Vivado AXI Interrupt

Robert Bara
Robert.Bara@temple.edu

Summary

Lab 3, "Vivado AXI Interrupt", introduces hardware interrupts for the Vivado Zynq Processor System. This lab stems from Chapter 2 of *The Zynq Book Tutorials* and utilizes code written in C, programmed to the Zybo Board for LED manipulation, depending on the interrupt generated by each input button.

Introduction

A hardware interrupt is when a signal generated by a hardware device is sent to the processor. In this lab, the input buttons on the Zybo board will be used to generate interrupts that will execute the following tasks for LEDs:

LEDs and display are initialized as 0 or "off".

When an interrupt is generated by Btn0:	LEDs hold the last LED count value, and turn off the LEDs
When an interrupt is generated by Btn1:	LEDs turn on and resume to display the last LED count value.
When an interrupt is generated by Btn2:	The LEDs will output the 1-bit complement of the 4-bit LED count, and is set as the new value of the LED count.
When an interrupt is generated by Btn3:	The binary LED count will increment by 1, and output to the corresponding LEDs.

If the LED count exceeds 4b1111, then there is a roll over back to 4b0000.

Further information about the Hardware design to generate the interrupt, as well as the C program will be discussed in the discussion section, starting with how the *interrupt_controller_tut_2B.C* program runs, and how it will be modified to complete the lab.

Discussion

Hardware Design

The hardware design is like the design created within Lab1. The first step is to add the IP: ZYNQ7 Processing system(processor_system7_0) and use the run automation in Vivado to generate and wire the reset block (rst_ps7_0_100M) and processor(ps7_axi_periph). A Gpio_0 should then be added which wires the push buttons of the Zybo board as an input. To enable an interrupt, click on the Gpio_0(axi_gpio_0) block, select the IP Configuration tab, and select the enable interrupt on the towards the bottom of the screen. Then add an additional output port for the interrupt request:

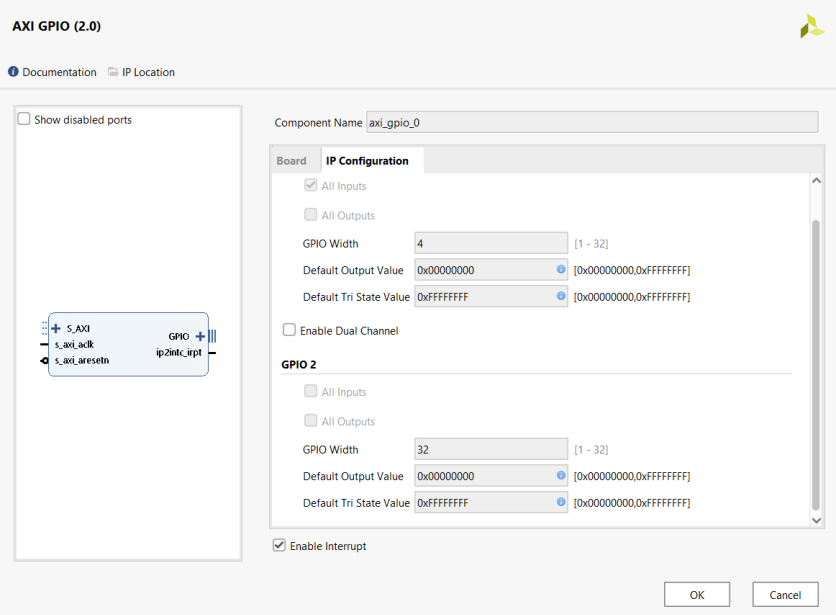

Figure 1. Enable interrupt on GPIO_0

Additionally, another Gpio_1(axi_gpio_1) should wire the LEDs as an output display, without an interrupt since there are no inputs in this Gpio block. Then click on the processing system block and enable the PL-PS interrupt port and select IRQ_F2P:

