Lab 01 Report: Vivado SDK Basic IP Integrator

Peter Mowen
peter.mowen@temple.edu

Summary
Lab 01 introduces the Vivado HLx, Xilinx SDK, and the Zybo Board. The Vivado HLx software is used to generate the code to program the FPGA. The FPGA interfaces with the switches and LEDs. SDK is used to program the microprocessor on the Zybo Board. The processor communicates to the hardware through the FPGA using AXI. A finite state machine is used to model the control logic and is implemented in the C code.

Introduction
The objective for Lab 01 is to become familiar with the Vivado development environment (both Vivado HLx and SDK) and the basic architecture of the Zybo Board. The Zybo Board is a system on a chip. The main chip has a dual-core ARM Cortex A9 microprocessor and a Xilinx FPGA on it. The processor communicates to the other hardware on the board through the FPGA using the AXI protocol. In this lab, the switches and LEDs are used.

To achieve the main objective, we are asked to create the hardware block diagram using Vivado HLx, export the hardware design to the Xilinx SDK, and write code based on the following design requirements taken from the lab manual (Silage, Spring 2020):

1. The rightmost slide switch 0 (SW0) is the system RESET and when ON the LED count and display is set at 0 (0000) and when OFF the LED process set by the other slide switches is enabled. SW0 as a RESET overrides all other slide switch operations.
2. If slide switch 1 (SW1) is ON and no other slide switches are ON, all further LED count and operations are suspended (LED count is fixed). When slide switch 1 is OFF the LED operations and display continue.
3. If slide switch 2 (SW2) is ON and no other slide switches are ON the LED count and display is set to 0 (0000) and proceeds as an increasing and decreasing, repeating bar graph with a delay (that is, 0000, 0001, 0011, 0111, 1111, 0111, 0011, 0001, 0000, 0001, 0011…).
4. If slide switch 3 (SW3) is ON and no other slide switches are ON the LED count and display is set to 1001 and then is a repeating pattern with a delay (that is, 1001, 0110, 1010, 0101, 1100, 0011, 1001, 0110, …).

A finite state machine was designed to meet the design specification. The FSM moves between states based on the switch configuration. The output of the FSM is the LED count.

Discussion

<u>Hardware Setup in Vivado HLx</u>
The following block diagram was made in Vivado HLx based on the designs in exercises 1 and 2 from *The Zynq Book Tutorials* (Louise H. Crocket, 2015):

