift register should start working. The clock signal should have 10 falling edges, and then it shoulds stay at high. The shift register should start working simultaneously. The DFF in the shift register should shift one bits towards the 9th bit per clock falling edge, and the the 10th bit in the buffer in the shift register should output its data to the serial output. In other words, the 10 serial data signal in the each clock falling edge should be equal to the data in the first state of the shift register buffer. The last bit that output to the serial data should always be 1, and the first bits should always be 0. At the 10th falling edge, which means all the 10 bits are shifted out, the char sent signal should turn to 1.

For the receiving procedure, the serial data in signal should be exactly the same with the serial data out sent by the sending procedure. The procedure should start after the first bit has been detected and the receive enable signal should turn to 1. The clock signal should start working right after, and the clock signal should have 10 falling edges. The shift register should start working simultaneously. At each clock falling edge, the 0 bit in the shift register should equal to the data of the serial data in signal at that falling edge, and it will shift towards 9th bit at each falling edge. At the 10th falling edge, the char received signal should go to 1, which means all 10 bits of the data have been received. Simultaneously, the 9th to the second bits in the buffer should output the signal to the parallel output signal. The data in the parallel output should only hold for one second, and then all the bits should go back to 1. After that, the receive enable signal should go back to 0, which means one receiving procedure is done.

For the Pokemon game, KEY[0], KEY[2], and KEY[3] are the inputs of the system, and the KEY[2] and KEY[3] are the controller for the menu bar. These two keys will allow the user to switch between each bar in the menu. When the user press KEY[2], the arrow in front of the bar should move left. When the user press KEY[3], the arrow in front of the bar should move right. Key[0] is the confirm button, which means when the user press KEY[0], the user want to select the bar that have the arrow in front of it. The system should receive which item or skill the user have been select. Then the system should react to the damage and cost of the skill or the recovery with an item. If the user selects a skill, the system will send the damage to the other board with 0 for recovery, and if the user select an item, the system will send the recovery with 0 damage to the other board. After that both system should start computing their own and their enemies' information, and these data would be written to the SRAM for updating. Our graphic driver then will grab the data from the SRAM to display that on the screen . For testing, the system also will print out the data in the console as well. After one system finished attacking, the other system should start the attacking round.

Test Cases

The team tested the communication system by SignalTap and gtkwave, and tested the Pokemon game by directly loading the program onto the DE1_SoC board and looking at the print out data on the console and the graphic output on the screen.

Test Setup

For testing the communication system, the team first connected two DE1_SoC board together by connecting TX to another board's RX and connecting RX to another board's TX. The team then loaded the verilog design onto both boards and ran the C program. The team tested the communication system by sending the character 'a' back and forth, and monitored the input and output wave forms on the SignalTap.

For testing the game, the team keep the connection of the communication system and connected two VGA cable to two screens for observing the graphic outputs. Besides that, for monitoring the data flow and the data in the SRAM, the team added many print out commands in the C program to print the data that received and sent and the data that stored in the SRAM. The program will also print out the current HP and MP for both players.

Identification of Signals and Values to be Applied as Inputs to the System

For the sending procedure in the communication system, the input is the character, which is 8bits plus one starting bit and one ending bit, that the user enters in the console. For the receiving procedure, the input signal is the serial output data from the sending procedure.

For the Pokemon game, the inputs are KEY[0], KEY[2], and KEY[3] on each DE1_SoC board.

Outputs Signals to be Measured

For the sending procedure in the communication system, the output signal is the serial output data of the shift register. For the receiving procedure is the parallel output data of the shift register.

For the Pokemon game, the team tested the print out outputs on the console that demonstrated the data in the SRAM and the data that received and sent. The team also let the program print out the current HP and MP for both players to see if the program did the computation correctly. Another output signal is the DE1_SoC board's graphic output on the screen through VGA cables.

