Basic I/O, Timers and Interrupts - Laboratory #3 Report

ECSE 324 - Computer Organization

October 31st, 2018

Sarah Swanson - 260660847

Bryan Jay - 260738764

Group 19

Introduction

The purpose of laboratory #3 is to learn the basics of I/O capabilities on the DE1-SoC computer. We will look into the slider switches, push buttons, LEDs and the 7-segment displays. Timers and interrupts will also be used to demonstrate polling and interrupt based applications that are written in C.

Part 1 - Creating the Project

The first part of the laboratory was to setup the correct parameters in order to be able to continue the lab using interrupts. The architecture was set to ARM Cortex-A9 and the system selected was De1-SoC Computer with the program type as C Program. The ‘linker section presents’ was changed from ‘basic’ to ‘exceptions’.

Part 2 - Basic I/O

The purpose of part two of the laboratory was to understand the hardware of the I/O components. Also, the section deals with writing assembly code to control the I/O interact by reading from and writing to memory.

Part 2.1 - Drivers for Slider Switches and LEDs

The purpose of part 2.1 of the laboratory was to understand the I/O capabilities of the De1-SoC computer: primarily the LEDs and slider switches on the board. The main purpose of this challenge was to be able to light up the LED that corresponded with the correct slider switch. The switches state is sent to the LED in an infinite while loop in order to always be able to switch on the LED.

The code was mostly given for the implementation of lighting up the LED lights to turn on and off when the corresponding slider switch is toggled. The code we wrote was in the ‘main.c’ file under ‘drivers’, where we made an infinite while loop that continuously sent the switches state to the LEDs. This was done by polling the register that had the switches data, where the slider switch was associated to 1 bit. The function given to us was the read_slider_switches_ASM() which was a subroutine that loaded the value at the LED’s memory location into R0 and then branched to the link register, which essentially gave the value of the slider switch. We were tasked to write a similar function called write_LEDS_ASM() to be able to store the value in R0 at the LED’s memory location, then branch to the link register. This essentially placed the value of the slider switch into the LED register. We did this the same way read_slider_switches_ASM()

was done, except for instead of loading the contents of R1 (which points to the LED's address and contents) into R0, we stored the contents of R0 into R1.

This part of the laboratory was pretty simple because of the code given, so there were not many hard challenges. An improvement could have been to use an interrupt system because polling uses more CPU than interrupt based system does. Continuously polling the switches constantly uses the CPU.

Part 2.2 - Drivers for HEX Displays and Push-Buttons

The purpose of part 2.2 of the laboratory was to learn how to use the pushbuttons and 7 segments display on the board in order to be able to display more than one digit at the same time. The drivers that were created use enumerations in C.

We were challenged to write three functions in order to interact with the 7 segment display. The three functions are: HEX_clear_ASM, HEX_flood_ASM and HEX_write_ASM. The first function, HEX_clear_ASM, was to be able to turn off all the segments of all the HEX displays passed in the argument. We wrote this function by designating a register to have a null value (zero) and putting zero into any of the HEX displays that were associated with being turned on. To know that the HEX display was on, the bitwise number involved with the logical AND function was determined (one hot encoding scheme). We looped through the HEX displays HEX0-HEX3 in one loop and made sure all of the lights were turned off. Then we looped through HEX displays HEX4-HEX5 with the same process. The second function, HEX_flood_ASM, was implemented to be able to turn on all of the segments that were passed as parameters. We used a loop that determined which bit had the value of 1 (since one hot encoded) to get the HEX display, and then stored the flood bit into the address of the HEX display to turn it on. This looped through the HEX displays HEX0-HEX3 in one loop to turn on the lights of the segments that were passed as parameters. Then we looped through HEX displays HEX4-HEX5 with the same process. This turned all the lights on for the segments that were passed as parameters. The third function, HEX_write_ASM, was to take a second argument 'val' (a number between 0-15) and based on that number, the subroutine would display the corresponding hexadecimal digit on the display. We used a loop that determined which bits had the value of 1 (since one hot encoded) to get the HEX display, and if it was 1, we cleared it. Once 0, we stored the desired value into the HEX display from the HEX_VAL (array of bytes of the bit values for numbers 0-15). We looped through the HEX displays HEX0-HEX3 in one loop and another loop for HEX4-HEX5 with the same process. The pushbutton file 'pushbutton.s' simply let us press the four buttons on the board to light up the corresponding spot on the display.

