Alberto Diaz - UID: 604967268

Evan He - UID: 705619333

Brandon Truong - UID: 705326387

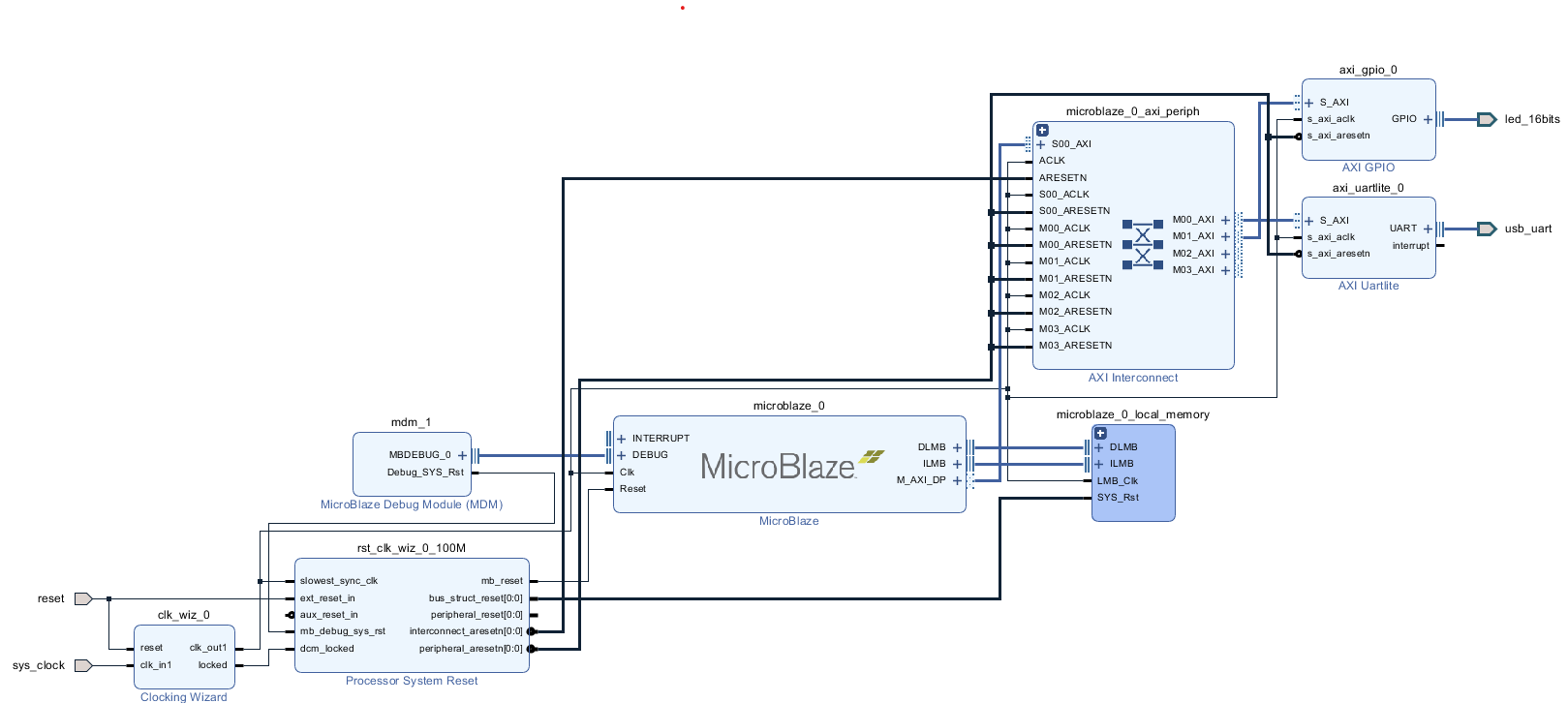
Lab 3 Report

**Introduction**

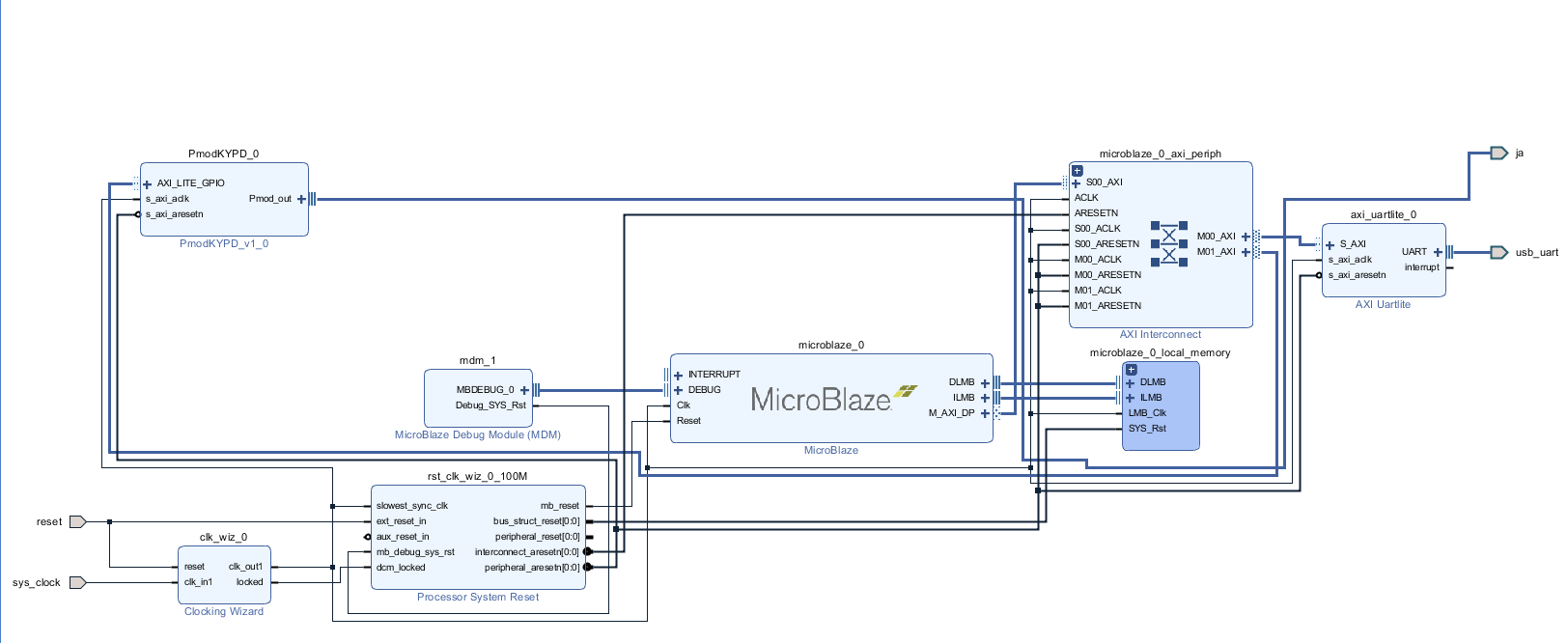
This lab involved implementing several small projects using a synthesized MicroBlaze processor, serial communication protocols and external Pmod hardware. Part 1 was a “hello world” demo to set up the environment, part 2a was a multiplication calculator using UART and the RS232 standard, and part 2b was a rock-paper-scissors simulation incorporating a Pmod KYPD module in addition to the above components.

