Lab 2: Microprocessor System Design

Samuel Huang
ECEN 449

1. Introduction

The Microprocessor System Design lab focuses on reinforcing Vivado as a software based solution for controlling LEDs. This lab, unlike the previous one, utilizes a MicroBlaze processor system using the Vivado Block Design Builder to create functional General Purpose Input/Output that take in Buttons and Switch inputs and display LEDs. This is accomplished with Intellectual Property hardware blocks from Xilinx, which together create a system that can be mapped onto the FPGA hardware. This lab uses C language instead of Verilog source code to determine the behavior of the circuit. Ultimately this lab brings in new software languages and hardware programming techniques, to illustrate a hardware system that utilizes GPIO to demonstrate its behavior.

2. Procedure

The first part of the lab illustrates how to build a MicroBlaze IP Processor. It goes through adding IP blocks and connecting them, to generate a system that handles GPIO. From there it covers how constraint files link buttons, switches, LEDs to specific GPIO ports so that the generated bitstream successfully maps out the MicroBlase IP Processor and its specific GPIO to the FPGA. The lab then teaches how to program the behavior of the board through the utilization of C language.

The first demonstration's C code is a demonstration on how software languages other than Verilog can determine hardware behavior. The demonstration shows that it's not much different, and the main difference between them is the included libraries that allow the software to control the board. The results of the experiment are verifiable both on the board and through print statements in the console. This is an important reminder to include print statements that outline what's going on, because it speeds up debugging and also shows what the intended behavior is.

The second demonstration expands on the simple counter, to include switches and LEDs. With button 0 incrementing, button 1 decrementing, button 2 showing what switches are displayed, and button 4 showing what the count is. This final part of the lab continues to emphasize the importance of synchronization, and introduces the concept of bit masking. This allows for an 8 bit input GPIO to control both Buttons and Switches so that the program can clearly define what the target input is.

3. Results

The lab produced two different implementations of a MicroBlaze Processor being mapped onto a FPGA. The first implementation utilized a delay function that acted as a clock divider, and continually incremented the count after each delay. This created a program on the FPGA that controlled LEDs to display what the current count was. The second implementation used the same delay function, but set certain conditions which controlled a count. Based on different inputs the program determined what was displayed by the LED, whether the current count or what switches were activated or the count being incremented or decremented.

4. Conclusion

The lab overall was not too complicated, the important part to keep in mind was all the steps required to make sure the bitstream was properly generated and programmed onto the FPGA, and the steps required to get the C code to control the behavior of the board. There were a couple hardware issues where, when generating the bitstream the amount of operations had to be reduced otherwise the program would crash. Overall what was learned from all the steps taken to program the FPGA, is that each step matters because they work together to create a fully functioning system. The entire process of the lab also showed how different software languages can be readily used to control hardware systems.

5. Questions

The count value was the same between both implementations, and should have reduced the clock or delay cycle to be at a frequency of 1Hz. However the second implementation's LED update speed was noticeably faster than the last, which is most likely due to C code being a lower level language requiring less processing, and that the Clock Divider in the first implementation and the Delay Function are vastly different in size. The Delay function would take approximately two clock cycles to execute one iteration, one to check if delay_count is less than WAIT_VAL, and the other to increment count.

The count variable is declared as volatile because this declaration C, notifies the compiler that Count can be affected at the hardware level.

The while(1) expression creates an infinite loop, which allows the program to run continuously.

The first implementation felt easier because I was more familiar with it. A pure software implementation is advantageous in the way that allows for functionality with a reduced amount of steps, but disadvantageous in the way that it does not always account for the hardware response and limitations and that it requires specific libraries to connect to the board. The hardware implementation is advantageous in the way that it accounts for the hardware design and limitations and directly programs the board, but is disadvantageous in the way that some functionality is harder to achieve requiring more lines.