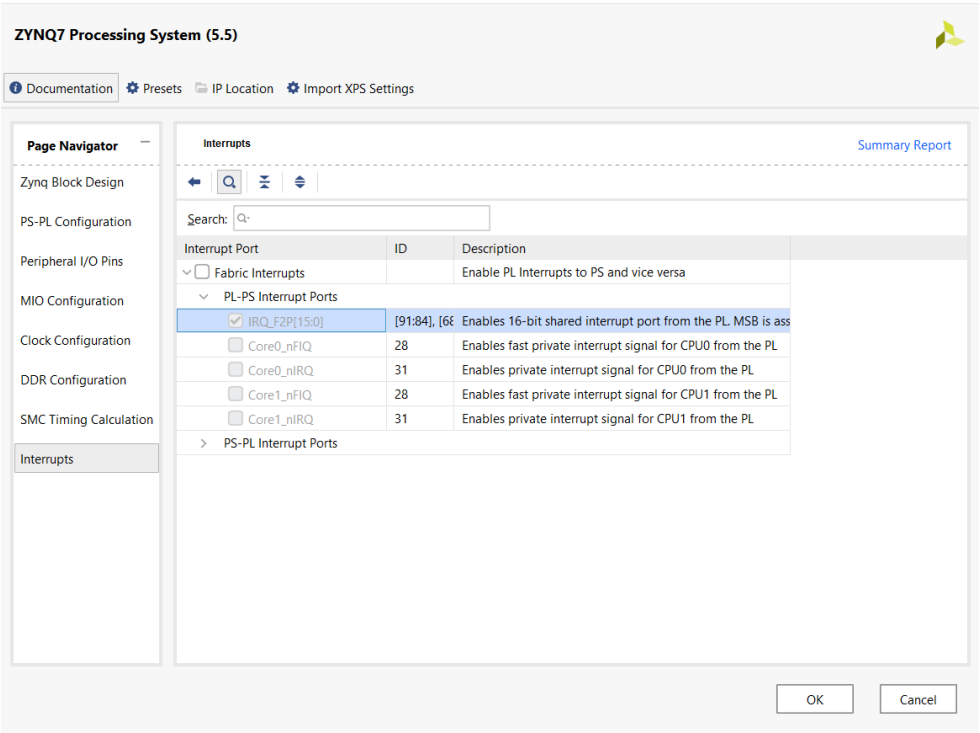

Figure 2. Enabling PL-PS interrupt port

Finally, select the interrupt port from Gpi0 and wire it to the IRQ_F2P interrupt port on the processor system:

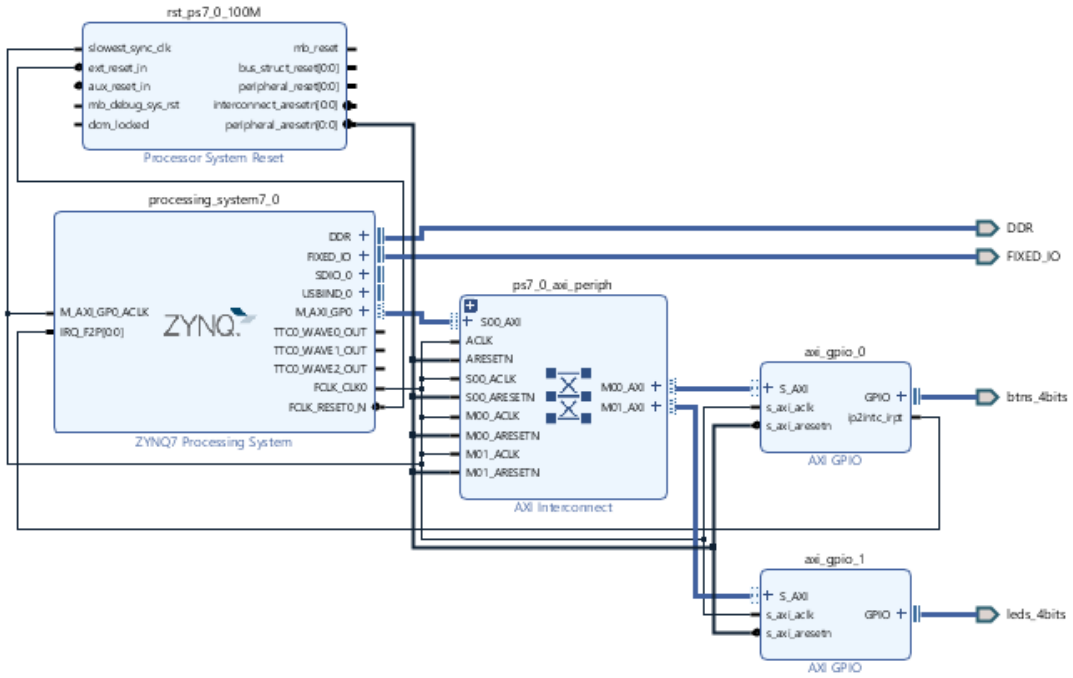

Figure 3. Final Block Diagram with interrupt connection

Save the block diagram, validate the design, create an HDL wrapper, and generate the bitstream. Once the bitstream is written, the design can be exported to hardware, and SDK will be launched for the C program portion of the lab.

C Program

To understand how the program is written, let's examine how the *interrupt_controller_tut_2B.C* runs as a program, and then explain the modifications to the program in order to generate the interrupts for this lab.

Interrupt_controller_tut_2B.C

Initialization

The tutorial begins with initializing the program by including the necessary libraries, defining input devices to their respective hardware Gpio's, and declaring variables:

```
// interrupt_counter_tut_2B.c

// Zynq Tutorial Exercise 2B  ECE3622

#include "xparameters.h"
#include "xgpio.h"
#include "xscugic.h"
#include "xil_exception.h"
#include "xil_printf.h"

// Parameter definitions
#define INTC_DEVICE_ID      XPAR_PS7_SCUGIC_0_DEVICE_ID
#define BTNS_DEVICE_ID      XPAR_AXI_GPIO_0_DEVICE_ID
#define LEDS_DEVICE_ID      XPAR_AXI_GPIO_1_DEVICE_ID
#define INTC_GPIO_INTERRUPT_ID XPAR_FABRIC_AXI_GPIO_0_IP2INTC_IRPT_INTR

#define BTN_INT         XGPIO_IR_CH1_MASK
#define printf          xil_printf

XGpio LEDInst, BTNInst;
XScuGic INTCInst;
static int led_data;
static int btn_value;
```

Figure 4. Includes, definitions, declarations

The function prototypes are then called in which BTN_Intr_Handler function will be called from the timer and check for hardware interrupts from button using a pointer to a base address for the interrupt to occur. The InterruptSystemSetup passes in an instance pointer from the XScuGic header to be worked on. The IntcInitFunction is a setup function that will enable the interrupt and will be discussed further in the subroutine section, this passes in the Zybo's Device ID, and uses a pointer from the XGpio header locate the correct address for the Gpio that will enable the interrupt.

```
//----------------------------------------------------
// PROTOTYPE FUNCTIONS
//----------------------------------------------------
static void BTN_Intr_Handler(void *baseaddr_p);
static int InterruptSystemSetup(XScuGic *XScuGicInstancePtr);
static int IntcInitFunction(u16 DeviceId, XGpio *GpioInstancePtr);
```

Figure 5. Prototyping functions to be used

Similar to previous labs, the main function begins by declaring a variable status to check the initialization of the LEDs and push buttons. This is done by using XGpio_Initialize and passing in their respective IDs and addresses the LEDInst and BTNInst point to. If status is not equal to the macro XST_SUCCESS, then the initialization fails. After this the LEDs are assigned as outputs and btns are assigned as inputs. Since the interrupt controller is also an input that was added to Gpio_0, the same process to check initialization is done for the interrupt controller, where IntcInitFunction looks up the interupt device ID and compares it to the address the BTNInst points to. Upon success, the interrupt controller will be initialized and connect the interrupt to the handler to enable interrupts when a button will be pushed, this is further explained in the IntcInitFunction subroutine:

```
// Initialize interrupt controller
status = IntcInitFunction(INTC_DEVICE_ID, &BTNInst);
if(status != XST_SUCCESS) return XST_FAILURE;
```

Figure 6. Checking the interrupt controller initialization

Since this lab focuses on hardware interrupts, an empty while loop runs in main until an interrupt is generated.