Expected Results

For the sending procedure in the communication system, the 10 bits of the serial output signal at 10 clock falling edge should be the same with the binary ASCII number of the character that the user input in the console plus a starting bit, which is 0, and a ending bit, which is 1. For example, if the user entered ‘a’, which binary ASCII number is 8’b01100001, in the console, the serial output should be 10’b0011000011.

For the receiving procedure in the communication system. The output signal should be exactly the same with the binary ASCII number of the character that entered by the user. The control path’s expected results were stated in the previous Test Specification section.

For the Pokemon game, when the user selects a skill, the system should send the damage to the other board with 0 recovery. On the console, it should print out the data that receive and sent, and the data in the SRAM. The received data should be exactly the same with the data that been sent by the other system. The graphic driver should output corresponding HP and MP for their own and the enemy.

Pass/Fail Criterion

The system has met all the expected results to pass the test.

Presentation, Discussion, and Analysis of the Results

Results

The team used gtkwave to test their verilog design first. Figure 8 below shows the waveform that is generated by the gtkwave. The P_data_in signal is the input of the sending procedure, and the P_data_out signal is the output of the receiving procedure. From the figure, the system have a input of 61 at the beginning of the test, and after several clock cycles, the p_data_out output 61 as well. This means the verilog design of the communication system is correct.

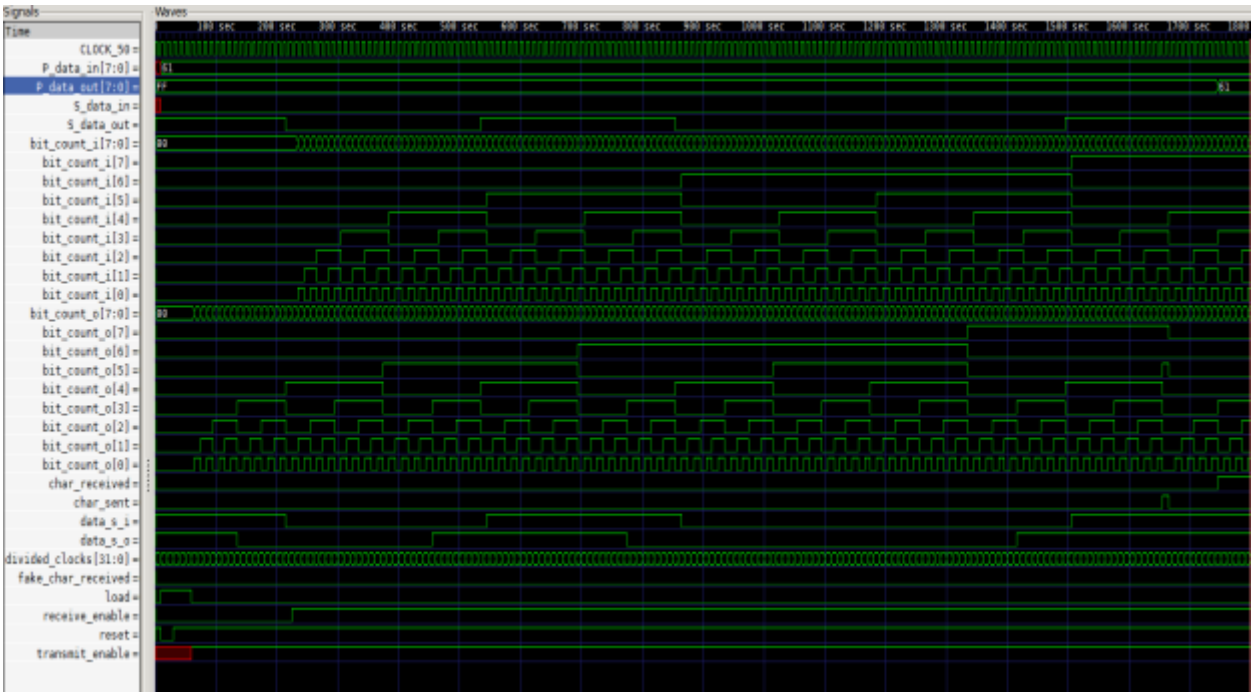

Figure 8-Waveform of the Communication Module by "gtkwave"