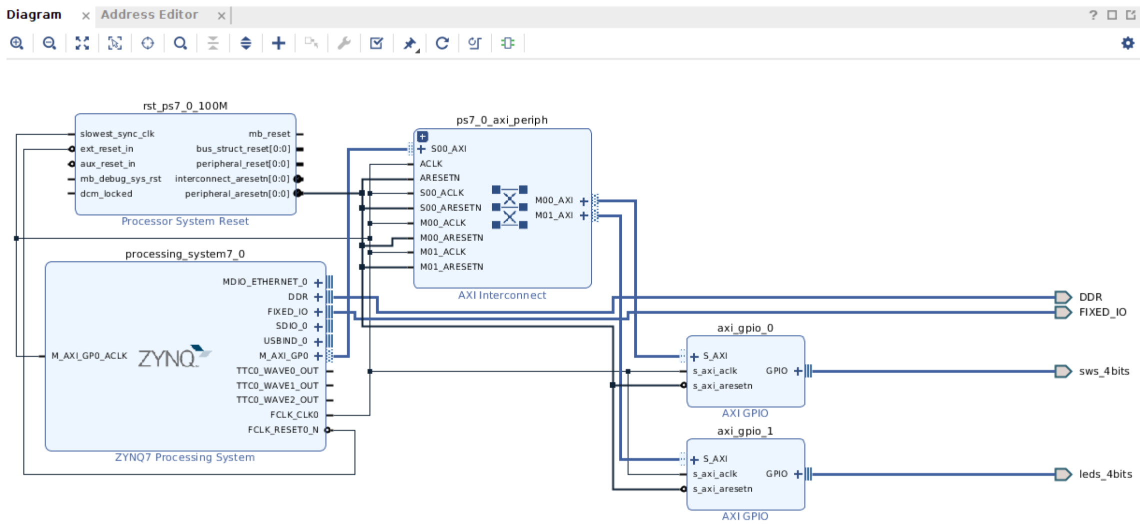

Figure 1: Block Diagram created in Vivado HLx

The main processor block is connected to the AXI peripheral block, which is in turn connected to two AXI GPIO blocks. GPIO 0 is connected to the four switches. GPIO 1 is connected to the four LEDs. Each block is known as an Intellectual Property (IP) block. These blocks were supplied by the Vivado HLx software. Each block was connected using the automated connection tools within Vivado HLx. After finishing this design, a bit stream was created, and the hardware was exported to the SDK. This concludes the work needed to write the software for the FPGA.

Hardware Setup in SDK

Once in the SDK, a new blank application was created and Exercise 1 from *The Zynq Book Tutorials* was imported. This exercise provided much of the structure needed to interface with the hardware. The following code snippet shows the required include files and the preprocessor defines that allow the software to talk to the hardware:

```c
// Include Files
#include "xparameters.h"
#include "xgpio.h"
#include "xstatus.h"
#include "xil_printf.h"

// Definitions for Xilinx Hardware
#define SW_DEVICE_ID   XPAR_AXI_GPIO_0_DEVICE_ID     // GPIO device that Switches are connected to
#define LED_DEVICE_ID  XPAR_AXI_GPIO_1_DEVICE_ID     // GPIO device that LEDs are connected to
#define LED_DELAY 25000000                           // Software delay length
#define SW_CHANNEL 1                                 // GPIO port for Switches
#define LED_CHANNEL 1                                // GPIO port for LEDs
#define printf xil_printf                            // smaller, optimized printf

// GPIO
XGpio SWInst, LEDInst;                               // GPIO Device driver instance

```

Figure 2: Include files and Preprocessor defines for hardware

The include files *xparameters.h*, *xgpio.h*, and *xstatus.h* bring in all the code we need to be able to interact with the Xilinx hardware. The *xil_printf.h* file brings in an optimized printf function. The first two defines map a short descriptive name to the longer, more cryptic names of the GPIO device IDs. The next define maps a number to be used in the LED delay to a descriptive name. The next two defines map the GPIO channel numbers to descriptive names. These will be used later in the code when initializing the GPIOs. Finally, the GPIOs for the switches and LEDs are instantiated.

The following is a screenshot showing the code to initialize the GPIOs:

```
    // Initialize Switch GPIO and check status
    Status = XGpio_Initialize(&SWInst, SW_DEVICE_ID);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    // Set the direction for the switches to input
    XGpio_SetDataDirection(&SWInst, SW_CHANNEL, 0xF);

    // Initialize LED GPIO and check status
    Status = XGpio_Initialize(&LEDInst, LED_DEVICE_ID);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    // Set the direction for the LEDs to output
    XGpio_SetDataDirection(&LEDInst, LED_CHANNEL, 0x0);

```

Figure 3: C code to initialize hardware

The *XGpio_Initialize* function is used to setup the hardware. It takes two arguments. The first is the handle to the instance of the GPIO. The second is the GPIO's device ID. The result of the function call is stored in the status variable. If the setup was not a success, then the code returns a failure to the main function.

Once the GPIO is initialized, the data direction is set using the *XGpio_SetDataDirection* function. This function takes three inputs. The first is the handle to the GPIO instance. The second is the channel one would like to set. The third is the data direction. To set a GPIO as an input, one writes all ones (0xF in hexadecimal) to the data direction register. To set a GPIO as an output, one writes all zeros (0x0 in hexadecimal) to the data direction register.

