ECSE 324 - Lab #4: Basic I/O, Timers, and Interrupts

Alex Hale (260672475) and Tre Mansdoerfer (260680162)

November 10, 2017

Section 1 - Creating a Project in the Altera Monitor Program

This section taught us how to properly organize our assembly source files. This is important not only because it was essential to ensure that the drivers provided later in the lab compiled properly, but also because it made our source code easier to read and manipulate.

Our approach to this section was to read the instructions, then do the things that the instructions told us to do.

This section of the lab came with the massive challenge of deciding to stop procrastinating and start working on this lab. Otherwise, it was straightforward.

This section of the lab could be improved by having a better debugging and compiling software.

Section 2 - Basic I/O

From here on, the real challenge began!

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.

Since we were given no assembly code, and since this program was much more complex than any of the previous I/O tasks, we didn't have a defined approach to follow. As a result, we wrote and re-wrote our code many times as we made mistakes and corrected them. The procedure our code uses is as follows:

1) Create an 8-bit string of the required values (1s, 0s, or a hexadecimal digit)
2) Load that string into one byte of a full word, at a specific offset of bytes
3) Taking one bit of the input string at a time, check if the value should be displayed to the corresponding 7-segment display (i.e. is that 7-segment display in the input list?)
4) If this display is on the list, clear the appropriate byte in memory using an AND statement, then enter the prepared byte into that same memory with an OR statement.
5) At the end of each loop, increment counters and shift pre-prepared byte strings using the ROR instruction.

This section presented the largest challenge in this lab. Our most difficult hurdle was the realization that using the STRB and LDRB instructions *do not work* to load and store individual bytes. Instead, those instructions overwrite the rest of the word. It took us a long time to realize that AND and OR instructions are the most effective ways to implement byte operations. This realization also helped us in other parts of the lab, where we used the same AND and OR techniques to separate bits from the rest of their binary string.

After a lot of debugging, we think that there aren't many improvements we could make to this section. This is especially true because we reused the same loop structure for all three subroutines, making our file much shorter. One inefficiency in our strategy is that we entirely clear the byte before writing to it regardless of the operation requested, meaning that for the clear operation, we end up clearing the same byte twice.

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.

Section 3 - Polling Stopwatch

Assembly Code

The timer program in assembly creates a basic timer, with subroutines of configure, clear, and read. All of these subroutines share a common approach, in which there is an outer loop counter examining what timer we should be looking at, and a short script identifying this counter's location in memory.

We'll initially look at the configure subroutine, since it served as a basis for the other two subroutines that followed. The configure subroutine takes in a pointer, since there are multiple inputs for this subroutine. As a result, we coded an initial part for the subroutine, that placed the pointer into a register, while also clearing everything ahead of it. This is also where we pushed the needed registers for the subroutine and created the outer counter.

The next segment of this subroutine is indicating if we need to configure this timer and what timer we would be configuring. To do so, there is a compare statement, evaluating if the specific timer needs to be configured. If this is true, it is passed on to the next stage and evaluated for what location in memory that timer is located. The major issue for this section was figuring out how to find what timer needs to be configured, through trial and error we realized it was simply just a 1 bit shift to the right for each individual timer.

The final segment of the configuration subroutine is storing all of the parameters into the appropriate timer memory position. This was relatively simple compared to the previous part.

The read subroutine borrowed from the first two parts of configuration, but required fewer registers to be pushed initially. Outside of the similar first two parts, the final part pulls the correct s-bit from memory and loads it into right-most bit of R0. The register can now be read outside of the subroutine.

The final subroutine for timers uses similar initial parts as the first two subroutines, then clears the values in selected timers. Again, this part of the code was fairly simple, since it borrowed extensively from the configuration code.

C Code (main)

The main c code developed configures 3 timers initially, based off of the configuration subroutine. The firs timer represents the actual 'timer' for the board. The second timer scans the push-buttons and sees if they have been activated. If the timers are configured and timer start is

activated, then the timer begins to run itself. A millisecond, second, and minute counters are created, which are updated each time any of these values hit their respective limit. The appropriate values for the millisecond, second, and minute counters are written into the hex displays of the FPGA board.

The second part of the code looks at the inputted pushbutton values through timer 1. The code evaluates if any of the push buttons are being pressed and how to react to any of these push buttons being used.

The C code was fairly straightforward to write as well. The latter part (pushbuttons) was simple and intuitive. The only struggles would have come from the actual structure of the code itself and translating thoughts and pseudocode into C.

Section 4 - Interrupt Stopwatch

This section was very similar to the last section. In the previous section, a separate timer was polled to determine when to check whether or not a button had been pressed. In this section, a button being *released* fires an interrupt which starts, stops, or resets the timer. The only functional difference between the two timers is that the polling timer acts as soon as button is pressed, while the interrupt timer acts once a button is released. Most of the code was provided to implement the interrupts, so all we had to write was the ISR.s file and the addition in main.c. In the ISR.s file, spaces in memory were set apart for flags for each of the timers, and simple interrupt service routines that handled timer selection were written. In main.c, the structure was mostly copied from Section 3, requiring only the replacement of the timer poller with an if statement that watches for the interrupt flag to be raised.

There were few challenges faced with this subroutine because it was so similar to the previous section. After reading the DE1-SOC computer manual and installing the provided code for handling interrupts, we experienced few setbacks en route to a successful timer.

Since most of the code was provided, we don't have much in the way of potential areas of improvement. One improvement that could be made in main.c is to allow the user to user one of the slider switches to switch between the polling stopwatch and the interrupt stopwatch. This would not only speed up the demo process, but it would also reuse more code in main.c, making that file shorter.