Then the team use SignalTap to test the signal on the DE1_SoC board. Figure 9 below shows the waveform from the SignalTap of the sending procedure. In the C program, the team sent a character 'a' for testing. From the figure, after the load goes to 1, the data has been loaded to the shift register, and the data is 10'b0011000011, which is character 'a''s binary ASCII code plus one starting bit and one ending bit. The clock starts working simultaneously, and it has exactly 10 falling edges. At each falling edge, the buffer in the shift register shift towards the 10th bit, and the serial output signal is equal to the 10th bit. After 10 falling edges of the clock signal, the 10 bits are successfully shifted out. By marking the serial output value at every falling edge, the data is the same with 10'b0011000011, which means our sending procedure work as the team expected and pass the test.

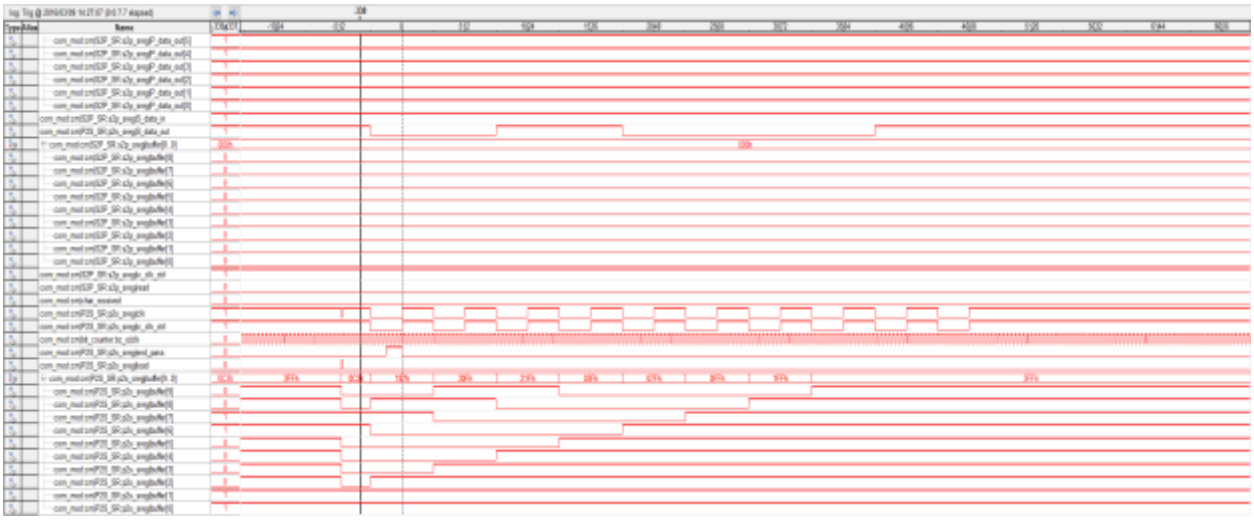

Figure 9-Waveform of Sending Procedure of the Communication Module

Figure 10 below shows the waveform of the receiving procedure from the SignalTap. From the figure, when the system detects the starting bit, the receive enable signal goes to high, and the the clock of the shift register starts working. At the same time, the serial input data shifts into the first bit of the shift register. At each falling clock edge, the serial input data shifts into the first bit of the buffer in the shift register, and the shift register shifts towards the 9th bit. At the 10th falling age, the first 8 bits in the buffer output their values to the parallel signal, which is 8'b01101111. This is the same with the binary ASCII code of the character 'a'. Thus the receiving procedure is working properly and pass the test.