Control Logic Design

The following finite state machine was designed to control the LEDs based on the switches:

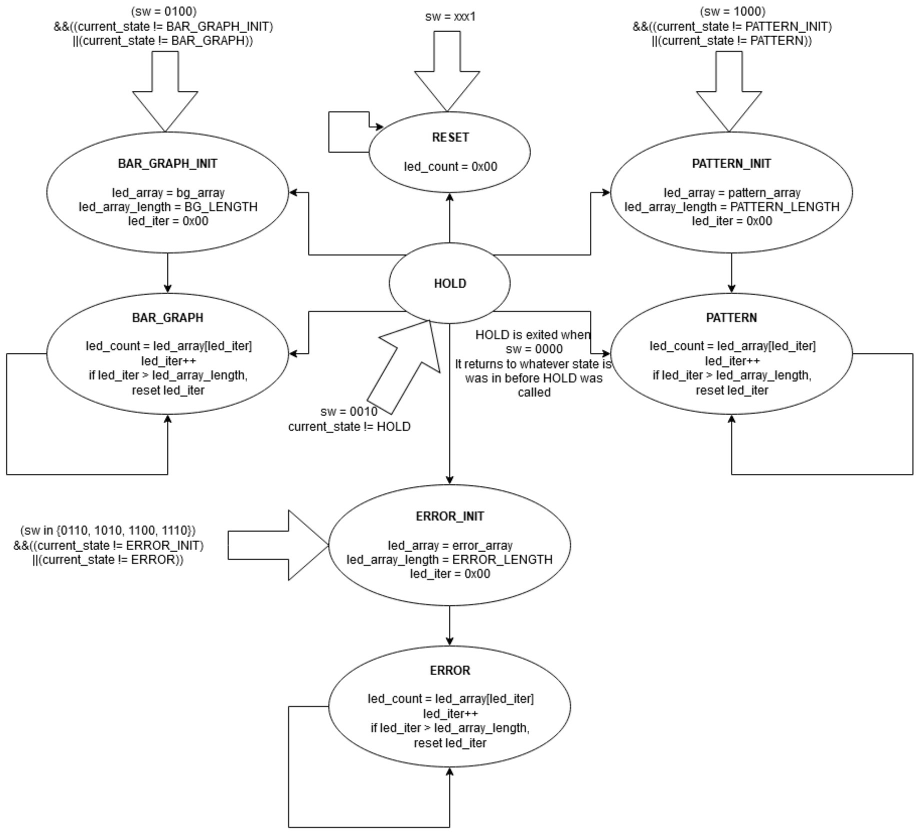

Figure 4: Finite State Machine for control logic

The following code snippet shows the setup for the FSM code.

```c
// Variables to hold LED state and switch state
static char led_count;                          // Number displayed on LEDs
static char sw;                                 // Switch state

// States for finite state machine
enum state {RESET, HOLD, BAR_GRAPH_INIT, BAR_GRAPH, PATTERN_INIT, PATTERN, ERROR_INIT, ERROR};

// State variables for flow control
enum state current_state;
enum state next_state;
enum state saved_state;

// Place holders for LED array, length of led array, LED array iterator
char *led_array;
unsigned char led_iter = 0x00;
int led_array_length = 0;

// Define variables related to bar graph
char bg_array[] = {0x00, 0x01, 0x03, 0x07, 0x0F, 0x07, 0x03, 0x01};
#define BG_LENGTH sizeof(bg_array)/sizeof(bg_array[0])

// Define variables related to random pattern state
char pattern_array[] = {0x09, 0x06, 0x0A, 0x05, 0x0C, 0x03};
#define PATTERN_LENGTH sizeof(pattern_array)/sizeof(pattern_array[0])

// Define variables related to random pattern state
char error_array[] = {0x09, 0x03};
#define ERROR_LENGTH sizeof(error_array)/sizeof(error_array[0])
```

Figure 5: FSM variable setup

The led_count variable is the output to be displayed on the LEDs and sw is the variable to hold the switch input. The enum type was used to enumerate each possible state for the FSM. This makes it easy to compare and assign states. Next, the state variables current_state, next_state, and saved_state were created. These variables control the flow of the FSM. The next set of variables are used in the FSM to display the various patterns on the LEDs.

