Lab 1 – Vivado SDK Basic IP Integrator

Leomar Durán

Summary

This analysis involves an IP integrator. It uses a GPIO to accept input from the buttons, and uses that input to control a second GPIO which sends output to the LEDs. This design demonstrates an infinite loop for accepting the input and updating the LEDs for output.

The motivation for this analysis is that the student will be able to create future experiments with multiple IP components and understand how the input and output loop works.

Introduction

The objective of this analysis is that we want to be able to select various patterns that the LED will cycle through by using the buttons.

An alternative analysis that was considered is to perform the same experiment, but using interrupts when the button is pressed, rather than the loop to accept input.

Discussion

The hardware design used as a version of the hardware design from 2B, but without the interrupts, in Fig 1. This hardware design uses two GPIOs, one for the buttons, and another for the LEDs, so it can also be used to represent this problem.

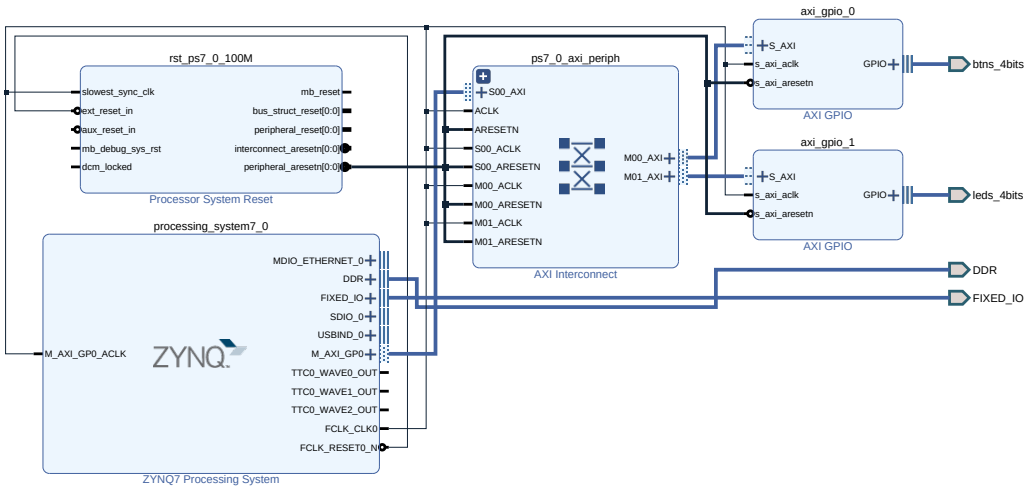

Figure 1: Hardware design used for the analysis.

The software problem was solved using a 7-state finite state machine. This solution was chosen primarily because of when the design counts by 2s and 4s, which is after leaving the SUSPEND state, that is, it will be start counting by 2s for example, if the student presses

button 0 while the design is counting by 1s.

The default logic is to set the increment to 0 unless the state is COUNT_AFTER_SUSPEND.

The states are adapted by the finite state machine in Fig. 2.

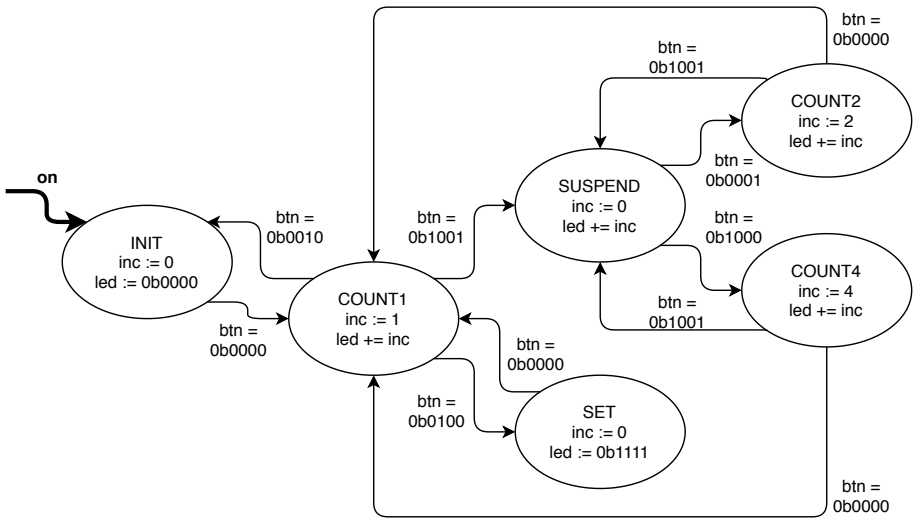

Figure 2: Finite state machine for the analysis before optimization.