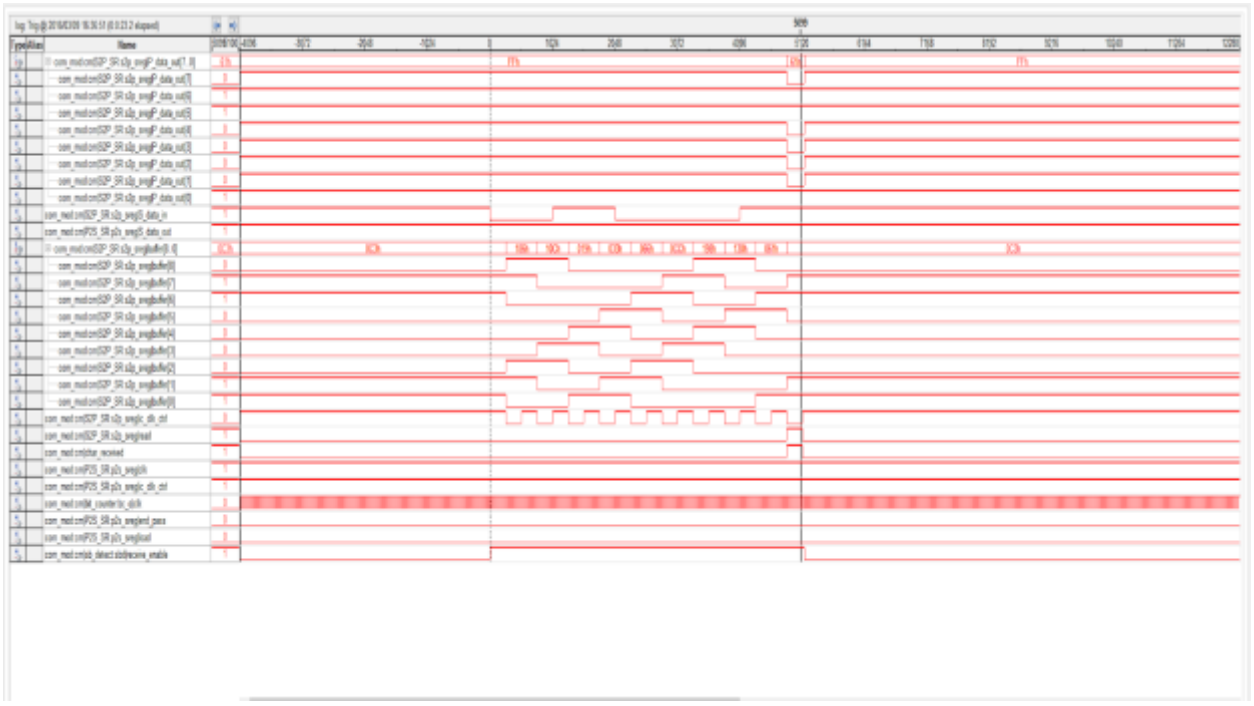

Figure 10-Waveform of Receiving procedure of the Communication Module

The following figures are the console print out of when the game is running. All the numbers below are in hex. The number array in the following figures are the data in the SRAM. The team stored every character's health, mana, skills damage and skills cost in the SRAM. Based on the memory allocate function that the team wrote, each data is written in the expected slot, which means the SRAM output is correct.

Figure 11 below shows the console print output at the beginning of the game. Besides the data in the SRAM, the program also printed out the player's current health and mana and enemy's current health. For this test case, the player's full health is 100, full mana is 90, and enemy's full health is 110.