**Schematics**

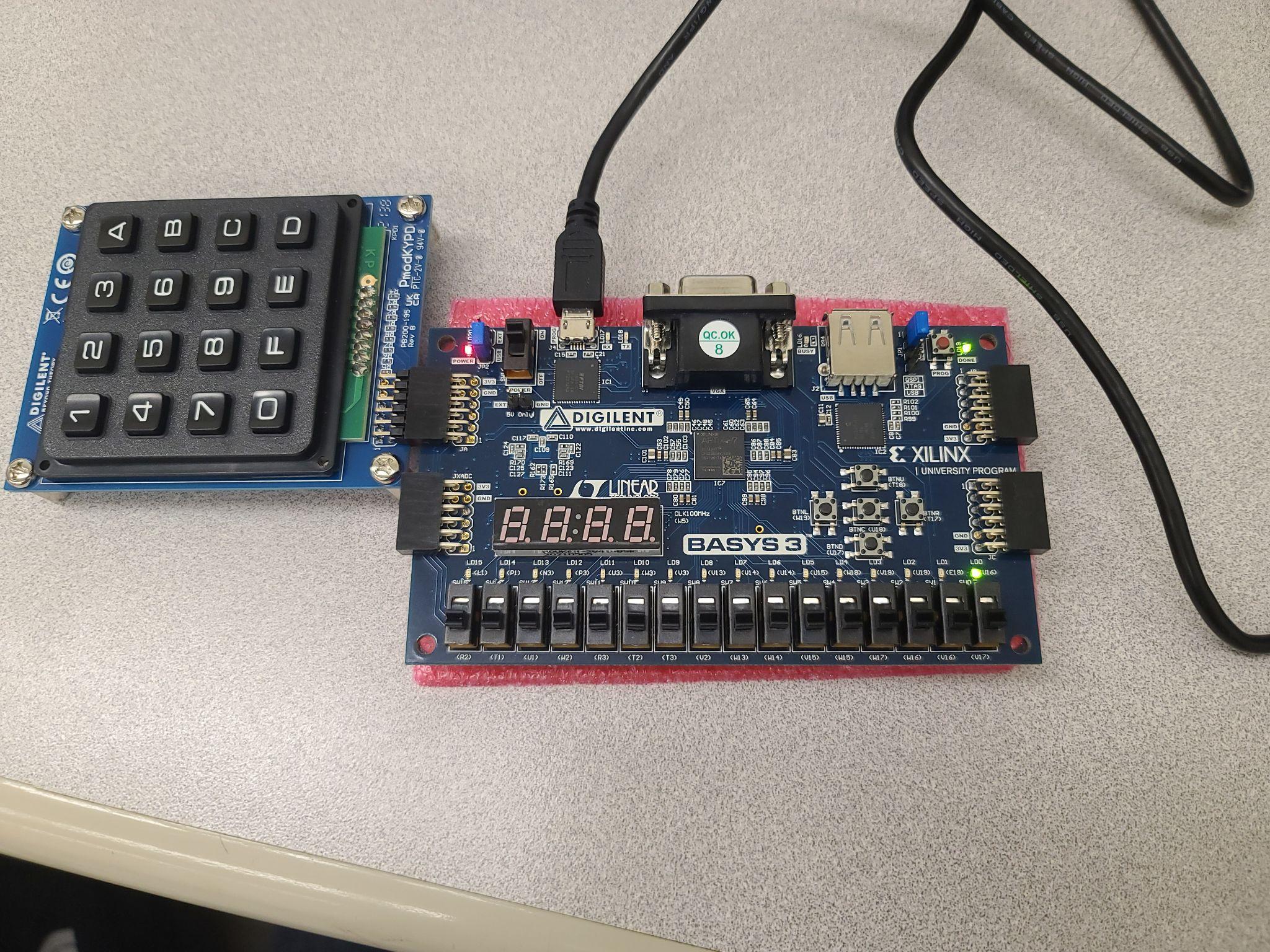
Part 2a:



Part 2b:

