ECEN 749

Laboratory Exercise #2

Using the Software Development Kit (SDK)

Name: Pranav Anantharam

UIN: 734003886

Course: ECEN 749 Microprocessor System Design

Section: 603

Introduction:

The second laboratory exercise aims to introduce us to Vivado by transitioning from a pure FPGA hardware solution to a software-based one for controlling LEDs. Through this process, we learn to develop a MicroBlaze processor system using the Vivado Block Design Builder. By incorporating GPIO capabilities using Xilinx Intellectual Property hardware blocks, we enhance the microprocessor's functionality. Ultimately, we write software in C to run on the MicroBlaze processor, enabling the desired LED functionality. This lab is crucial as it demonstrates the integration of hardware and software components, a fundamental aspect of FPGA development, and prepares us for more complex designs involving both hardware and software interaction.

Procedure:

1) Create a working directory in the home folder.
2) Load the environment variables on the workstation using appropriate commands and open Vivado 2022.1.
3) Create a new project and select the 'Create Block Design' option.
4) Right click within the diagram tab and add the 'MicroBlaze' IP to our design. Setup the configuration for the MicroBlaze processor as given in the lab manual.
5) Customize the Clocking Wizard to set the primary input clock source to 'Single ended clock capable pin'.
6) Next, run connection automation to setup the processor.
7) Add General Purpose IO (GPIO) blocks to interact with LEDs, Switches and Buttons on the Zybo Z7-10 board.
8) Once again, run connection automation and select the 'Regenerate Layout' option to clean up the circuit.
9) Rename the GPIO block to 'led'.
10) Add a constant IP block named 'VDD' and connect it to 'ext_rst_in' of the Processor System Reset to configure the reset port to a push button. Regenerate the layout.
11) Map the IO ports to the LEDs and buttons by defining and adding a constraints file.
12) Select 'Validate Design' to check for errors in the design. Create HDL wrapper and generate bitstream.
13) Once the bitstream has been generated, export the hardware design (.xsa file).
14) Launch Vitis IDE and create a new 'Application Project'.
15) In the Platform page, select the 'Create a new hardware platform' option and select the exported hardware design file.
16) Give an application project name and select the 'Empty Application (C)' template.
17) Using a text editor, write the code for a counter using the onboard LEDs in C. Import the C file into the project in Vitis IDE.
18) In Vitis IDE, build the hardware platform project and software system project. The application binary file will be created.
19) Finally, Program the FPGA board by selecting the 'Program Device' option.
20) Attach the Vitis console to the hardware output to see the output of the printf statements on the console in Vitis IDE.
21) To perform the second component of the lab, add an 8-bit GPIO block to hardware diagram using the 'Add IP' option and configure all the lines to be inputs.
22) Update constraints file to map the lower 4 bits of the GPIO inputs to the on-board DIP switches and upper 4 bits of the GPIO inputs to the push buttons on the Zybo Z7-10 board.
23) Validate the design and create a HDL wrapper. Generate bitstream and export the hardware design.
24) Load the new hardware design into Vitis IDE and build the project for the hardware design.
25) Write a C program to keep track of a count value and perform operations as mentioned in the lab manual.
26) Import the C file into Vitis IDE and build the project for the software system.

27) Program the FPGA board with the new binary file and attach the Vitis console to the hardware output to view the printf outputs on the console.

(Block design, Source Code and Constraints file changes given in Appendix Section)

Results:

First Section:

A simple counter software project was created and developed (following the steps given in the lab manual) to light up the LEDs on the Zybo Z7-10 board based on the counter value. The output of the LEDs matched the behavior of the programmed up-counter.

(Screenshots of output provided in Appendix Section)

Second Section:

The second section of the laboratory exercise involved keeping track of counter value and DIP switch state making use of the push buttons on the Zybo Z7-10 board. Changes were required in the constraints file to map the new 8 bit GPIO input block, followed by generating a new bitstream and exporting the updated hardware (.xsa file). A C program was written to read the GPIO input block and determine which push button was pressed and perform the appropriate operation – increment count, decrement count, display count value on LEDs, display state of switches on LEDs. The 1 delay of second was generated between each operation to make sure increment and decrement operations occurred at approximately 1 Hz frequency. The entire logic was wrapped in a 'while(1)' infinite loop.