A challenge encountered during this section was to collaborate the pushbuttons with the display by writing the functionality for the 7 subroutines to access the data, edgecapture and interrupt mask registers. An improvement could have been to use and interrupt based system because this would make the CPU more efficient.

Part 3 - Timers

The purpose of part three of the laboratory was to learn about timers. Timers are a hardware counter that measures time or synchronizes events. We were challenged to create a stopwatch that increments its count every 10 milliseconds using the display values for HEX0-HEX5 LEDs and the pushbuttons (three of them) in order to be able to reset, start and pause the stopwatch. We used a single HPS timer to count time, with milliseconds on HEX1-0, seconds on HEX3-2 and minutes on HEX5-4. PB0, PB1 and PB2 are used on the stopwatch to start, stop and reset respectively. Another HPS timer is used to set a faster timeout value of 5 milliseconds or less in order to poll the pushbutton edgecapture register.

The challenge was to create the stopwatch and implement the timers. The implementation of display HEX0-HEX5 and pushbuttons were already done from previous parts of this laboratory. In order to implement to timer, we stored it in an address in memory and created a structure. A structure is a composite data type that allows a single pointer to access a grouping of several variables. Our structure had the timeout time, the LD_en value, the INT_en value, the name of the structure variable and the timer enable variable to know if it is on or not. Each of the values are assigned to a specific bit within the register, which gives easy access to the values defined above by accessing the register with the equivalent offset of the structure value.

We created two functions for two timers: one to count time and one to poll the pushbuttons edgecapture register. The first function keeps track of the time, with milliseconds, seconds and minutes, incrementing the stopwatch every 10 milliseconds. The function checks the start timer to see if it’s boolean value is set to true. If true, the HEX displays of HEX0-HEX5 are updated based on the timer’s values of time kept. The second function polls the pushbuttons every 5ms to see if any of the buttons were pressed. There are three pushbuttons than can be pressed. When the first one is pressed, the timer is set to 1 and the timer starts and shows the HEX display of the values of the timer. When the second button is pressed, the timer is paused and sets the start timer value back to 0. When the third button is pressed, it clears the values of the timer and display and sets the timer value to 0.

A challenge faced in this part of the laboratory was understanding the code given to us and how the headers are used within the stopwatch and also understanding how to use a structure. The structure involved many variables which we had to figure out where to use and when. An improvement for our two functions could have been to again, use interrupts instead of polling to improve the efficiency of the CPU.The processor handles a lot of tasks at once so interrupts would have been a major improvement in this case.

Part 4 - Interrupts

The purpose of part four of the laboratory was to learn about interrupts. Interrupts are hardware or software signals that are sent to the processor to indicate that an event has occurred that needs immediate attention. The processor pauses the current code execution when it receives the interrupt and executes the code defined in an Interrupt Service Routine (ISR), then continues normal execution. Our challenge was to implement the same stopwatch as part 3, but instead use an interrupt to know if a button has been pressed instead of continuously polling the pushbuttons. This should maximize the use of the CPU and fix the improvement we mentioned in part 3 of the report. The Linker Section Presets has to be changed from ‘Basic’ to ‘Exceptions’ for this part of the laboratory.

In order to implement the interrupts, a pushbutton flag was created and set to an empty pushbutton (PB4), which is also used to reset the pushbutton flag. To see which pushbutton was pressed from the interrupt sent, the fpga_pb_int_flag function uses the edge capture register to compare the value put in that register to the value of the pushbuttons. The function also clears the register in order for another pushbutton to be pressed. A while loop was created to check if and when the interrupt occurs. It also checks if the timer interrupt occurs. If the interrupt is pushbutton zero, the timer is set to 1 to start the timer. If the interrupt is pushbutton 1, the timer is set to 0 to stop the timer. If the interrupt is pushbutton 2, the timer is set to 0 and all of the timer display values are also set to 0 (ms, sec, min). All the displays are reset to show 0. After all of interrupts for all pushbuttons, the interrupt flag is cleared to the pushbutton 4 (which is empty, no use).

A challenge encountered in this part of the laboratory was understanding how to implement an interrupt though the file ‘int_setup.c’ and understand how to use the given code’s functions and parameters within our own code. An improvement for this part of the laboratory would be find a way to use the timer without polling it because it continuously uses the CPU, which was avoided with the other interrupts in this program.