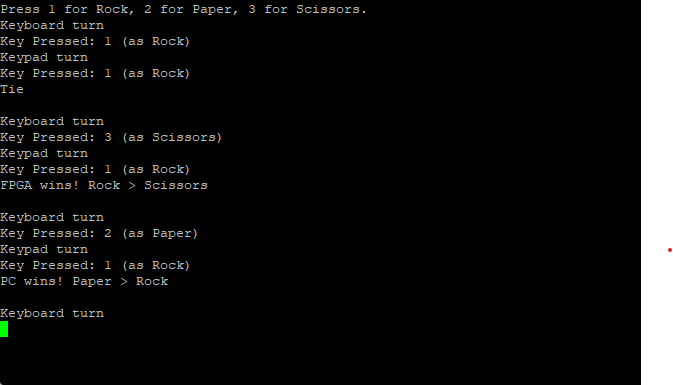


**Hardware Configuration**

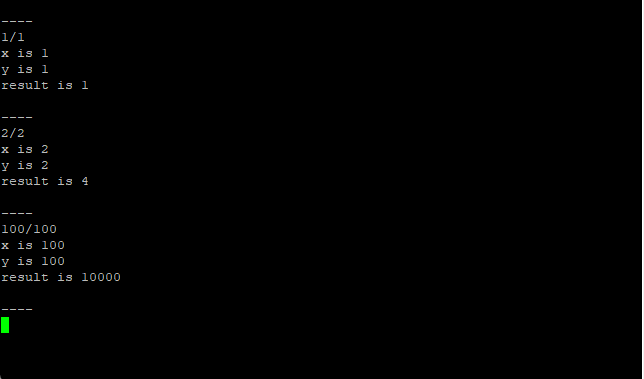


**Sample Output**

Part 2a:



Part 2b:



**Code**

Part 2a (multiply\_serial\_nums.c)

#include <stdio.h>

#include "platform.h"

#include "xuartlite\_i.h"

#include "stdlib.h"

#include "string.h"

#include "xparameters.h"

#include "xgpio.h"

#include "limits.h"

#define LED\_1 0x01

#define LED\_CHANNEL 1

#define false 0

#define true 1

//Continuously checks for inputs in the format "{int}/{int}", and outputs the resulting integer to the serial when enter is pressed

//If the value is over 100, then the led will light up

//Currently only supports integers (atoi does not handle larger than expected numbers and overflows, can use strtol instead)

int doesOverflow(int x, int y)

{

if((y > 0 && (x > INT\_MAX / y)) || (y < 0 && (x > INT\_MIN / y))) return true;

return false;

}

int main()

{

init\_platform();

XGpio gpio;

XGpio\_Initialize(&gpio, XPAR\_AXI\_GPIO\_0\_DEVICE\_ID);

XGpio\_SetDataDirection(&gpio, LED\_CHANNEL, 0x00);

while(1) {

print("\r\n----\r\n");

//Input taken

char buf[100];

scanf("%s", buf);

print(buf);

print("\r\n");

//String-Checking by finding delimiter '/'

char\* delimiter = strchr(buf, '/');

int pos = delimiter - buf;

char\* str1 = (char\*) malloc(pos + 1);

char\* str2 = (char\*) malloc(strlen(buf) - pos);

memcpy(str1, buf, pos);

memcpy(str2, delimiter+1, strlen(buf) - pos);

/\* Replacement code if we want to handle numbers larger than expected

char\* ptr;

long x = strtol(str1, &ptr, 10);

long y = strtol(str2, &ptr, 10);

\*/

int x = atoi(str1);

int y = atoi(str2);

//checks for overflow, and if found, will not do multiplication, otherwise output the result and turn led on or off depending on value

if(!doesOverflow(x, y)) {

int result = x\*y;

xil\_printf("x is %d\r\n", x);

xil\_printf("y is %d\r\n", y);

xil\_printf("result is %d\r\n", result);

//LED handling

if(result > 100) {

XGpio\_DiscreteWrite(&gpio, LED\_CHANNEL, 0x01);

} else {

XGpio\_DiscreteWrite(&gpio, LED\_CHANNEL, 0x00);

}

} else

xil\_printf("Overflow detected\r\n");

free(str1);

free(str2);

}

cleanup\_platform();

}

After setting up the GPIO channel to be an output channel, the program takes in input from stdin using the scanf function, which is stored in the 100-byte buf buffer. The user’s input is then printed to stdout. The strchr function is used to find the “/” character, which acts as the delimiter for the two substrings representing the numbers to be multiplied, and thus separate out the two substrings. These substrings are then stored into the dynamically allocated buffers str1 and str2 respectively. The atoi function then converts the strings into ints. The ints are checked to ensure that their product does not overflow using a custom doesOverflow function, and if indeed no overflow occurs, the product is actually computed (overflow detection does not rely on this computation). Both the operands and the product are then printed to stdout. This product is then checked to determine if it is greater than 100, in which case the value 1 is written across the GPIO channel to the FPGA to light up an LED. Finally str1 and str2 are freed to prevent a memory leak from occurring.