The finalized states are as follows:

INIT sets the LED counter to 0. This is the original state of the design when it starts.

COUNT1 sets the increment to 1. If buttons 3 and 0 are pressed, it moves to SUSPEND. If button 1 is pressed, it moves back to INIT. If button 2 is pressed, it moves to SET.

SUSPEND has the effect of suspending the LED counter at its current state. It just moves to COUNT2 if button 3 is released. Either that or if button 0 is released, the state is instead COUNT4.

SET sets the LED counter to all LEDs on.

COUNT2 sets the increment to 2 and unconditionally moves to COUNT_AFTER_SUSPEND.

COUNT4 sets the increment to 4 and unconditionally moves to COUNT_AFTER_SUSPEND.

COUNT_AFTER_SUSPEND will move to suspend if both buttons 3 and 0 are pressed again. This state means one of the counts after suspend, so that the count continues.

The priority logic is to increase the LED counter by the increment, and then to go to COUNT1 if no buttons are pressed, unless the state is SUSPEND, which does not go directly to COUNT1.

This priority logic was chosen rather than introducing a state after INIT and SET because COUNT_AFTER_SUSPEND follows the same pattern, so it was simpler to put this in the priority logic and exclude SUSPEND instead.

The results can be seen at `https://youtu.be/b2_hd8kkWoY`

Conclusions

The objective of this pattern is to use the buttons to control a counter that is displayed by the LEDs. This design satisfies the objective because it matches those patterns.

Appendices

The following pages include the complete source code, which is also available at `https://github.com/lduran2/ece3623-lab1-LED_BUTTON/blob/master/LED_BUTTON.prj.sdk/LED_BUTTON/src/LED_test_tut_1C.c`

```
/*
 * LED_test.c
 *
 *  Created on:     13 June 2013
 *      Author:     Ross Elliot
 *     Version:     1.2
 */

/*****************************************************************************
***************
* VERSION HISTORY
*****************************************************************************
**************
*   v1.5 - 14 September 2020
*       Optimized the finite state machine.
*
*   v1.4 - 13 September 2020
*       Implemented the finite state machine described in ece3623-lab1-
fsm.drawio
*
*   v1.3 - 13 September 2020
*       Integrated the button definitions and setup from
interrupt_controller_tut_2B.c
*
*   v1.2 - 13 February 2015
*       Modified for Zybo Development Board ~ DN
*
*   v1.1 - 27 January 2014
*       GPIO_DEVICE_ID definition updated to reflect new naming conventions
in Vivado 2013.3
*       onwards.
*
*   v1.0 - 13 June 2013
*       First version created.
*****************************************************************************
*************/

/*****************************************************************************
***************
 * This file contains an example of using the GPIO driver to provide
communication between
 * the Zynq Processing System (PS) and the AXI GPIO block implemented in the
Zynq Programmable
 * Logic (PL). The AXI GPIO is connected to the LEDs on the Zybo.
 *
 * The provided code demonstrates how to use the GPIO driver to write to the
memory mapped AXI
 * GPIO block, which in turn controls the LEDs.

*****************************************************************************
**************/

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

```
/* Definitions */
#define BTNS_DEVICE_ID      XPAR_AXI_GPIO_0_DEVICE_ID
#define GPIO_DEVICE_ID  XPAR_AXI_GPIO_1_DEVICE_ID   /* GPIO device that LEDs
are connected to */
#define LED 0b0000                                  /* Initial LED value -
X00X */
#define LED_DELAY 20000000                          /* Software delay length
(twice for visualization) */
#define LED_CHANNEL 1                               /* GPIO port for LEDs */
#define printf xil_printf                           /* smaller, optimised
printf */