Subroutines

Within the InterruptSystemSetup function, the first part of the function enables the interrupt by calling the function XGpio_InterruptEnable and XGpio_InterruptGlobalEnable and passes in the addresses of the push buttons. Xil_ExceptionRegisterHandler comes from the xil_exception header, which makes a connection between the ID from the exception source and the handler that is supposed to recognize there is an exception, in this case, the interrupt handler and XScuGic instance pointer will locate the interrupt address from the buttons.

```
Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
    (Xil_ExceptionHandler)XScuGic_InterruptHandler,
    XScuGicInstancePtr);
Xil_ExceptionEnable();
```

Figure 7. Xil_Exceptions from the InterruptSystemSetup Function

Within the IntcInitFunction, the interrupt controller is first initialized. IntcConfig is assigned to the lookup configuration function from the XScuGic header, which finds the unique Device ID by using the XScuGic_LookupConfig pointer. Then to check if the interrupt controller is initialized correctly, XScuGic_CfgInitialize examines the address of the interrupt from Gpio_0, the config table for the driver of the device, as well as accesses the physical CpuBaseAddress from the struct using the config table pointers. Upon running successfully, the initialization initializes the vector table with stub function calls and disables all interrupt sources.

```
// Interrupt controller initialisation
IntcConfig = XScuGic_LookupConfig(DeviceId);
status = XScuGic_CfgInitialize(&INTCInst, IntcConfig,
             IntcConfig->CpuBaseAddress);
if(status != XST_SUCCESS) return XST_FAILURE;
```

Figure 8. Interrupt Controller initialization from the IntcInitFunction

The XScuGic_Connect function operates similarly to the XScuGic_CfgInitialize function. This function is responsible for checking the connection between the interrupt to the handler and it does this by pointing to the address of the interrupt from Gpio_0, then it examines the ID of the Gpio_0 interrupt, and makes the connection between the button

interrupt handler function, which will generate the interrupt and determine what happens based upon a hardware interrupt, to the Gpio_0 interrupt that is connected to the controller.

```
// Connect GPIO interrupt to handler
status = XScuGic_Connect(&INTCInst, INTC_GPIO_INTERRUPT_ID,
    (Xil_ExceptionHandler)BTN_Intr_Handler,
    (void *)GpioInstancePtr);
if(status != XST_SUCCESS) return XST_FAILURE;
```

Figure 9.Connecting the GPIO interrupt and checking if it is initialized correctly

The rest of this subroutine simply enables the Gpio interrupts when enabled, and activates the interrupt using pointers as well as the Gpio and timer interrupts are enabled within the controller.

Within BTN_Intr_Handler, the function starts by disabling the interrupt and ignores any btn inputs. Btn is then read to see if any buttons are pressed, and based upon the btn_value, that is btn0=1, btn1=2, btn2=4, btn3=8, then the LEDs will take the current value and add the btn_value to the LEDs and output them using XGpio_DiscreteWrite(&LEDInst, 1, led_data), upon an interrupt. Two lines that are crucial to making this happen are the XGpio_InterruptEnable function, which points to the btn address and enables the interrupt, however it is not until XGpio_InterruptGlobalEnable which points to the Btn address, that actually enables the interrupt output signal. This can be seen at the end of the Btn_Intr_Handler which clears the interrupt so the program can resume after the interrupt is enabled and the output signal is interrupted.

```
void BTN_Intr_Handler(void *InstancePtr) //MODIFY THIS FUNCTION
{
    // Disable GPIO interrupts
    XGpio_InterruptDisable(&BTNInst, BTN_INT);
    // Ignore additional button presses
    if ((XGpio_InterruptGetStatus(&BTNInst) & BTN_INT) !=
            BTN_INT) {
            return;
        }
    btn_value = XGpio_DiscreteRead(&BTNInst, 1);
    // Increment counter based on button value

    led_data = led_data + btn_value;
    printf("Led=Led_data+btn_value=%d\t+\t%d\n",led_data,btn_value);
    XGpio_DiscreteWrite(&LEDInst, 1, led_data);
    (void)XGpio_InterruptClear(&BTNInst, BTN_INT);
    // Enable GPIO interrupts
    XGpio_InterruptEnable(&BTNInst, BTN_INT);
}
```