The outputs of the program were demonstrated using the on-board LEDs of the Zybo Z7-10 board and the logs were printed onto the console.

(Source Code, Constraints file changes and Output Screenshots given in Appendix Section)

Conclusion:

In conclusion, the second laboratory exercise serves as a vital introduction to Vivado software and its integration with FPGA hardware, facilitating the transition from hardware-centric to software-controlled LED management. Through the development of a MicroBlaze processor system, incorporation of GPIO capabilities, and programming in C, we explore the essential combination of hardware and software components. This integration not only emphasizes a fundamental aspect of FPGA development but also equips us with the foundational skills necessary for tackling more intricate projects requiring seamless hardware-software interaction. Thus, this lab lays a solid groundwork, enabling us to navigate the complexities of FPGA design.

Questions:

(a) In the first part of the lab, we created a delay function by implementing a counter. The goal was to update the LEDs approximately every second as we did in the previous lab. Compare the count value in this lab to the count value you used as a delay in the previous lab. If they are different, explain why? Can you determine approximately how many clock cycles are required to execute one iteration of the delay for-loop? If so, how many?

Answer:

In Lab 1, we utilized a clock divider reaching a count of 125 million, aligned with the 125MHz clock cycle frequency. This large divider was necessary to achieve a 1Hz clock rate by dividing the primary clock. In Lab 2, however, the processor operates on a 100 MHz clock. The delay is executed via software, requiring more instructions, approximately 10 cycles to execute when translated into assembly code. With the count incremented at a rate of 1Hz (equivalent to 100 million clock cycles at 100 MHz), we aimed for a divider value of around 10000000, matching the one utilized in our C code.

(b) Why is the count variable in our software delay declared as volatile?

Answer:

The "volatile" keyword in C is a qualifier used when declaring a variable. It informs the compiler that the value of the variable may change unexpectedly, without any corresponding code nearby. In this case, the delay function is intended to introduce a delay by looping until a certain count value is reached. By declaring delay_count as a volatile we can ensure that the compiler does not optimize the loop and modify the behavior, thereby ensuring that the delay is achieved as intended.

(c) What does the while(1) expression in our code do?

Answer:

The while (1) loop in the expression acts as an infinite loop, running the code infinitely and checking for user inputs, writing to GPIO pins and printing outputs to console. This way the inputs from the board are detected correctly and the program keeps running. This is similar to the always block used in Verilog code.

(d) Compare and contrast this lab with the previous lab. Which implementation do you feel is easier? What are the advantages and disadvantages associated with a purely software implementation such as this when compared to a purely hardware implementation such as the previous lab?

Answer:

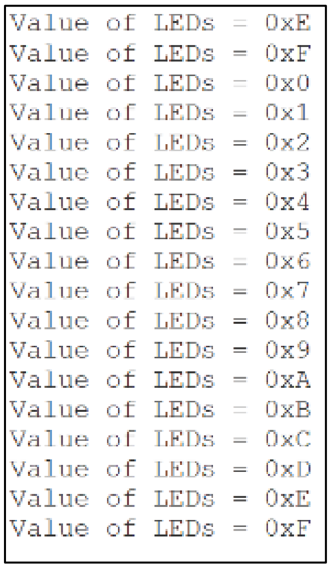
Both labs involved programming Switches and LEDs, but the distinction between them was that the Lab 1 focused on purely hardware implementation, while Lab 2 used purely software implementation.