XGpio BTNInst;                                  /* GPIO Device driver
instance for buttons */
XGpio LEDInst;                                  /* GPIO Device driver
instance for LEDs */

int LEDOutputExample(void)
{

    volatile int Delay;
    int Status;
    int inc = 0;    /* Increment for the LED counter */
    int led = LED; /* Hold current LED value. Initialise to LED definition */
    int btn;    /* Hold current button value */

    /* the possible states of the finite state machine
     *   INIT = initial state, all LEDs off
     *   COUNT1 = count by ones,
     *   COUNT2 = count by twos,
     *   COUNT4 = count by fours,
     *   SUSPEND = keep LEDs in last stage,
     *   COUNT_AFTER_SUSPEND = keep counting after changing from suspend
     *   SET = all LEDs on
     *   */
    enum { INIT, COUNT1, COUNT2, COUNT4, SUSPEND, SET, COUNT_AFTER_SUSPEND }
state;

        //-----------------------------------------------------
        // INITIALIZE THE PERIPHERALS & SET DIRECTIONS OF GPIO
        //-----------------------------------------------------
        /* GPIO driver initialisation */
        Status = XGpio_Initialize(&LEDInst, GPIO_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);

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

```
/* Loop forever blinking the LED. */
    while (1) {
        /* Write output to the LEDs. */
        XGpio_DiscreteWrite(&LEDInst, LED_CHANNEL, led);
        /* Read input from the switches. */
        btn = XGpio_DiscreteRead(&BTNInst, 1);

        /* default logic */
        /* hold the count, unless counting after suspend */
        if (state != COUNT_AFTER_SUSPEND) {
            inc = 0;
        }

        /* state logic */
        switch (state) {
            case INIT:
                led = 0b0000;
            break; /* case INIT */

            case COUNT1:
                inc = 1;
                switch (btn) {
                    /* initialize again if button 1 is pressed */
                    case 0b0010:
                        state = INIT;
                    break; /* case 0b0010 */
                    /* suspend if buttons 3 and 0 are pressed */
                    case 0b1001:
                        state = SUSPEND;
                    break; /* case 0b1001 */
                    /* set if button 2 is pressed */
                    case 0b0100:
                        state = SET;
                    break; /* case 0b1001 */
                    default: break;
                } /* end switch (btn) */
            break; /* case COUNT1 */

            case SUSPEND:
                switch (btn) {
                    /* count by 2s if button 3 is released */
                    case 0b0001:
                        state = COUNT2;
                    break; /* case 0b0001 */
                    /* count by 4s if button 0 is released */
                    case 0b1000:
                        state = COUNT4;
                    break; /* case 0b1000 */
                    default: break;
                } /* switch (btn) */
            break; /* case SUSPEND */

            case SET:
                led = 0b1111;
            break; /* case SET */
```

```
                    case COUNT2:
                        inc = 2;
                        state = COUNT_AFTER_SUSPEND;
                    break; /* case COUNT2 */

                    case COUNT4:
                        inc = 4;
                        state = COUNT_AFTER_SUSPEND;
                    break; /* case COUNT4 */

                    case COUNT_AFTER_SUSPEND:
                        /* suspend if both buttons pressed again */
                        if (btn == 0b1001) {
                            state = SUSPEND;
                        }
                    break;

                    default: break;
                }

                /* priority logic */
                led += inc;
                /* go back to COUNT1 if no buttons are pressed, *unless*
state is SUSPEND */
                if ((btn == 0b0000) && (state != SUSPEND)) {
                    state = COUNT1;
                }

                /* Wait a small amount of time so that the LED blinking is
visible. */
                for (Delay = 0; Delay < LED_DELAY; Delay++);
            }

        return XST_SUCCESS; /* Should be unreachable */
}

/* Main function. */
int main(void){

    int Status;

    /* Execute the LED output. */
    Status = LEDOutputExample();
    if (Status != XST_SUCCESS) {
        xil_printf("GPIO output to the LEDs failed!\r\n");
    }

    return 0;
}
```