The following code snippet shows the main logic for the finite state machine:

```c
    // Loop forever updating the LEDs based on the switch input
    while (1)
    {
        // Update state based on last pass through while loop
        current_state = next_state;

        //---------------------------FINITE STATE MACHINE---------------------------
        switch(current_state)
        {
            case RESET: // turn-off LEDs
                led_count = 0x00;
                next_state = RESET;
                break;
            case HOLD: // hold LEDs
                led_count = led_count;
                next_state = HOLD;
                break;
            case BAR_GRAPH_INIT: // initialize the bar graph pattern
                led_array = bg_array;
                led_array_length = BG_LENGTH;
                led_iter = 0x00;
                next_state = BAR_GRAPH;
                break;
            case PATTERN_INIT: // initialize the random pattern
                led_array = pattern_array;
                led_array_length = PATTERN_LENGTH;
                led_iter = 0x00;
                next_state = PATTERN;
                break;
            case ERROR_INIT: // initialize the error pattern
                led_array = error_array;
                led_array_length = ERROR_LENGTH;
                led_iter = 0x00;
                next_state = ERROR;
                break;
            default: // Run LED pattern based on current_state (BAR_GRAPH, PATTERN, ERROR)
                led_count = led_array[led_iter]; // update LED state
                led_iter++; // increment iterator
                if (led_iter > led_array_length - 1) led_iter = 0x00; // reset iterator
                next_state = current_state;
                break;
        }

```

Figure 6: Finite State Machine main logic

The BAR_GRAPH_INIT, PATTERN_INIT, and ERROR_INIT states assign the appropriate array to be cycled through, the length of the array, and initialize the LED iterator. The BAR_GRAPH, PATTERN, and ERROR states all run a repeating pattern on the LEDs. The pattern is created by cycling through the elements in the array and assigning each to LED count.

The following code snippet shows the priority logic that updates the state based on the switches.

```
        //--------------------------PRIORITY LOGIC--------------------------

        // Read switches
        sw = XGpio_DiscreteRead(&SWInst, 1) & 0x0F;

        // Check switches to update next_state if necessary
        if (sw % 2 == 1)
        { // then the lest significant bit is a 1
            next_state = RESET;
        }
        else if (sw == 0x02)
        { // enter the HOLD state if we're not already in it
            if (current_state != HOLD)
            { // save the current_state and set the next state to HOLD
                saved_state = current_state;
                next_state = HOLD;
            }
        }
        else if (sw == 0x04)
        { // Start running the bar graph LED display if not running it already
            if ((current_state < BAR_GRAPH_INIT)||(current_state > BAR_GRAPH))
            {
                next_state = BAR_GRAPH_INIT;
            }
        }
        else if (sw == 0x08)
        { // Start running the random LED pattern if not running it already
            if ((current_state < PATTERN_INIT)||(current_state > PATTERN))
            {
                next_state = PATTERN_INIT;
            }
        }
        else if (sw == 0x00)
        { // Keep doing what you're doing unless coming out of the HOLD state
            if (current_state == HOLD) next_state = saved_state;
        }
        else
        { // all of the error cases are caught here
            if (current_state < ERROR_INIT)
            {
                next_state = ERROR_INIT;
            }
        }

```

Figure 7: Priority Logic for FSM

The *XGpio_DiscreteRead* function takes two inputs. The first is the handle for the GPIO and the second is the channel number. The output of this function is ANDed with 0x0F to mask the upper bits and let the lower four bits pass. Since the reset switch is the least significant bit, it is only set if the number is odd.

