ECSE 324 Lab 3 Report

Yuhang Zhang 260787441

Pengnan Fan 260768510

Content

Basic I/O

Slider switches

In this section, we learned how to read the value of each slider switch from a given location in memory. The value of each slider switch is automatically placed in memory at address 0xFF200040. We wrote a subroutine to read the data from that address and return it in register R0. The approach to this section involved typing in the code that was provided to us.

Other than some troubles differentiating underscores from periods, we had no difficulties with this section. We don't know of any ways to improve this section.

LEDs

In this section, we displayed the state of each switch on the LEDs. To do this, we used two subroutines: one to read the values of all the switches from the slider memory location, and another to write those values to the LED memory location. The approach to this section was different from the previous section, because the code was not provided. We mirrored the structure of slider_switches.h in LEDs.h, adding a second function declaration that accepts an integer as an argument. Knowing that the argument would be passed through in R0 according to convention, we were then able to write the assembly code for each subroutine. In hindsight, the challenges faced in this section were not complicated. We struggled for some time before realizing that the integer passed in to write_LEDs_ASM gets passed in through R0. We also had difficulty discovering the format that our assembly source file needed to take.

Other than some condensed PUSH and POP statements, there is not much in these subroutines that could be improved. What could be improved is our programming process. We spent too much time staring at a screen without knowing what to do, when we should have been iteratively testing our code.

HEX Displays

In this section, we wrote subroutines that manipulate the 7-segment displays. The HEX_clear_ASM function turns off all of the segments of the specified display(s). The HEX_flood_ASM function turns on all of the segments of the specified display(s). The HEX_write_ASM function displays the specified hexadecimal digit on the specified display(s). Each subroutine creates an 8-bit block of values, then all the subroutines make use of the same block of code to display that value to the correct 7-segment display.

This part, to be honest, is the most difficult part. What we came up was to write a sub function, LOAD_MULTIPLE_VAR to handle for HEX_clear_ASM HEX_flood_ASM and HEX_write_ASM.

In the call functions, we pass signal variables into the R1 register. For the clear function, the R1 is stored as all 0, while for the flood function, the R1 is stored as all 1, in the write function, the passed in variable checked the hex number according to the variable passed in.

Pushbuttons

In this section, we wrote subroutines that manipulate the memory associated with the push keys. The read functions return the pushbuttons that are pressed, the read edge-capture functions return the value of the edge-capture memory location, the clear edge-capture function clears the edge-capture memory location to zero, and the interrupt functions disable or enable interrupts. These functions were quite similar to the LED and slider switch functions implemented above, so our approach was modeled off of those subroutines. Each subroutine involves accessing the appropriate memory address, reading or writing a value, then returning it if applicable. Minimal challenges were faced in writing this subroutine. After reading the appropriate sections of the DE1-SOC manual and reviewing our previously-written subroutines, we wrote the subroutines correctly with minimal debugging. These subroutines are relatively simple, so there is not much improvement to be made.

Polling based stopwatch

There are two types of stopwatches required to implement in this lab, which are polling based and interrupt based. In this section, we will introduce the polling based stopwatch. There is a subroutine implemented by us: HPS_TIM_config_ASM. This method takes a pointer of construct, which contains information about timer, timeout, and control bits. These values will be passed to the timer memory corresponding to the predefined timer type. The second one is used to read the interruption flag, Once the timer is set up, it will automatically run until timed out. Therefore, two timers, a 100 ms timer and a 50 ms timer, are used as time counter and push button listener. For the 100 ms timer, every time when it timed out, the time counter will be increased by 1 and used to update LED display accordingly. And for the 50 ms timer, every time when it timed out will result to check the status of push buttons. If any of them is pressed, the stopwatch will stop/restart/resume accordingly.

In general, it is not very hard for us to implement this subroutine. But still it takes us some time to read the documents to figure out the mechanism of timers. Also, it takes us extra care when read/write correct values to the control bits without changing the rest of the memory. We believe this subroutine is the best work we can make and there is not much improvement to be made.

Interruption based stopwatch

Interrupt based stopwatch works in a different way: it updates the stopwatch by monitoring the interruption bit of a timer. For each timer, when it is timed out, an interruption flag will be marked in the memory corresponding to the timer. Therefore, in order to access and reset the interruption flag, we implemented two subroutines: HPS_TIM_read_INT_ASM and HPS_TIM_clear_INT_ASM. Both of them will take an one-hot encoded value indicating a specific timer and read (HPS_TIM_read_INT_ASM) or clear (HPS_TIM_read_INT_ASM) the interruption flag. We also applied similar codes for the push buttons, which will raise different flags corresponding to four push buttons. Another difference between poll based and interruption based stopwatch is for the latter, only one timer is used. Since both timer and push buttons are monitored by interruptions, therefore we can save one timer. As for this task, we just met a minor difficulty, which is about reset the flags. We solved this issue by carefully masked out other bits and only write one bit to the corresponding address.
In general, we believe this is the best we can and there is no need to improve.