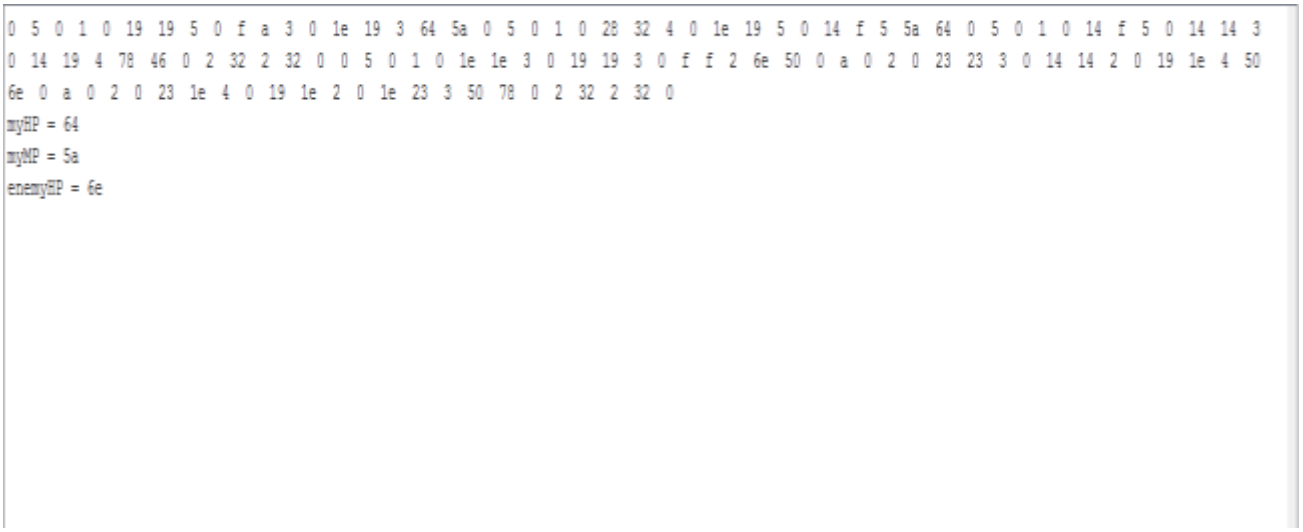

Figure 11-Console Print Out of the Starting of the Game

Figure 12 and Figure 13 below shows the console print outputs when the player is attacking and when the player is being attacked. When attacking, the system would print out the damage, cost and the recovery of the skill. From figure 12, for this test case, the skill's damage is 5 and the cost is 0. Figure 13 shows that the other system receive the damage and minus this damage from the current health.

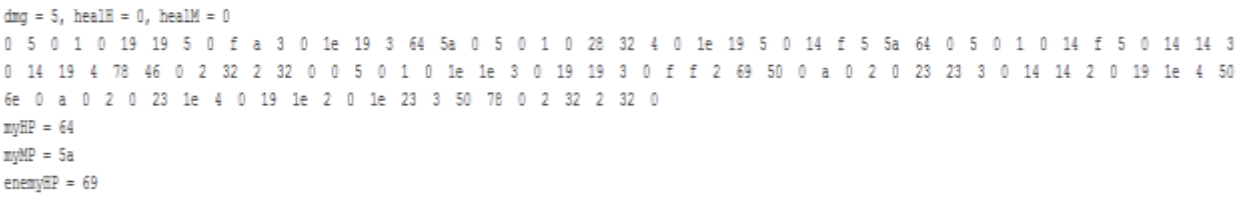

Figure 12-Console Print Out When Attacking

```
inDmg = 5, eHeal = 0
0 5 0 1 0 19 19 5 0 f a 3 0 1e 19 3 5f 5a 0 5 0 1 0 28 32 4 0 1e 19 5 0 14 f 5 5a 64 0 5 0 1 0 14 f 5 0 14 14
3 0 14 19 4 78 46 0 1 32 2 32 0 0 5 0 1 0 1e 1e 3 0 19 19 3 0 f f 2 6e 50 0 a 0 2 0 23 23 3 0 14 14 2 0 19 1e
4 50 6e 0 a 0 2 0 23 1e 4 0 19 1e 2 0 1e 23 3 50 78 0 2 32 2 32 0
myHP = 5f
myMP = 5a
enemyHP = 6e
```