.6 Appendix

i) lab2a.c

```
#include <xparameters.h>
#include <xgpio.h>
#include <xstatus.h>
```

```
#include <xil_printf.h>

/*
 * Definitions
 */
#define GPIO_DEVICE_ID XPAR_LED_DEVICE_ID  // GPIO device that LEDs are
connected to
#define WAIT_VAL 1000000

int delay(void);

int main() {
    int count;
    int count_masked;
    XGpio leds;
    int status;

    // Initialize the GPIO driver
    status = XGpio_Initialize(&leds, GPIO_DEVICE_ID);

    // Set the direction for the GPIO to output (all 0 for output)
    XGpio_SetDataDirection(&leds, 1, 0x00);

    if (status != XST_SUCCESS) {
        xil_printf("Initialization failed\n");
    }

    count = 0;

    // Infinite loop to update LEDs
    while (1) {
        // Mask the count value to lower 4 bits
        count_masked = count & 0xF;

        // Write the masked value to the GPIO (LEDs)
        XGpio_DiscreteWrite(&leds, 1, count_masked);

        // Print the current value of LEDs
        xil_printf("Value of LEDs = 0x%x\n\r", count_masked);
        // Delay function
        delay();

        // Increment the counter
        count++;
    }

    return 0;
}

// Simple delay function
int delay(void) {
    // volatile tells  the compiler that delay_count can change at anytime
    volatile int delay_count = 0;

    // Wait loop
    while (delay_count < WAIT_VAL) {
        delay_count++;
    }

    return 0;
}
```

ii) lab2b.c

```
#include <xparameters.h>
#include <xgpio.h>
#include <xstatus.h>
#include <xil_printf.h>

/*
 * Definitions
 */
#define GPIO_DEVICE_ID XPAR_GPIO_0_DEVICE_ID
#define LEDS_DEVICE_ID XPAR_GPIO_1_DEVICE_ID
#define WAIT_VAL 1000000
// Masks to define which button is being pressed
#define PB0_MASK 0x01
#define PB1_MASK 0x02
#define PB2_MASK 0x04
#define PB4_MASK 0x08
```

```
int delay(void);
void display_switch_status(XGpio *gpio);
void display_leds(XGpio *gpio, int count);

int main() {
    int count = 0;
    int status;
    XGpio gpio;
    XGpio leds;
    int push_buttons;

    // Initialization of the GPIO Input (Switches and Buttons)
    status = XGpio_Initialize(&gpio, GPIO_DEVICE_ID);
    if (status != XST_SUCCESS) {
        xil_printf("GPIO Initialization failed\n");
        return XST_FAILURE;
    }
    // Initialization of LED
    status = XGpio_Initialize(&leds, LEDS_DEVICE_ID);
    if (status != XST_SUCCESS) {
        xil_printf("LEDs Initialization failed\n");
        return XST_FAILURE;
    }

    //Sets up GPIO Input
    XGpio_SetDataDirection(&gpio, 1, 0xFF);

    // Sets up Leds
    XGpio_SetDataDirection(&leds, 1, 0x00);

    xil_printf("Program Started. COUNT = %d\n", count);
    // infintite loop to keep the program running
    while (1) {
        // defines what button is pressed
        push_buttons = XGpio_DiscreteRead(&gpio, 1) >> 4;

        if (push_buttons & PB0_MASK) { // if first button pressed

            count++;
            xil_printf("Incrementing COUNT: %d\n", count);
            delay();
        }

        if (push_buttons & PB1_MASK) { // if second button is pressed

            count--;
            xil_printf("Decrementing COUNT: %d\n", count);
            delay();
        }

        if (push_buttons & PB2_MASK) { // if third button is pressed

            display_switch_status(&gpio);
            delay();
        }

        if (push_buttons & PB4_MASK) { // if fourth button is pressed

            display_leds(&leds, count);
            xil_printf("Displaying COUNT on LEDs: 0x%x\n", count);
            delay();
        }
    }

    return 0;
}


int delay(void) { // sets up delay to simulate proper synchronization
    volatile int delay_count = 0;
    while (delay_count < WAIT_VAL) {
        delay_count++;
    }
    return 0;
}


void display_switch_status(XGpio *gpio) { // function behind displaying switch status
    int switches = XGpio_DiscreteRead(gpio, 1) & 0x0F;
    xil_printf("Switch status: 0x%x\n", switches);
}


void display_leds(XGpio *gpio, int count) { // function that displays LED based on count
    int count_masked = count & 0xF;
    XGpio_DiscreteWrite(gpio, 1, count_masked);
}
```