The advantages of using a software-based approach include easier debugging of any errors, as compared to hardware implementation. Additionally, it is easier to test the code, as there will be outputs displayed in the TCL console (with the help of print statements). This type of visibility is not available in hardware implementation. However, a purely hardware implementation such as in Lab 1 required much lesser logic to implement and was easier to setup

Appendix:

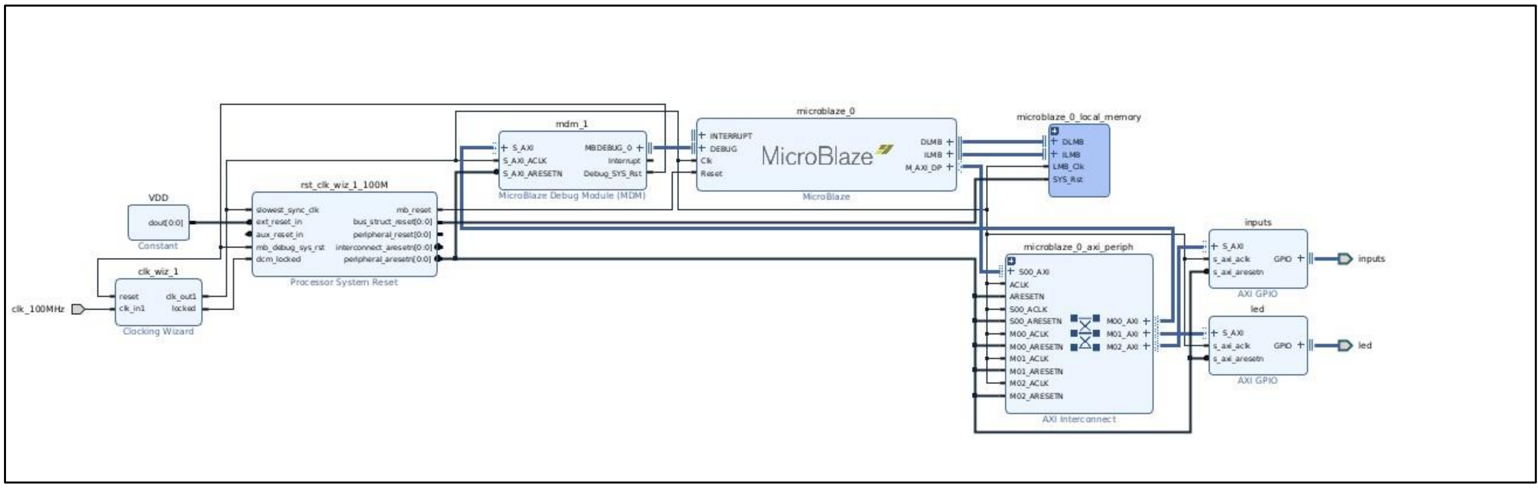
(Screenshots of Block design, Source Code, Constraints files and Output Screenshots)

Output for first section:

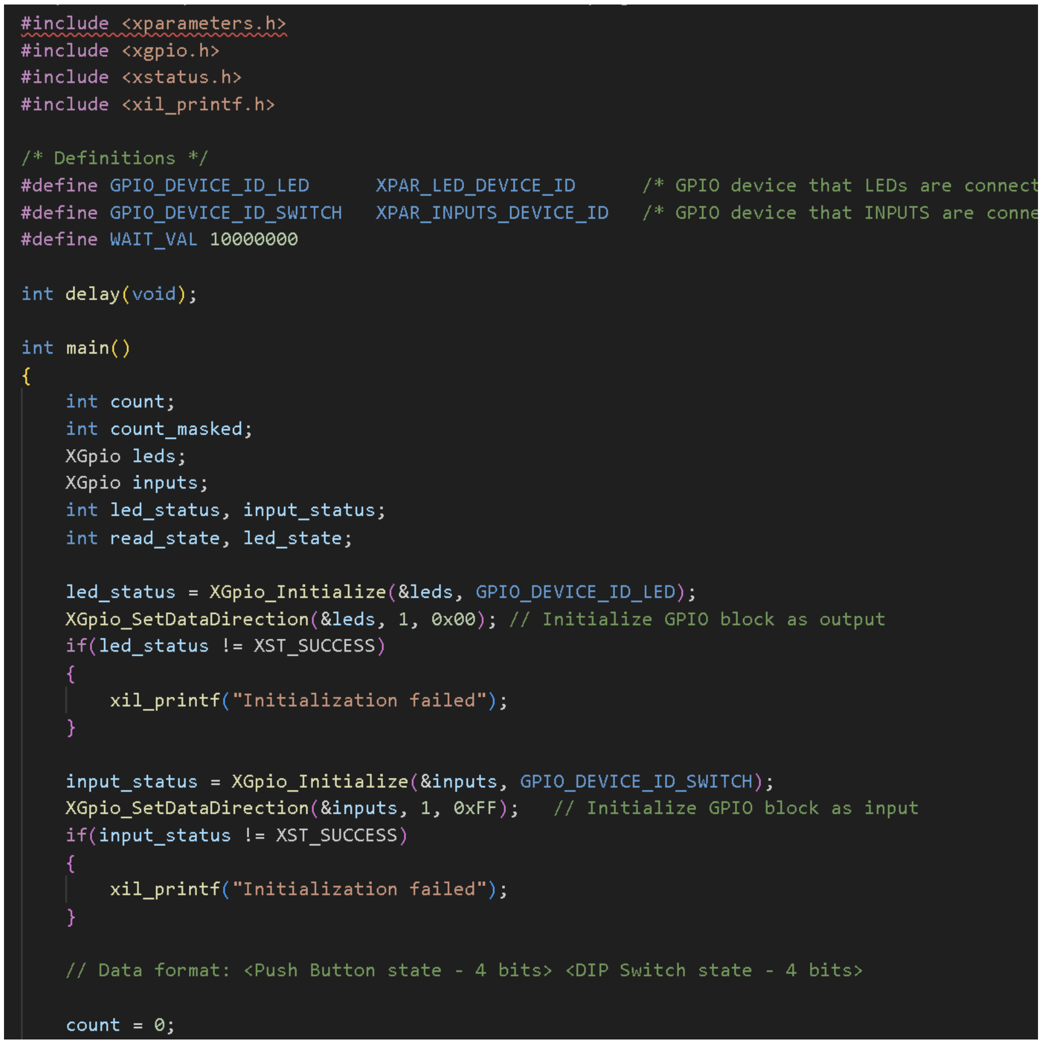


```
Value of LEDs = 0xE
Value of LEDs = 0xF
Value of LEDs = 0x0
Value of LEDs = 0x1
Value of LEDs = 0x2
Value of LEDs = 0x3
Value of LEDs = 0x4
Value of LEDs = 0x5
Value of LEDs = 0x6
Value of LEDs = 0x7
Value of LEDs = 0x8
Value of LEDs = 0x9
Value of LEDs = 0xA
Value of LEDs = 0xB
Value of LEDs = 0xC
Value of LEDs = 0xD
Value of LEDs = 0xE
Value of LEDs = 0xF
```