Figure 13-Console Print Out When being Attacked

Figure 14 below shows the console print output when the player use an item. From the figure, the damage of the item is 0, the healM, which is the recovery, is 32 in hex.

```
dmg = 0, healH = 32, healM = 0
0 5 0 1 0 19 19 5 0 f a 3 0 1e 19 3 4b f 0 5 0 1 0 28 32 4 0 1e 19 5 0 14 f 5 5a 64 0 5 0 1 0 14 f 5 0 14 14
3 0 14 19 4 78 46 0 1 32 2 32 0 0 5 0 1 0 1e 1e 3 0 19 19 3 0 f f 2 55 50 0 a 0 2 0 23 23 3 0 14 14 2 0 19 1e
4 50 6e 0 a 0 2 0 23 1e 4 0 19 1e 2 0 1e 23 3 50 78 0 2 32 2 32 0
myHP = 4b
myMP = f
enemyHP = 55
```

Figure 14-Console Print Out when Using Item

Error Analysis

The biggest issue in this design is the timing issue of the communication module. As mentioned before, the timing constraint in the communication module is extremely strict. The clock of the shift register is generated by an 8-bit counter. In order to set the baud rate the same for both parallel to serial shift register and the serial to parallel shift register, the clocks for both shift registers are set to the inverse of the fourth bit of the bit counter in the communication module. For the parallel to serial shift register, the register requires two clocks for storing the data into the shift register and sending the data out in serial. Since the clocks are different (one of them is a clock with frequency of 50 MHz), it initially causes the data to be shifted 11 times (10 times are required). The problem can be solved by restricting the time when the clock is selected. In detail, the shift register will only select the 50 MHz clock when the data needs to be loaded to the shift register. For the serial to parallel shift register, due to the fact that the data will be reset to all 1s after the clock stops working, the moment the shift register outputs the parallel data to the microprocessor should be right before the the counter's first four bit counts to 1010. As a result, in this design, the moment of reading the data is set to the time when the counter counts to 10011000 in binary so that the data in the serial to parallel shift register can be properly read.

Analysis of Why the Project may not of Worked and What Efforts were Made to Identify the Root Cause of Any Problems

As mentioned in previous section, the timing issue is the most serious problem that may cause the whole system stop working, because this is a game based on the communication module. In detail, the failure of passing data through the communication module, and receiving it back from the communication module properly will definitely affect the outcome of the whole project. In order to make the communication module a solid and stable communication hardware module, the team spends most of time on monitoring the waveform of the shift registers in the communication module while they are transmitting data and receiving data by using Signaltap. To identify the root cause of the problems, the team compares the simulation waveform and the Signaltap waveform to identify the potential sub-module that may cause the problem, and re-checking the logic and the timing constraints in that module.

Summary

In this project, team set out to make a primitive version of Pokemon using an FPGA and NIOS II processor. Visuals are displayed on a VGA screen and user input is made via the buttons on the FPGA board. Communication is handled between two boards to allow for another user to play as the opponent with all the same features available to the second player. Information such as pokemon health, mana, skill set and more were stored on our designed SRAM as well.

Conclusion

The project is quite successful and all of the goals are met. The players are able to play the game entirely with just what is displayed on the monitor. The turn based battle system works well and the modularity of the design affords the team to ability to add further features in the future, such as skill accuracy.

Appendix

All of the C and the Verilog Code have already been uploaded to dropbox. Refer to the dropbox for details.

Contribution Table

System Design	Yuran Wu, Jun Hao & Yi-hsin (Chris) Jong
Hardware Design	Yuran Wu & Jun Hao
Software Design	Yuran Wu, Jun Hao & Yi-hsin (Chris) Jong
Simulation	Yuran Wu & Jun Hao
Data Analysis	Yuran Wu & Jun Hao
Report Write up	Yuran Wu, Jun Hao & Yi-hsin (Chris) Jong