The LEDs were updated using the *XGpio_DiscreteWrite* function and a for loop was used to run the delay. It is similar to the read function in that it takes two arguments, the handle for the GPIO and the channel.

```
        // Write output to the LEDs
        XGpio_DiscreteWrite(&LEDInst, LED_CHANNEL, led_count);

        // Wait so user can read LEDs.
        for (Delay = 0; Delay < LED_DELAY; Delay++);
    }

    return XST_SUCCESS; // Should be unreachable
}
```

Figure 8: Write to LEDs and wait

Finally, the main function calls the DecodeSwitchesAndUpdateLEDs. This starts the finite state machine. Assuming everything runs correctly in that function, the code will never go past line 179.

```
173 // Main function
174 int main(void){
175
176     int Status;
177
178     // Start running LED pattern based on switch state
179     Status = DecodeSwitchesAndUpdateLEDs();
180     if (Status != XST_SUCCESS) {
181         xil_printf("GPIO output to the LEDs failed!\r\n");
182     }
183
184     return 0;
185 }
186
```

Figure 9: main function

A demonstration of the lab can be found on YouTube at the following link:

https://youtu.be/PXB7WhxC3m4

Conclusions

The main objectives for this lab were met. The Vivado HLx software was used to successfully setup the FPGA. The SDK software was successfully used to code and program the Zybo board. Finally, the design requirements were met, and the code worked as expected.

Appendix

References

Louise H. Crocket, R. A. (2015). *The Zynq Book Tutorials for Zybo and ZedBoard.* Glasgow, Scotland, UK: University of Strathclyde.

Silage, D. D. (Spring 2020). Vivado SDK Basic IP Integrator.

