Samuel Proulx November 8th, 2021

Lab 2 Report

Part 1

Part 1.1 and 1.2 were very similar with 1.2 having more I/O devices. Part 1.2 had to take switches input to light up the corresponding LEDs (all 10 of them) and use the last 4 switches to display the corresponding number to all the pushbuttons that recorded a falling edge.

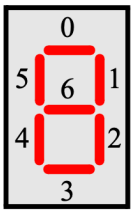
Interacting with the switches and the LEDs was straightforward. The problem of writing to HEX displays was divided into two smaller problems: how to know which display must be selected and what value should we pass to the display based on the hexadecimal number taken as an input. To find the displays that we want to use from the sum of the indices taken as a parameter, I used a loop to decompose the sum into powers of 2. If the sum minus the current factor of 2 is smaller than 0, skip this iteration. If the sum minus the current factor of 2 is greater or equal to 0, we must print to the corresponding display and decrement the sum of indices. After 2 iterations, it was also necessary, since I started from HEX5 and going to HEX0, to change the display register so that I would print to the correct register. The next problem was printing the right number to the display's corresponding byte since the display registers will light up the corresponding segments based on a one-hot encoding scheme. Therefore, from a number in hexadecimal that we want to display (ex.: 0xa or 10), we must determine that all segments minus segment 3 must be lit up. Since there is no formula that can be used to make a relation between the number to display and the segments to light up, I used a lookup table (i.e. a 1D array) to hold the sum of the segment indices corresponding to the number to display. Then, all that is needed to print a number is add the number to display to the base address of the array to find the segment representation of that number and then store it in the display register. Another problem I encountered is how to print to a certain byte of the display register without generating a warning about not using standard word-aligned memory operations. To do this, I used a mask to combine the value currently in the display register and a the inverse of that mask to get the new value at a certain display. For example, to display at HEX2, the mask would look like *0xff00ffff* and for the input *0x00ff000000*. Then, I can do a logical AND between the masks and the corresponding registers and a logical OR to combine the results and then a normal store can be used to write to the display register. I had to rotate the mask by a byte at each iteration of the loop.

```
write_loop:
    @ compare sum with current index
    CMP R4, R5
    BLT skip_write_store
    @ save CPSR state for later
    MRS R11, APSR

    SUB R4, R5
    LDR R9, [R6]
    AND R9, R8
    MVN R10, R8
    AND R10, R3
    ORR R10, R9

    STR R10, [R6]

skip_write_store:
    LSR R5, #1 @ divide by 2
    ADD R7, #1 @ i++
    CMP R7, #2 @ after 2 iterations, we go to HEX0 register
    LDREQ R6, =HEX0_MEMORY
    ROR R8, #8 @ rotate mask a byte to the right
    ROR R3, #8 @ rotate value to display
    CMP R5, #1
    BLT write_return
    @ saved CPSR state from first compare in clear_loop
    MSR APSR, R11
    BGE write_loop
```



Segments

The last I/O device was the pushbuttons. I needed to record a falling edge when the button was released. To do it, I could easily find if an edge was detected based on the *edge_cap* register and the value of the button could be found from the current value of the pushbuttons. If the value of the pushbuttons is 0 but the *edge_cap* register indicates an edge, then the edge must necessarily have been a falling edge.

The weak points of my program are the *HEX_write_ASM* subroutine which is quite lengthy and the size of the program (632 bytes). Also, *HEX_clean_ASM* and *HEX_write_ASM* share a lot of similar code. If they took the encoding for the HEX display instead of the number to display, the clean subroutine could use the write subroutine and substantially reduce the size of the program.

Part 2

Part 2.1 simply required counting from 0x0 to 0xf in increments of 1 second and starting over. To do it, I used a loop that continuously polled the *F* bit. When the *F* bit is 1, the counter is incremented by 1 as long as it is less than 0x10, in which case we reset the counter before printing it to the *HEX0* display.

Part 2.2 was a little more complicated since we had to count in milliseconds, in seconds, and in minutes. The idea is the same, except that the counter runs 100x faster, since the least significant millisecond is omitted, counting it is pointless. Since we also want to count in base 10, I used a loop to divide a hex number in tens and units. The base conversion algorithms loops through the number, subtracting 10 while it's possible. When it can no longer subtract 10, it returns the amount of time it subtracted 10 which is the tens and it returns the reminders that represents the units.

```
@ returns the decimal reprensentation of a number (tens and units)
@ input R0: the number
@ return
@ R0: tens
@ R1: units
base_conversion_ASM:
    PUSH {R4}
    MOV R4, #0
add_tens_loop:
    @ if >= 10 (0xa), then add to tens
    CMP R0, #0xa
    ADDGE R4, #1
    SUBGE R0, #0xa
    BGE add_tens_loop
    @ if < 10, this is the units
    MOV R1, R0
    MOV R0, R4

    POP {R4}
    BX LR
```

The most challenging element was figuring out the load value and the prescaler. To do it, I picked the biggest prescaler value that would give an integer load value. For part 2.1, since a 1Hz frequency is desired, the clock frequency must be divided by 200M. The biggest possible prescaler value that divides 200M is 255, or *0xfe*. This leads to a load value of *0xbebc2*. For part 2.2, a 100Hz frequency is desired. I divided the maximum prescaler to the second biggest base of 2, which is 128, or 127 since 1 is added to the prescaler value. My prescaler value was thus *0x7f*, and that lead to a load value of *0x3d09*.

For a possible improvement, using a *STRB* instruction could reduce the number of instructions by 5 instructions, speeding up every *HEX_write*, *HEX_clear*, or *HEX_flood* quite a bit. I didn't use *STRB* since a warning was generated and I wanted to avoid it. It seems that the DE1-SoC supports byte-aligned loads and stores, so it would not be a problem on this hardware.

Part 3

For part 3, I used the provided code to deal with interrupts. I added the missing part. The most interesting part is the *IDLE* section, which is the most similar to the *LOOP* section in part 2.2. It does essentially the same thing. It increments the counter and writes to the HEX displays when the timer is enable (i.e. when *PB_int_flag*) is asserted. To do it, I modified the *KEY_ISR* to set the *PB_int_flag* to 1 when PB0 is pressed and released and 0 otherwise. The *KEY_ISR* also sets the timer to E bit to 1 to enable or disable the counter.

The biggest challenge with part 3, although most of the code was provided and the structure to deal with interrupts was mostly done, was to figure out the logic of how to respond to timer interrupts and pushbuttons interrupts. As mentioned previously, the *KEY_ISR* handles the timer control (enable, disable) and the *IDLE* loop increments the counter value that is displayed, so it handles the timer interrupts similarly to part 2.2 with the exception that there is now *ARM_TIM_ISR* to clear the interrupt and write to the *tim_int_flag*.

The code could be simplified by making sure the control register of the timer is modified correctly when we want to change the E bit. This is not a problem if we assume that each time we start the timer, we will stop it next. In fact, I only add or subtract 1 to the control register value to change it, which is error prone if we don't check the current value of the E bit.