Part 2b (rock\_paper\_scissors.c)

#include <stdio.h>

#include "platform.h"

#include "xuartlite\_i.h"

#include "stdlib.h"

#include "string.h"

#include "xparameters.h"

#include "PmodKYPD.h"

#include "sleep.h"

#include "xil\_cache.h"

PmodKYPD myDevice;

#define DEFAULT\_KEYTABLE "0FED789C456B123A"

//Setup

void EnableCaches() {

#ifdef \_\_MICROBLAZE\_\_

#ifdef XPAR\_MICROBLAZE\_USE\_ICACHE

Xil\_ICacheEnable();

#endif

#ifdef XPAR\_MICROBLAZE\_USE\_DCACHE

Xil\_DCacheEnable();

#endif

#endif

}

void DisableCaches() {

#ifdef \_\_MICROBLAZE\_\_

#ifdef XPAR\_MICROBLAZE\_USE\_DCACHE

Xil\_DCacheDisable();

#endif

#ifdef XPAR\_MICROBLAZE\_USE\_ICACHE

Xil\_ICacheDisable();

#endif

#endif

}

//Keypad init

void DemoInitialize() {

EnableCaches();

KYPD\_begin(&myDevice, XPAR\_PMODKYPD\_0\_AXI\_LITE\_GPIO\_BASEADDR);

KYPD\_loadKeyTable(&myDevice, (u8\*) DEFAULT\_KEYTABLE);

}

/\* Plays the game of Rock, Paper, Scissors

Checks for input from the keyboard and keypad with mapping of

1. Rock

2. Paper

3. Scissors

Logic is handled sequentially, starting with input from keyboard and then keypad

\*/

void DemoRun() {

u16 keystate;

XStatus status, last\_status = KYPD\_NO\_KEY;

u8 key, last\_key = 'x';

// Initial value of last\_key cannot be contained in loaded KEYTABLE string

Xil\_Out32(myDevice.GPIO\_addr, 0xF);

char\* pcChoice;

char\* fpgaChoice;

xil\_printf("Press 1 for Rock, 2 for Paper, 3 for Scissors.\r\n");

while (1) {

//Keyboard Logic

int keyboardInputNotValid = 1;

int keyboardChoice;

xil\_printf("Keyboard turn\r\n");

while(keyboardInputNotValid) {

char buf[2];

scanf("%s", buf);

keyboardChoice = atoi(buf);

//Valid input checker

if (keyboardChoice < 1 || keyboardChoice > 3)

xil\_printf("Error: Invalid input\r\nKeyboard turn\r\n");

else {

switch(keyboardChoice) {

case 1: pcChoice = "Rock"; break;

case 2: pcChoice = "Paper"; break;

case 3: pcChoice = "Scissors"; break;

}

xil\_printf("Key Pressed: %d (as %s)\r\n", keyboardChoice, pcChoice);

keyboardInputNotValid = 0;

}

}

//Keypad logic

//Capture state of each key

int keypadNotPressed = 1;

int keypadChoice;

xil\_printf("Keypad turn\r\n");

while(keypadNotPressed) {

keystate = KYPD\_getKeyStates(&myDevice);

// Determine which single key is pressed, if any

status = KYPD\_getKeyPressed(&myDevice, keystate, &key);

// Print key detect if a new key is pressed or if status has changed

if (status == KYPD\_SINGLE\_KEY && (status != last\_status || key != last\_key)) {

keypadChoice = key - '0';

if(keypadChoice < 1 || keypadChoice > 3)

xil\_printf("Error: Invalid input\r\nKeypad Turn\r\n");

else {

switch(keypadChoice) {

case 1: fpgaChoice = "Rock"; break;

case 2: fpgaChoice = "Paper"; break;

case 3: fpgaChoice = "Scissors"; break;

}

xil\_printf("Key Pressed: %d (as %s)\r\n", keypadChoice, fpgaChoice);

keypadNotPressed = 0;

}

last\_key = key;

//Too many keys pressed, re-check for input

} else if (status == KYPD\_MULTI\_KEY && status != last\_status) {

xil\_printf("Error: Multiple keys pressed\r\n");

usleep(10000);

}

last\_status = status;

usleep(10000);