iii) lab2a.xcv

```
#clock_rtl
set_property PACKAGE_PIN K17 [get_ports clk_100MHz]
set_property IOSTANDARD LVCMOS33 [get_ports clk_100MHz]
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports clk_100MHz]

#led_tri_o
set_property PACKAGE_PIN M14 [get_ports gpio_rtl_0_tri_o[0]]
set_property IOSTANDARD LVCMOS33 [get_ports gpio_rtl_0_tri_o[0]]

set_property PACKAGE_PIN M15 [get_ports gpio_rtl_0_tri_o[1]]
set_property IOSTANDARD LVCMOS33 [get_ports gpio_rtl_0_tri_o[1]]

set_property PACKAGE_PIN G14 [get_ports gpio_rtl_0_tri_o[2]]
set_property IOSTANDARD LVCMOS33 [get_ports gpio_rtl_0_tri_o[2]]

set_property PACKAGE_PIN D18 [get_ports gpio_rtl_0_tri_o[3]]
set_property IOSTANDARD LVCMOS33 [get_ports gpio_rtl_0_tri_o[3]]
```

iv) lab2b.xcv

```
#clock_rtl
set_property PACKAGE_PIN K17 [get_ports clk_100MHz]
set_property IOSTANDARD LVCMOS33 [get_ports clk_100MHz]
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports clk_100MHz]

#led_tri_o
set_property PACKAGE_PIN M14 [get_ports gpio_rtl_0_tri_o[0]]
set_property IOSTANDARD LVCMOS33 [get_ports gpio_rtl_0_tri_o[0]]

set_property PACKAGE_PIN M15 [get_ports gpio_rtl_0_tri_o[1]]
set_property IOSTANDARD LVCMOS33 [get_ports gpio_rtl_0_tri_o[1]]

set_property PACKAGE_PIN G14 [get_ports gpio_rtl_0_tri_o[2]]
set_property IOSTANDARD LVCMOS33 [get_ports gpio_rtl_0_tri_o[2]]

set_property PACKAGE_PIN D18 [get_ports gpio_rtl_0_tri_o[3]]
set_property IOSTANDARD LVCMOS33 [get_ports gpio_rtl_0_tri_o[3]]
```

```
#button
set_property PACKAGE_PIN K18 [get_ports gpio_rtl_1_tri_i[4]]
set_property IOSTANDARD LVCMOS33 [get_ports gpio_rtl_1_tri_i[4]]

set_property PACKAGE_PIN P16 [get_ports gpio_rtl_1_tri_i[5]]
set_property IOSTANDARD LVCMOS33 [get_ports gpio_rtl_1_tri_i[5]]

set_property PACKAGE_PIN K19 [get_ports gpio_rtl_1_tri_i[6]]
set_property IOSTANDARD LVCMOS33 [get_ports gpio_rtl_1_tri_i[6]]

set_property PACKAGE_PIN Y16 [get_ports gpio_rtl_1_tri_i[7]]
set_property IOSTANDARD LVCMOS33 [get_ports gpio_rtl_1_tri_i[7]]
#switch
set_property PACKAGE_PIN G15 [get_ports gpio_rtl_1_tri_i[0]]
set_property IOSTANDARD LVCMOS33 [get_ports gpio_rtl_1_tri_i[0]]

set_property PACKAGE_PIN P15 [get_ports gpio_rtl_1_tri_i[1]]
set_property IOSTANDARD LVCMOS33 [get_ports gpio_rtl_1_tri_i[1]]

set_property PACKAGE_PIN W13 [get_ports gpio_rtl_1_tri_i[2]]
set_property IOSTANDARD LVCMOS33 [get_ports gpio_rtl_1_tri_i[2]]

set_property PACKAGE_PIN T16 [get_ports gpio_rtl_1_tri_i[3]]
set_property IOSTANDARD LVCMOS33 [get_ports gpio_rtl_1_tri_i[3]]
```