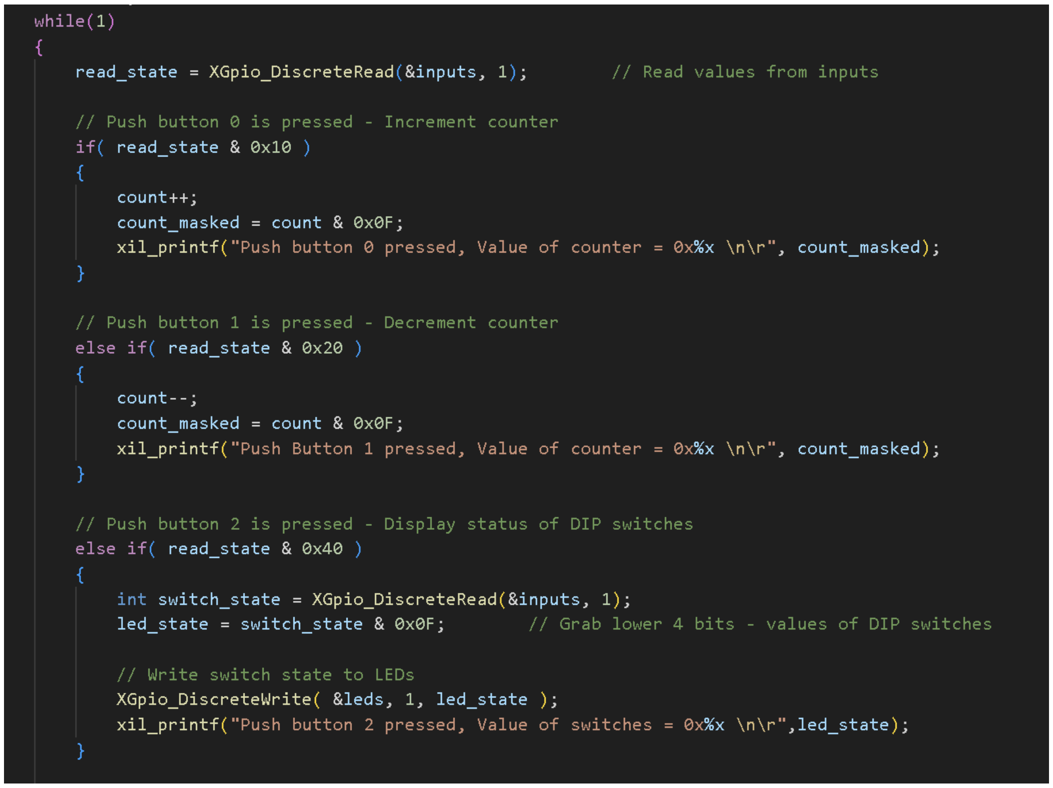
Block Design for second section:



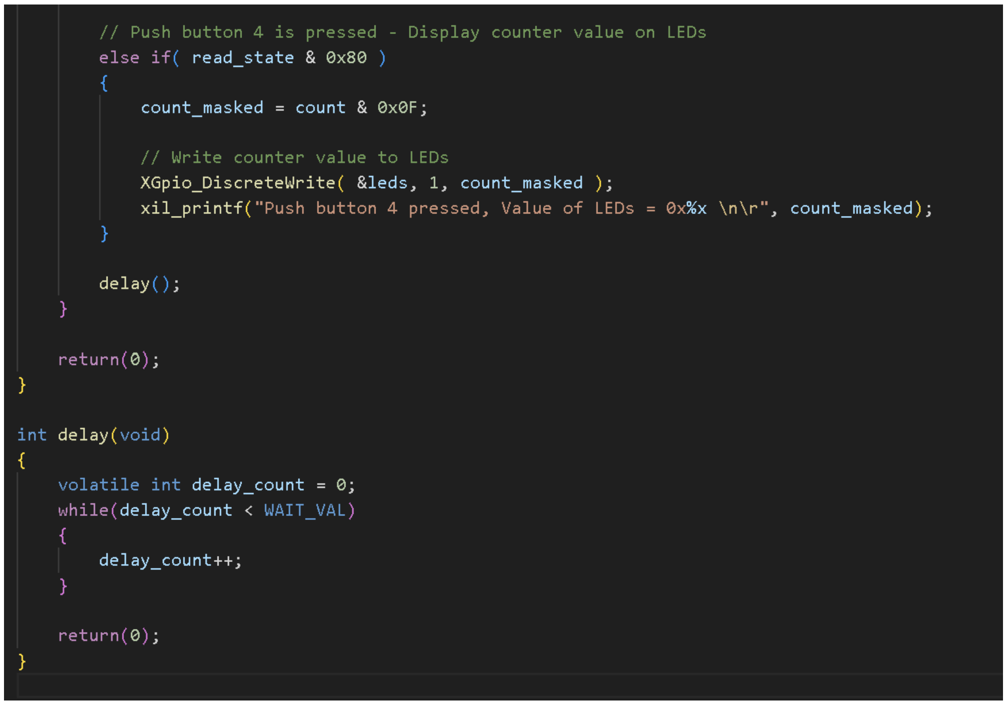
Source Code for second section:



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

/* Definitions */
#define GPIO_DEVICE_ID_LED      XPAR_LED_DEVICE_ID      /* GPIO device that LEDs are connect
#define GPIO_DEVICE_ID_SWITCH   XPAR_INPUTS_DEVICE_ID   /* GPIO device that INPUTS are conne
#define WAIT_VAL 10000000

int delay(void);

int main()
{
    int count;
    int count_masked;
    XGpio leds;
    XGpio inputs;
    int led_status, input_status;
    int read_state, led_state;

    led_status = XGpio_Initialize(&leds, GPIO_DEVICE_ID_LED);
    XGpio_SetDataDirection(&leds, 1, 0x00); // Initialize GPIO block as output
    if(led_status != XST_SUCCESS)
    {
        xil_printf("Initialization failed");
    }

    input_status = XGpio_Initialize(&inputs, GPIO_DEVICE_ID_SWITCH);
    XGpio_SetDataDirection(&inputs, 1, 0xFF);   // Initialize GPIO block as input
    if(input_status != XST_SUCCESS)
    {
        xil_printf("Initialization failed");
    }

    // Data format: <Push Button state - 4 bits> <DIP Switch state - 4 bits>

    count = 0;
```

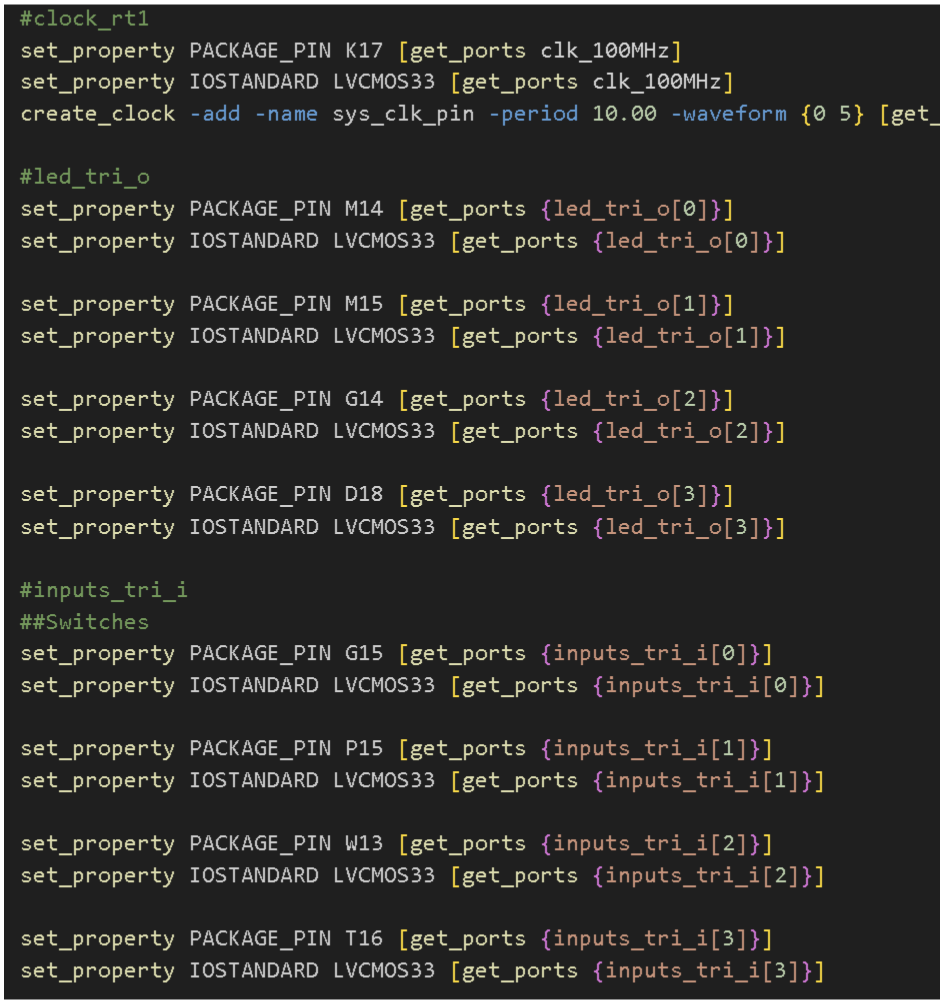


```c
while(1)
{
    read_state = XGpio_DiscreteRead(&inputs, 1);        // Read values from inputs