}

//RPS logic, check for tie first, then check for all possible combinations in which the PC player wins, and if those checks fail, FPGA must be the winner

if(keyboardChoice==keypadChoice) xil\_printf("Tie\r\n");

else if ((keyboardChoice == 1 && keypadChoice == 3) || (keyboardChoice == 3 && keypadChoice == 2) || (keyboardChoice == 2 && keypadChoice == 1))

xil\_printf("PC wins! %s > %s\r\n", pcChoice, fpgaChoice);

else xil\_printf("FPGA wins! %s > %s\r\n", fpgaChoice, pcChoice);

xil\_printf("\r\n");

usleep(1000);

}

}

void DemoCleanup() {

DisableCaches();

}

int main() {

DemoInitialize();

DemoRun();

DemoCleanup();

return 0;

}

After initializing the keypad, the program prompts the keyboard player to enter an input and enters into an infinite loop to first wait for the input. The scanf function will read the input string from stdin and place it inside the buffer buf. The input string is then converted into an int and stored inside the variable keyboardInput. A value of 1 corresponds with rock, 2 corresponds with paper, and 3 corresponds with scissors; all other inputs are invalid. The input and corresponding object are printed to stdout. After reading in a valid keyboard input, the program will prompt the keypad player to enter an input and enter into an infinite loop to wait for the input. After a check to ensure that the user only presses a single key at a time using the KYPD\_getKeyPressed function, the keypad input is processed into an int and stored in the keypadChoice variable. Once again, only the inputs 1, 2, and 3 are valid, with the mappings identical to those for the keyboard input. The input and corresponding object are printed to stdout. With both inputs stored as variables, the program will explicitly check the inputs against a set of cases to determine the winner. The result is printed to stdout. The program then sleeps for a short period before the loop repeats to begin another game.

**System Components Description**

Part 1 was a simple application for printing “Hello World” to the terminal via serial connection. The components present in the block diagram are a clocking wizard, processor system reset, MicroBlaze debug module, MicroBlaze IP core, MicroBlaze local memory, AXI interconnect module, and AXI Uartlite module. The clocking wizard and processor system reset modules handled timing and reset functionalities on the FPGA. The MicroBlaze IP core is the processor that will allow C code to be executed on the FPGA, and the local memory gives the IP core access to a small amount of RAM. The AXI interconnect module is used to connect external modules to the FPGA. The AXI Uartlite module will facilitate communication between the Microblaze IP core on the FPGA and the host PC through UART using the RS232 standard.

Part 2a was a multiplication calculator that took in two inputs separated by a slash and printed the output to the terminal, while also lighting up an LED on the FPGA for an output greater than 100. In addition to all of the components from part 1, an AXI GPIO module is present in the block diagram. This module provides an input/output interface to light up the LED on the FPGA.

Part 2b was a two-player rock-paper-scissors game in which one player uses the PC and keyboard input while the other player uses the FPGA and keypad input, with the result of the game once again output to the terminal. The block diagram includes all the components present in part 2a except for the AXI GPIO module. In its place is a PmodKYPD module, which corresponds with an external Pmod keypad component used as the second player’s input device. This module is connected to the FPGA via port ja.

**Challenges Faced**

For part 1, the biggest challenge faced was in figuring out how to set up serial communication given that the tutorial used Tera Term, which was unavailable on the lab PC. The solution to this challenge was to use PuTTY, which provided the same functionality and was familiar to the group from previous experience.

For part 2a, the biggest challenge faced was debugging the C code, thanks to the group’s unfamiliarity with C string and memory manipulation. The bug turned out to be the timeless mistake of forgetting to free dynamically allocated memory, which was leading to undefined behavior across multiple successive executions of the program.

For part 2b, the biggest challenge faced was efficiently implementing the game state machine logic. Due to the circular precedence order between rock, paper, and scissors, simply assigning each outcome to a number and comparing the numbers was insufficient. Ultimately, the group was unable to devise a clean solution to this problem and resorted to hard coding outcomes for both players.

**Workload Distribution**

Alberto configured the LED for part 2a, and wrote the string input processing logic in part 2a. Evan set up the initial Hello World project, and handled the state machine logic in part 2b. Brandon adapted the input processing logic for part 2b, and configured the keypad as the game’s second input source. All group members contributed to debugging and refactoring the code.