Figure 10. Tutorial 2B's BTN_Intr_Handler

Modifications for Lab3

Since the interrupts are correctly set up for the push buttons, the only modification to this tutorial code is what will happen when an interrupt is generated by the respective button. This is coded changing the BTN_Intr_Handler function and uses if/else statements to identify which button is being held. Additionally, three variables are created, an int off, which is to set the LEDs equal to 0 (this is technically unnecessary but I think it helps read the program), int next_led, which is actually more so a temporary variable that can be examined in SDK terminal that I use to correct the last led values when there is a roll over from counting up upon enabling btn3's interrupt, and int led, which just takes the final value of led_data after every interrupt and outputs them across the LEDs. The bulk of these modifications are located in this block of if/else statements which can be analyzed based upon the comments written:

```
if(btn_value==0b0001){ //if Btn0 is pressed once, LEDs shut off and sit at last known value
     last_val=led_data;//holds previous led value
     next_led=last_val+1; //predicts what should be the next val if btn3 is pressed
     led=off; //turns off the leds
     printf("Led Count: %d\t Last Value: %d\t next_LED:%d\t button0\n",led_data,last_val,next_led);
    }
else if(btn_value==0b0010){ //if Btn1 is pressed, LEDs hold to last known led value
    led_data=last_val;//resumes the held led after btn0 is pressed
    last_val=led_data;//updates the last value
    next_led=led_data+1;//predicts next value if btn3 is pressed
    led=led_data;//ouputs to leds
    printf("Led Count: %d\t Last Value: %d\t next_LED:%d\t button1\n",led_data,last_val,next_led);
    }
else if(btn_value==0b0100){ //if Btn2 is pressed, LEDs use bitwise operations to perform 1's complement of current LED output
    last_val=led_data;//updates the last value
    led_data=~led_data;//takes the 1's compliment of the current led value
    next_led=led_data+1;//predicts next value if btn3 is pressed
    led=led_data;//outputs to leds
    printf("Led Count: %d\t Last Value: %d\t next_LED:%d\t button2\n",led_data,last_val,next_led);
    }
else if(btn_value==0b1000){//if Btn3 is pressed, LEDs increment by 1 for each btn depression
    led_data=next_led;//updates based upon a rollover
    last_val=led_data;//updates the last known value
    next_led=led_data+1;//predicts the next value incremented by 1
    led_data=led_data+1;//the current led value that gets incremented by 1 for each time an interrupt is generated
    led=led_data;//outputs to leds
    //priority logic
    //When LED=0b1111, roll over to LED=0b0000
    if(led_data>15){led_data=0;last_val=15;next_led=led_data+1;}//for positive roll overs
    if(led_data<0&&led_data>-1){led_data=0;last_val=-1;next_led=led_data+1; }//for negative roll overs

     printf("Led Count: %d\t Last Value: %d\t next_LED:%d\t button3\n",led_data,last_val,next_led);
    }
else{//if multiple btns are pressed, or any thing else happens, do nothing
    asm("NOP");
}
```

Figure 11. Modfied section of tutorial 2B's BTN_Intr_Handler, the cases of each btn interrupt

After these interrupts are executed, XGpio_DiscreteWrite(&LEDInst, 1, led) is called to read the value of led, upon exiting the conditional.

