ECSE 324 Lab 3 Report Group 11
Authors: Aleksas Murauskas, 260718389 and Brendan Kellam, 260718806

Slider Switches and LEDs (Basic I/O):

To begin, we started by implementing the slider switches driver. This was relatively simple to understand, requiring a single “read_slider_switches_ASM” subroutine. The subroutine returns the memory contained at the address 0xFF200040 (Data register in the slider switch parallel port).

The LED driver was very similar to the switches driver (included a “read_LEDs_ASM” that was virtually identical to “read_slider_switches_ASM”) with the added subroutine “write_LEDs_ASM”. This mutating subroutine simply took a integer passed through R0 and stores it at LED_BASE (0xFF200000).

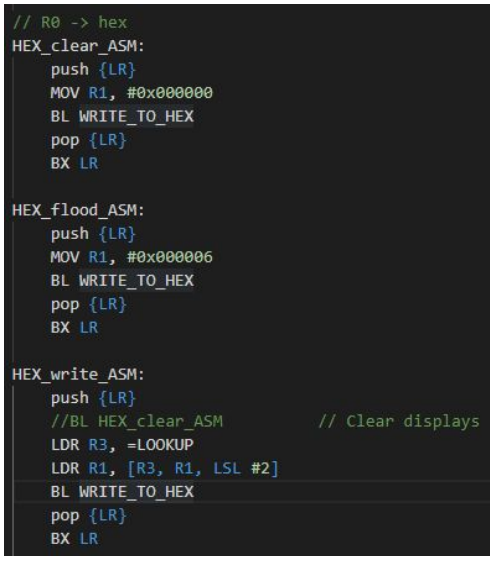
Slightly more advanced: Drivers for HEX displays and push-buttons:

Hex displays:

For this section, we were required to write the following three sub-routines:

- HEX_clear_ASM(HEX_t hex) : would clear all addressed displays in param “hex”
- HEX_flood_ASM(HEX_t hex) : would write 1s to all addressed displays in param “hex”
- HEX_write_ASM(HEX_t hex, char val) : would write a character (0-F) to all addressed displays in param “hex”

Before starting development, we noted that clear and flood subroutines where simplified forms of write. As such, we started by creating a fourth subroutine, WRITE_TO_HEX, that would not be callable from C. We felt this approach would be more modular and elegant, following the DRY principle. Figure 1 depicts the layout of this approach: clear, flood and write MOV the data they need to write into R1. HEX_write_ASM used a lookup table to convert the requested char into the 7-segment representation. WRITE_TO_HEX is where we ran into the most difficulty. We approached writing to the 5 HEX displays by iteratively addressing each one (if the corresponding bit within R1 was set). By doing a single loop, we found we had to add extra logic to determine if we were addressing the first HEX data register (0xFF200020) or the second (0xFF200030). We had allot of conditional instructions (like STRGE, STRLT, LSLT, LSGE etc..) for handling the two cases. In retrospect, it would have likely been easier to swap the HEX data register address when the iteration count surpassed 3. This would remove all the conditional instructions.



```
// R0 -> hex
HEX_clear_ASM:
    push {LR}
    MOV R1, #0x000000
    BL WRITE_TO_HEX
    pop {LR}
    BX LR

HEX_flood_ASM:
    push {LR}
    MOV R1, #0x000006
    BL WRITE_TO_HEX
    pop {LR}
    BX LR

HEX_write_ASM:
    push {LR}
    //BL HEX_clear_ASM          // Clear displays
    LDR R3, =LOOKUP
    LDR R1, [R3, R1, LSL #2]
    BL WRITE_TO_HEX
    pop {LR}
    BX LR
```

Figure 1: Routine layout for HEX display

We also had some issues applying the value: we had attempted to write individual bytes into the Hex register (STRB) but couldn't get that working. Instead, we settled on using a bit mask to select the set of 7 bits within the Data register we wanted to overwrite. This worked well enough, but also ran into some edge case errors that were tricky to debug. For example: if HEX0 is set with a value, writing to HEX displays 1-3 (the other 3 displays in the same data register as HEX0) would overwrite the memory in HEX0. This was due to a unknown issue with our bit mask that we were unfortunately unable to find.

Push Buttons:

The push button spec required us to write several read/write subroutines for various push-button functions. This was a relatively painless process, requiring only simple operations on memory.

Putting it together:

This sections required us to modify the main.c file to create a basic IO program, utilizing the pushbuttons, slider switches and hex displays. The program is simply a infinite loop that initially grabs the state of the slider switches via a call to "read_slider_switches_ASM()". By bitwise anding the returned integer with 0xF, we can get the states of the lower 4 switches. We then simply called "HEX_write_ASM" with the first parameter being the return value from "read_PB_data_ASM" and the second parameter being the state of the lower 4 switches.

This integration of all parts brought up a fair amount of errors in our drivers that were unknown to us until that point. Since we knew the expected functionality, we could go back into each one of our drivers and debug by setting breakpoints and stepping through the disassembly.

Timers:

This tasks required us to create a stopwatch that utilizes the hex displays to show the current time. We implemented three subroutines as required in the spec:

1. HPS_TIM_config_ASM
2. HPS_TIME_read_INT_ASM
3. HPS_TIM_clear_INT_SAM

While performing this task, we appreciated the usefulness associated with passing a pointer to a struct over pushing many parameters to a stack. It made the configuration subroutine quite clean as compared to the alternative of popping of the stack into a set of registers. In the C program, on initializing, all variables corresponding to the milliseconds, seconds and minutes are set to 0. The program then enters a while loop, continuously polling the push buttons to see if the timer should count, pause or reset. We found this section to be the most difficult, requiring the most development time by far. Understanding the DE1-SOC timers from the manual became complex and slightly convoluted. More prerequisite knowledge on our part would have likely made things smoother.

Interrupts:

Unfortunately, due to many issues we faced in the early sections of the lab, along with midterms in other classes, we were unable to implement the interrupt portion. We are planning to go back and implement this section to ensure we have the requisite knowledge to succeed in the proceeding labs.