Group 30
Zhuonan Wang 260714137
Suhui Shen 260705934

ECSE 324 Lab3 Report

Lab 3 is mostly focused on basic functions of I/O, input and output. The DE1-SoC computer was used to run the ARM code that incorporates switches, LEDs, 7-Segment display, pushbuttons, timers, and interrupts.

1 Basic I/O

1.1 Slider Switches

This part of the lab is for us to get started with I/O functions. The codes for the assembly file and the header file were provided for us. In the assembly file there is a subroutine labelled read_slider_switches_ASM that will read the data of the slider switches into R0, which would later act as our input. Inside the subroutine we load the value at the base address of the slider switches on to a register using two LDR instructions. At the end of this subroutine it's branched back to the link register. The header file is to declare the function of reading the data of the slider switches, which is going to be applied in the main C program in the driver.

1.2 LEDs

This program was very similar to the program of slider switches. For LEDs assembly file we had two different functions including read_LEDs_ASM and write_LEDs_ASM. Also, different from slider switches, this time the base addresses of the LEDS were used. The Read function was the same as Read Slider Switches, which would again load the data for LEDs into R0 and then branch to link register. For the Write function, a STR instruction was used as we were to accept an argument and store it at the base address of LEDs. In the header file, again we need to declare both of the functions. Specifically for the write_LEDs_ASM, it needs to accept an argument int val, and the function is void as it returns nothing.

1.3 Integration

Since our goal is to control the LEDs with the slider switches, we need to combine the two parts together in the C file. An infinite loop is used here because we need to continuously check the input from the push buttons, and write the value to LEDs.

2 Drivers for HEX displays and pushbuttons

2.1 HEX displays

Here we need to write 3 functions, including HEX_clear_ASM(HEX_t hex), HEX_write_ASM(HEX_t hex) and HEX_flood_ASM(HEX_t hex, char val). Clear is meant to store all bits as 0s at the addresses that correspond to the displays so nothing shows up in the display. Here the clear_loop continuous writes one byte of zeros to the display until all displays are 0. Flood does the opposite and stores all 1s and each segment is on. Similarly, the flood_loop continuously writes one byte of ones until all displays are 1. Write takes an additional parameter that is the character to write to the specified displays. It first clears the input and then checks which number it is by comparing with number 0 to 15. When the number is confirmed, the corresponding display number is written.

2.2 Pushbuttons

The functions used in this part include read_PB_data_ASM, PB_data_is_pressed_ASM(PB_t PB), read_PB_edgecap_ASM(), PB_edgecap_is_pressed_ASM(), PB_clear_edgecap_ASM(), enable_PB_INT_ASM() and disable_PB_INT_ASM(PB_t PB). Read_PB_data_ASM load the information in data register. PB_data_is_pressed_ASM(PB_t PB) checks which push button is pressed. PB_edgecap_is_pressed_ASM() and PB_clear_edgecap_ASM() deal with edge capture register in which bits are changed from 1 to 0 if push buttons are pressed. The last two functions deal with interrupt for pushbuttons.

2.3 Integration

The next part was to create a program in C that integrated all the existing parts into one basic I/O program. It would highlight that each subroutine was implemented correctly and did not have any extraneous effects. The integration had to maintain that at any time when a switch was flipped that the LED next to the switch was also turned on. Then the system needs to read the input from the first four switches as binary and write it the HEX display that corresponds with the pushbutton that is pressed. This is implemented by truncating the read data from the switches to the first 4 bits only. Then use the HEX write function and pass the pushbutton that was pressed so that the switch data could be displayed on the correct hex display. The third part is to constantly flood the final two displays, which can be accomplished by just calling the HEX flood method and passing those two displays. Then it is necessary to clear all the displays when switch 9 is flipped. We can simply call HEX_clear when switch 9 is flipped.

3 Timers: Stopwatch with Polling

The task was to create a stopwatch with the hex displays. First of all, we need to three subroutines, HPS_TIM_config_ASM(), HPS_TIM_read_ASM(), and HPS_TIM_clear_INT_ASM() to interface with the HPS timer drivers that are in the DE1 SoC. This is the foundation of the timer system. The configuration subroutine takes a struct pointer as an argument. It takes a base address for the timers, read the s-bit of each of the timers. The read subroutine would read the value of the s-bit and see if the function is able to read the s-bit value of multiple timers in the same call. The clear subroutine would initialize the timer by resetting the s-bit and the f-bit. Clear and read also both accept an enumeration that corresponds to the timer being requested.

The timers are configured by looping through the timers the same was we did with the HEX displays. The timer is set up by altering the data stored at its address. To alter that data it first has to be read, the time is loaded, as well as the E, I, and M bits individually. They each have to be shifted so that they can be added together in the form that is specified in the manual. The compiled data is then stored back to the timers address.

To create the stopwatch, we need two timers working together. One with increments of 10 milliseconds is used for the stopwatch and the other one with increments of 5 milliseconds is used as the poller to check if any button is pushed. In the C file, we had three variables corresponding to milliseconds, seconds, and minutes initialized to be zero at the beginning, and another two variables named push_buttons and timer_start used as boolean values are initialized to be zero. Then we enter the infinite while loop. The first part in the loop was the section that configures the counting features. We reset milliseconds to 0 when it reaches 1000, reset second count and minute count to 0 when they reach 60. In each while iteration, we call the HEX_write_ASM and update the 6 hex displays. The second part was the polling with TIM1. Here the buttons are continuously polled to check whether the timer should count, pause, or reset and clear the displays.

4 Stopwatch with Interrupts

In this part, we need to implement a stopwatch that has an interrupt service while the buttons are pressed. Different from the previous part using a polling application to periodically check the pushbuttons, this time we need to implement an Interrupt Service Routine (ISR) to check the status of the pushbuttons.

In the assembly file ISRs.s, we added two interrupt flags. The HPS_TIM0_ISR was as given in the manual to clear tim0 and assert the flag. The other one, FPGA_PB_KEYS_ISR was implemented in a similar way to TIM0. It would read the data of the pushbutton that was pressed, assert the flag, and clear the edgecap to reset the interrupt. In the header file of this part, we also declared the two new functions, hps_tim0_int_flag and pb_int_flag.

Now we are ready to integrate them together in the C file to actually create the new stopwatch. This time we only need one timer as we no longer need tim1 to do the polling. The first part inside the loop is again about configuring the counting features. The stopwatch would run whenever the tim0 interrupt occurs, and the interrupt status is cleared before the timer is start. The second part is checking the interrupt of the pushbutton within the iteration. We first check if there is an interrupt from the pushbutton, and if there is an interrupt, we would move on to decide which key is pressed and do the corresponding operation to the stopwatch. Finally we need to set the interrupt flag to zero.

5 Possible Improvement

We noticed that there is always an infinite loop in the main program ‘while(1)’, which means that the program will be running non-stop. It might be better if we could design it to be stop at some point. For example, for the stopwatch programs, we could have another condition to stop the program when the timer reaches 60 minutes.