Verification

The hardware demo for this lab may be viewed by the following link:

Check Canvas Comments this video may not be finished uploading by 2/16/2021, I may have to upload this over night, in which the upload may fail, at the time I am not sure if this link will work: https://youtu.be/VPym3WcUhHk

In terms of verification, I believe I have successfully implemented hardware interrupts and coded their corresponding LED outputs based upon interrupts generated by the push buttons on the Zybo board. The only factor that I may have left out, is if I should have included a button debounce to verify the interrupt, but that was not stated within the lab manual. Furthermore, you could see based upon how I programmed btn0's interrupt, pushing btn0 twice will reset the LEDs and counter, which one could argue is a system reset. While I explore various combinations of pushing buttons in the video to show no overlap exists, I will include the following screenshots to verify that each individual button generates the correct interrupt.

Due to slow wifi, please bare in mind the lower quality pictures of my web camera over my phone, it could have taken forever to email myself the photos taken on my phone, thankyou

LED=3=4b0011, using btn3 to increment to LED=4=4b0100, using btn2 to restore previous LED value:

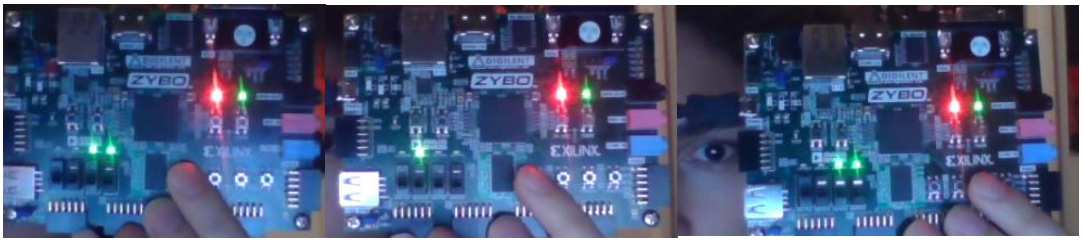

Figure 12a. Before incrementing 12b. After incrementing 12c. Btn2 previous Value

LED=3, turning off LEDs, then turning back on with btn1, then taking the 1's compliment with btn2, then incrementing the new value with btn3:

Figure 13a. Before shutting off 13b. After shutting off

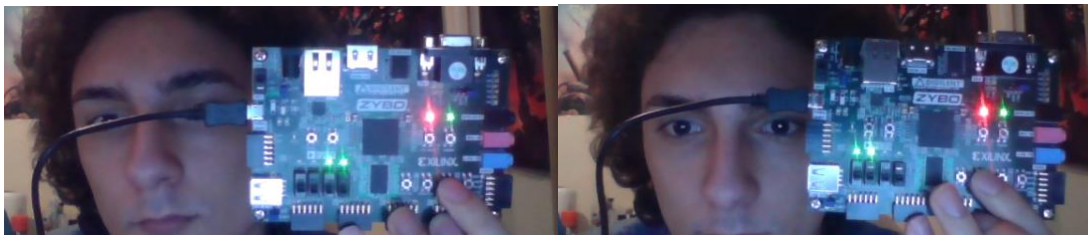

Figure 13c. Turning back up by restoring previous value 13d. Displaying 1's compliment

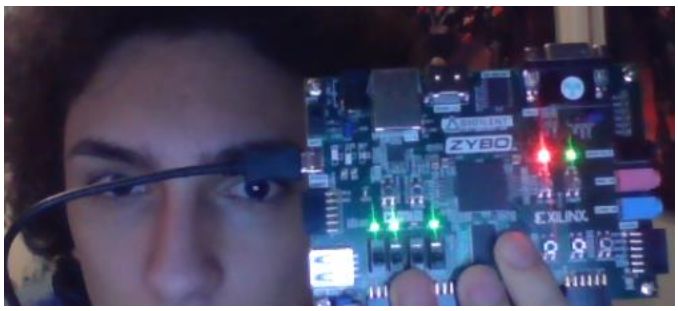

Figure 13e. Incrementing new LED value

Conclusion

This lab introduced hardware interrupts by mapping interrupts to the push buttons on the Zybo board and outputting various manipulations of LEDs, based upon which button generated a specific interrupt. I was successfully able to program the interrupts and verify

with my board that the interrupts function as they should based upon the lab manual. The most challenging part of this lab was examining through the tutorial 2B's program's header files, to understand how the hardware of the Zybo board is actually enabling and disabling an interrupt.