    // Push button 0 is pressed - Increment counter
    if( read_state & 0x10 )
    {
        count++;
        count_masked = count & 0x0F;
        xil_printf("Push button 0 pressed, Value of counter = 0x%x \n\r", count_masked);
    }

    // Push button 1 is pressed - Decrement counter
    else if( read_state & 0x20 )
    {
        count--;
        count_masked = count & 0x0F;
        xil_printf("Push Button 1 pressed, Value of counter = 0x%x \n\r", count_masked);
    }

    // Push button 2 is pressed - Display status of DIP switches
    else if( read_state & 0x40 )
    {
        int switch_state = XGpio_DiscreteRead(&inputs, 1);
        led_state = switch_state & 0x0F;        // Grab lower 4 bits - values of DIP switches

        // Write switch state to LEDs
        XGpio_DiscreteWrite( &leds, 1, led_state );
        xil_printf("Push button 2 pressed, Value of switches = 0x%x \n\r",led_state);
    }
```

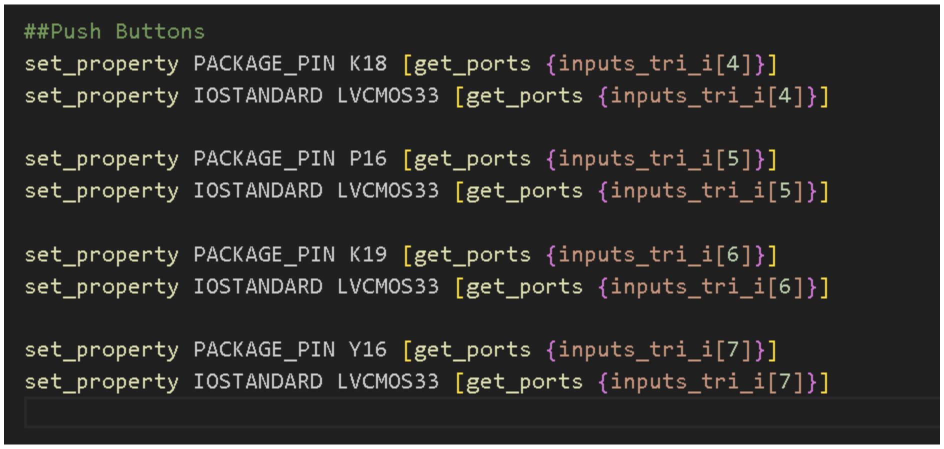


```c
        // Push button 4 is pressed - Display counter value on LEDs
        else if( read_state & 0x80 )
        {
            count_masked = count & 0x0F;

            // Write counter value to LEDs
            XGpio_DiscreteWrite( &leds, 1, count_masked );
            xil_printf("Push button 4 pressed, Value of LEDs = 0x%x \n\r", count_masked);
        }

        delay();
    }

    return(0);
}

int delay(void)
{
    volatile int delay_count = 0;
    while(delay_count < WAIT_VAL)
    {
        delay_count++;
    }