C Code

```
// Include Files
#include "xparameters.h"
#include "xgpio.h"
#include "xstatus.h"
#include "xil_printf.h"

// Definitions for Xilinx Hardware
#define SW_DEVICE_ID   XPAR_AXI_GPIO_0_DEVICE_ID
// GPIO device that Switches are connected to
#define LED_DEVICE_ID  XPAR_AXI_GPIO_1_DEVICE_ID    // GPIO device that LEDs are connected
to
#define LED_DELAY 25000000           // Software delay length
#define SW_CHANNEL 1                 // GPIO port for Switches
#define LED_CHANNEL 1                // GPIO port for LEDs
#define printf xil_printf            // smaller, optimized printf

// GPIO
XGpio SWInst, LEDInst;               // GPIO Device driver instance

// Variables to hold LED state and switch state
static char led_count;                                                     //       Number
displayed on LEDs
static char sw;
       // Switch state

// States for finite state machine
enum state {RESET, HOLD, BAR_GRAPH_INIT, BAR_GRAPH, PATTERN_INIT, PATTERN, ERROR_INIT,
ERROR};

// State variables for flow control
enum state current_state;
enum state next_state;
enum state saved_state;

// Place holders for LED array, length of led array, LED array iterator
char *led_array;
unsigned char led_iter = 0x00;
int led_array_length = 0;

// Define variables related to bar graph
char bg_array[] = {0x00, 0x01, 0x03, 0x07, 0x0F, 0x07, 0x03, 0x01};
#define BG_LENGTH sizeof(bg_array)/sizeof(bg_array[0])

// Define variables related to random pattern state
char pattern_array[] = {0x09, 0x06, 0x0A, 0x05, 0x0C, 0x03};
#define PATTERN_LENGTH sizeof(pattern_array)/sizeof(pattern_array[0])

// Define variables related to random pattern state
char error_array[] = {0x09, 0x03};
#define ERROR_LENGTH sizeof(error_array)/sizeof(error_array[0])

int DecodeSwitchesAndUpdateLEDs(void)
{
       volatile int Delay; // Create variable to hold delay count

       int Status;                   // Create variable to hold Status

       led_count = 0x00;      // Initialize LEDs

       // Initialize Switch GPIO and check status
       Status = XGpio_Initialize(&SWInst, SW_DEVICE_ID);
       if (Status != XST_SUCCESS) {
              return XST_FAILURE;
       }

       // Set the direction for the switches to input
       XGpio_SetDataDirection(&SWInst, SW_CHANNEL, 0xF);
```

```
	// Initialize LED GPIO and check status
	Status = XGpio_Initialize(&LEDInst, LED_DEVICE_ID);
	if (Status != XST_SUCCESS) {
		return XST_FAILURE;
	}

	// Set the direction for the LEDs to output
	XGpio_SetDataDirection(&LEDInst, LED_CHANNEL, 0x0);

	next_state = RESET;		// Initialize next_state
	saved_state = RESET;	// Initialize saved_state

	// Loop forever updating the LEDs based on the switch input
	while (1)
	{
		// Update state based on last pass through while loop
		current_state = next_state;

//----------------------FINITE STATE MACHINE----------------------------
		switch(current_state)
		{
			case RESET: // turn-off LEDs
				led_count = 0x00;
				next_state = RESET;
				break;
			case HOLD: // hold LEDs
				led_count = led_count;
				next_state = HOLD;
				break;
			case BAR_GRAPH_INIT:// initialize the bar graph pattern
				led_array = bg_array;
				led_array_length = BG_LENGTH;
				led_iter = 0x00;
				next_state = BAR_GRAPH;
				break;
			case PATTERN_INIT: // initialize the random pattern
				led_array = pattern_array;
				led_array_length = PATTERN_LENGTH;
				led_iter = 0x00;
				next_state = PATTERN;
				break;
			case ERROR_INIT: // initialize the error pattern
				led_array = error_array;
				led_array_length = ERROR_LENGTH;
				led_iter = 0x00;
				next_state = ERROR;
				break;
			default:  //  Run  LED  pattern  based  on  current_state  (BAR_GRAPH, 
PATTERN, ERROR)
				led_count = led_array[led_iter]; // update LED state
				led_iter++;	// increment iterator
				if (led_iter > led_array_length - 1) led_iter = 0x00; // reset 
iterator
				next_state = current_state;
				break;
		}

		//----------------PRIORITY LOGIC----------------------------

		// Read switches
		sw = XGpio_DiscreteRead(&SWInst, 1) & 0x0F;

		// Check switches to update next_state if necessary
		if (sw % 2 == 1)
		{ // then the lest significant bit is a 1
			next_state = RESET;
		}
		else if (sw == 0x02)
		{ // enter the HOLD state if we're not already in it
			if (current_state != HOLD)
```

```
                        { // save the current_state and set the next state to HOLD
                                saved_state = current_state;
                                next_state = HOLD;
                        }
                }
                else if (sw == 0x04)
                { // Start running the bar graph LED display if not running it already
                        if ((current_state < BAR_GRAPH_INIT)||(current_state > BAR_GRAPH))
                        {
                                next_state = BAR_GRAPH_INIT;
                        }
                }
                else if (sw == 0x08)
                { // Start running the random LED pattern if not running it already
                        if ((current_state < PATTERN_INIT)||(current_state > PATTERN))
                        {
                                next_state = PATTERN_INIT;
                        }
                }
                else if (sw == 0x00)
                { // Keep doing what you're doing unless coming out of the HOLD state
                        if (current_state == HOLD) next_state = saved_state;
                }
                else
                { // all of the error cases are caught here
                        if (current_state < ERROR_INIT)
                        {
                                next_state = ERROR_INIT;
                        }
                }

                // Write output to the LEDs
                XGpio_DiscreteWrite(&LEDInst, LED_CHANNEL, led_count);

                // Wait so user can read LEDs.
                for (Delay = 0; Delay < LED_DELAY; Delay++);
        }

        return XST_SUCCESS; // Should be unreachable
}

// Main function
int main(void){

        int Status;

        // Start running LED pattern based on switch state
        Status = DecodeSwitchesAndUpdateLEDs();
        if (Status != XST_SUCCESS) {
                xil_printf("GPIO output to the LEDs failed!\r\n");
        }

        return 0;
}
```