Appendix

C Code

```
// Lab 3 Robert Bara, Vivado Axi Hardware Interrupt

#include "xparameters.h"
#include "xgpio.h"
#include "xscugic.h"
#include "xil_exception.h"
#include "xil_printf.h"

// Parameter definitions
#define INTC_DEVICE_ID          XPAR_PS7_SCUGIC_0_DEVICE_ID
#define BTNS_DEVICE_ID          XPAR_AXI_GPIO_0_DEVICE_ID
#define LEDS_DEVICE_ID          XPAR_AXI_GPIO_1_DEVICE_ID
#define INTC_GPIO_INTERRUPT_ID XPAR_FABRIC_AXI_GPIO_0_IP2INTC_IRPT_INTR

#define BTN_INT           XGPIO_IR_CH1_MASK
#define printf                  xil_printf

XGpio LEDInst, BTNInst;
XScuGic INTCInst;
static int led_data;//initialize LEDs at 0
static int btn_value;
static int last_val=0;//initialize temporary hold value as 0
//----------------------------------------------------
// PROTOTYPE FUNCTIONS
//----------------------------------------------------
static void BTN_Intr_Handler(void *baseaddr_p);
static int InterruptSystemSetup(XScuGic *XScuGicInstancePtr);
static int IntcInitFunction(u16 DeviceId, XGpio *GpioInstancePtr);

//----------------------------------------------------
// INTERRUPT HANDLER FUNCTIONS
// - called by the timer, button interrupt, performs
// - LED flashing
//----------------------------------------------------

void BTN_Intr_Handler(void *InstancePtr)
{      int off=0;//turns off LEDs
       int next_led=led_data;//temporary variable for rollovers
       int led;//variable to actually output led_data to leds
       // Disable GPIO interrupts
       XGpio_InterruptDisable(&BTNInst, BTN_INT);
       // Ignore additional button presses
       if ((XGpio_InterruptGetStatus(&BTNInst) & BTN_INT) !=
                    BTN_INT) {
                    return;
             }
       btn_value = XGpio_DiscreteRead(&BTNInst, 1);

       if(btn_value==0b0001){ //if Btn0 is pressed once, LEDs shut off and sit
at last known value
              last_val=led_data;//holds previous led value
```

```
                next_led=last_val+1; //predicts what should be the next val if
btn3 is pressed
                led=off; //turns off the leds
                printf("Led Count: %d\t Last Value: %d\t next_LED:%d\t
button0\n",led_data,last_val,next_led);
               }
       else if(btn_value==0b0010){ //if Btn1 is pressed, LEDs hold to last
known led value
              led_data=last_val;//resumes the held led after btn0 is pressed
              last_val=led_data;//updates the last value
              next_led=led_data+1;//predicts next value if btn3 is pressed
              led=led_data;//ouputs to leds
              printf("Led Count: %d\t Last Value: %d\t next_LED:%d\t
button1\n",led_data,last_val,next_led);
               }
       else if(btn_value==0b0100){ //if Btn2 is pressed, LEDs use bitwise
operations to perform 1's complement of current LED output
              last_val=led_data;//updates the last value
              led_data=~led_data;//takes the 1's compliment of the current led
value
              next_led=led_data+1;//predicts next value if btn3 is pressed
              led=led_data;//outputs to leds
              printf("Led Count: %d\t Last Value: %d\t next_LED:%d\t
button2\n",led_data,last_val,next_led);
               }
       else if(btn_value==0b1000){//if Btn3 is pressed, LEDs increment by 1 for
each btn depression
              led_data=next_led;//updates based upon a rollover
              last_val=led_data;//updates the last known value