    return(0);
}
```

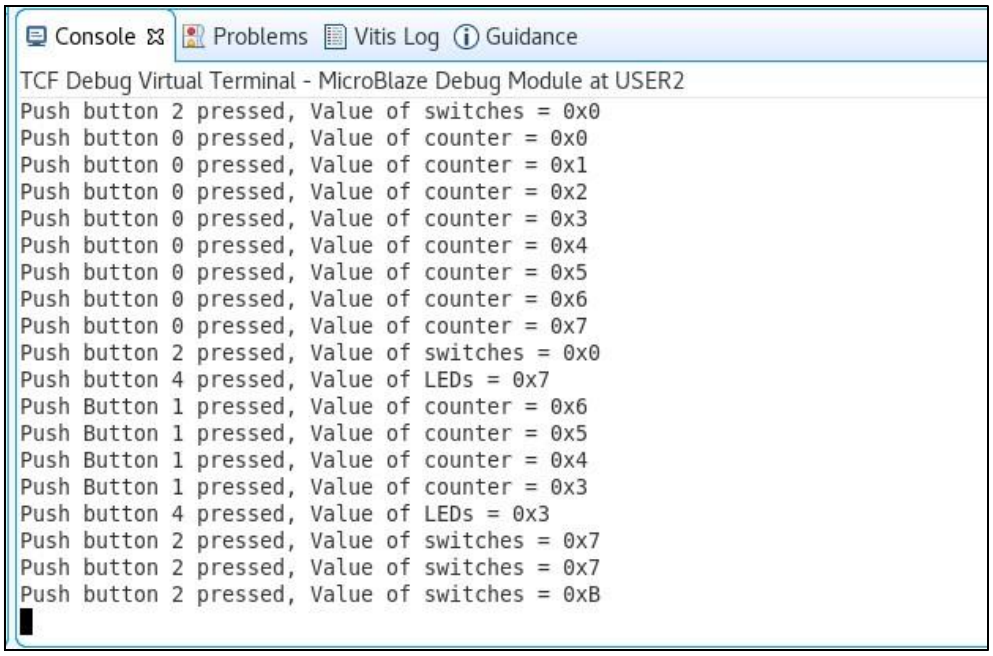
Constraints File for second section:



```
#clock_rt1
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_

#led_tri_o
set_property PACKAGE_PIN M14 [get_ports {led_tri_o[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led_tri_o[0]}]

set_property PACKAGE_PIN M15 [get_ports {led_tri_o[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led_tri_o[1]}]

set_property PACKAGE_PIN G14 [get_ports {led_tri_o[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led_tri_o[2]}]

set_property PACKAGE_PIN D18 [get_ports {led_tri_o[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led_tri_o[3]}]

#inputs_tri_i
##Switches
set_property PACKAGE_PIN G15 [get_ports {inputs_tri_i[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {inputs_tri_i[0]}]

set_property PACKAGE_PIN P15 [get_ports {inputs_tri_i[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {inputs_tri_i[1]}]

set_property PACKAGE_PIN W13 [get_ports {inputs_tri_i[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {inputs_tri_i[2]}]

set_property PACKAGE_PIN T16 [get_ports {inputs_tri_i[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {inputs_tri_i[3]}]
```


```
##Push Buttons
set_property PACKAGE_PIN K18 [get_ports {inputs_tri_i[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {inputs_tri_i[4]}]

set_property PACKAGE_PIN P16 [get_ports {inputs_tri_i[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {inputs_tri_i[5]}]

set_property PACKAGE_PIN K19 [get_ports {inputs_tri_i[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {inputs_tri_i[6]}]

set_property PACKAGE_PIN Y16 [get_ports {inputs_tri_i[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {inputs_tri_i[7]}]
```

Output for second section:



```
TCF Debug Virtual Terminal - MicroBlaze Debug Module at USER2
Push button 2 pressed, Value of switches = 0x0
Push button 0 pressed, Value of counter = 0x0
Push button 0 pressed, Value of counter = 0x1
Push button 0 pressed, Value of counter = 0x2
Push button 0 pressed, Value of counter = 0x3
Push button 0 pressed, Value of counter = 0x4
Push button 0 pressed, Value of counter = 0x5
Push button 0 pressed, Value of counter = 0x6
Push button 0 pressed, Value of counter = 0x7
Push button 2 pressed, Value of switches = 0x0
Push button 4 pressed, Value of LEDs = 0x7
Push Button 1 pressed, Value of counter = 0x6
Push Button 1 pressed, Value of counter = 0x5
Push Button 1 pressed, Value of counter = 0x4
Push Button 1 pressed, Value of counter = 0x3
Push button 4 pressed, Value of LEDs = 0x3
Push button 2 pressed, Value of switches = 0x7
Push button 2 pressed, Value of switches = 0x7
Push button 2 pressed, Value of switches = 0xB
```