              next_led=led_data+1;//predicts the next value incremented by 1
              led_data=led_data+1;//the current led value that gets incremented
by 1 for each time an interrupt is generated
              led=led_data;//outputs to leds
              //priority logic
              //When LED=0b1111, roll over to LED=0b0000
              if(led_data>15){led_data=0;last_val=15;next_led=led_data+1;}//for
positive roll overs
              if(led_data<0&&led_data>-1){led_data=0;last_val=-
1;next_led=led_data+1; }//for negative roll overs

                printf("Led Count: %d\t Last Value: %d\t next_LED:%d\t
button3\n",led_data,last_val,next_led);
               }
       else{//if multiple btns are pressed, or any thing else happens, do
nothing
              asm("NOP");
       }
    XGpio_DiscreteWrite(&LEDInst, 1, led);
    (void)XGpio_InterruptClear(&BTNInst, BTN_INT);
    // Enable GPIO interrupts
    XGpio_InterruptEnable(&BTNInst, BTN_INT);
}

//----------------------------------------------------
// MAIN FUNCTION
```

```
//----------------------------------------------------
int main (void)
{
  int status;
  //----------------------------------------------------
  // INITIALIZE THE PERIPHERALS & SET DIRECTIONS OF GPIO
  //----------------------------------------------------
  // Initialise LEDs
  status = XGpio_Initialize(&LEDInst, LEDS_DEVICE_ID);
  if(status != XST_SUCCESS) return XST_FAILURE;
  // Initialise Push Buttons
  status = XGpio_Initialize(&BTNInst, BTNS_DEVICE_ID);
  if(status != XST_SUCCESS) return XST_FAILURE;
  // Set LEDs direction to outputs
  XGpio_SetDataDirection(&LEDInst, 1, 0x00);
  // Set all buttons direction to inputs
  XGpio_SetDataDirection(&BTNInst, 1, 0xFF);

  led_data=0;
  XGpio_DiscreteWrite(&LEDInst, 1, 0); //initialize LEDs as off

  // Initialize interrupt controller
  status = IntcInitFunction(INTC_DEVICE_ID, &BTNInst);
  if(status != XST_SUCCESS) return XST_FAILURE;

  while(1);

  return 0;
}

//----------------------------------------------------
// INITIAL SETUP FUNCTIONS
//----------------------------------------------------

int InterruptSystemSetup(XScuGic *XScuGicInstancePtr)
{
	// Enable interrupt
	XGpio_InterruptEnable(&BTNInst, BTN_INT);
	XGpio_InterruptGlobalEnable(&BTNInst);

	Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
		(Xil_ExceptionHandler)XScuGic_InterruptHandler,
		XScuGicInstancePtr);
	Xil_ExceptionEnable();

	return XST_SUCCESS;
}

int IntcInitFunction(u16 DeviceId, XGpio *GpioInstancePtr)
{
	XScuGic_Config *IntcConfig;
	int status;

	// Interrupt controller initialisation
	IntcConfig = XScuGic_LookupConfig(DeviceId);
```

```
	status = XScuGic_CfgInitialize(&INTCInst, IntcConfig,
			IntcConfig->CpuBaseAddress);
	if(status != XST_SUCCESS) return XST_FAILURE;

	// Call to interrupt setup
	status = InterruptSystemSetup(&INTCInst);
	if(status != XST_SUCCESS) return XST_FAILURE;

	// Connect GPIO interrupt to handler
	status = XScuGic_Connect(&INTCInst, INTC_GPIO_INTERRUPT_ID,
		(Xil_ExceptionHandler)BTN_Intr_Handler,
		(void *)GpioInstancePtr);
	if(status != XST_SUCCESS) return XST_FAILURE;

	// Enable GPIO interrupts interrupt
	XGpio_InterruptEnable(GpioInstancePtr, 1);
	XGpio_InterruptGlobalEnable(GpioInstancePtr);

	// Enable GPIO and timer interrupts in the controller
	XScuGic_Enable(&INTCInst, INTC_GPIO_INTERRUPT_ID);

	return XST_SUCCESS;
}
```