**Description, approach, challenge, and improvement**

**Slider switches and LED program**

In this program, we seek to control the LED on the FPGA board with the slider switches. To do this, we must write driver files in assembly that read the inputs from their associated memory addresses. The main processing code is written in C, so we must also include header files that tell the C program to look in external assembly files for the functions called by main.c. In the driver subroutine that reads the switches, we first load a register with the memory address found in the reference manual. Then, we load R0 with the content in that address, thereby returning the “value” the slider switches. Writing to the LED is similar, except that we must write to the memory instead of reading form it. We declare the subroutines “global” to make them visible to the C program.

In the C program, we have a loop that reads the value of the slider-switches. The code directly passes that value to the write\_LEDs\_ASM function.

The header files are simply the signatures of the assembly function in the syntax of C.

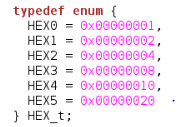
An early challenge we had was forgetting to declare the subroutines global for EACH of the subroutines.

**Basic I/O**

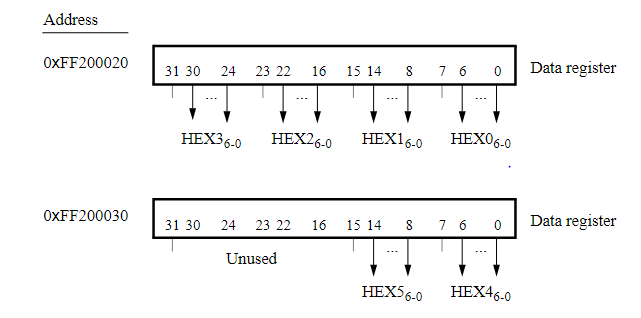
This program operates the six seven-segment displays in the following manner:

* The two HEX displays to the left are always “flooded” (i.e. all segments are on).
* The state of the last four slider switches SW3-SW0 will be used to set the value of a number from 0-15, according to its binary number encoding. (ex. “1111” causes “F” to be set).
* The number set by the switches is displayed on the HEX display when the corresponding pushbutton is pressed. For example, pressing the right-most button activates the right-most HEX display. The HEX display remains on even when the pushbutton is no longer pressed. When the button has not been pressed, the HEX display is off.
* Asserting slider switch SW9 “clears” the four right-most HEX displays

To implement this program, we had to first find a way to use the HEX display to display characters 0-9 and A-F. We created two files, HEX\_displays.s and HEX\_displays.h. The second file is a header file that defined an enumeration with values HEX\_0, HEX1…HEX5. Each value is represented by a one-hot encoding at the corresponding position.

These enumerations will be used as inputs to the subroutines HEX\_clear\_ASM, HEX\_flood\_ASM, and HEX\_write\_ASM. They tell the subroutine which HEX display to clear/flood/write. Using one-hot encoding to represent HEX displays allows us to OR multiple HEX displays, pass the resulting value to the subroutines, and clear/flood/write to multiple displays.

The files HEX\_displays.s implements the clear, flood, and write functions in assembly.

To implement the “clear” subroutine, one first need to determine from the input which HEX display(s) needs to be manipulated. Then, the correct memory address to write to must be chosen. Then, the correct bits in the memory address must be all set to zero. The bit locations and the address for HEX displays are shown below:

Since the input is a combination of one-hot encodings, and since action on multiple register is required, we used a loop in the subroutine to iterate through the different bits of the input. If the bit is 1, we clear the corresponding HEX display. We used simple branching to determine the correct memory address. We noticed that each time we are required to clear 7 bits, so we stored the number 127 (“1111111” in binary) in register R1 as a mask. To clear HEX0, for example, we simply “bit clear” the data register value with the mask and write the value back. We found that a simple method to determine the correct bits to manipulate for subsequent HEX displays involved rotating the mask 8 bits to the left for each subsequent register. Therefore, after each HEX display is looped through, we rotate the mask.

The code for the “flood” subroutine, designed to light all segments of the display, was identical to the “clear” subroutine, except we “bitwise or” the data register value with the mask, instead of “bitwise clear.”

The “HEX\_write\_ASM” subroutine takes in an integer in the range of 0-15 and displays the hexadecimal character corresponding to it. This subroutine assumes the display has already been cleared. We replace the mask of 127 with the appropriate seven-segment configuration of each integer. Specifically, we compare the input to each of the 16 integers and branches to one of the sixteen “display” branches. We then set the configured mask and used “bitwise or” to change the data register value.

The largest challenge in this task is finding an elegant way to determine indices of manipulated bits based on the input. The solution of rotations is elegant.

The push-button and switch drivers were easier to implement. All involved simple reading and writing from data registers. See code for comment.

For the final C program, we clear all HEX displays at the start and enter a loop. In the loop, we detect if the SW\_9 is on and clear all HEX displays if so. We flood the two left-most displays, write to LEDs, detect the configuration of switches and pushbuttons, and pass the configuration to HEX\_write\_ASM.

**Polling-based stopwatch**

For this portion of the lab, we had to consolidate everything that we did previously and create a polling based stopwatch. In short, this had to utilize the previous pushbutton, display and LED drivers, along with a new timer driver, to implement a simple stopwatch functionality, with one timer tracking the actual time and another one polling the pushbuttons for inputs.

For the timer drivers, we followed the instructions and implemented 3 functions: one to initialize a timer, one to clear the bits and another one to read the values. This was done by following the DE1-SoC computer manual provided as well as the header provided. We began by using the loop and continue subroutines to find out in which bits the zero values in the struct pointer were passed, which we knew could only go up to 4 bits. We would only skip to the config subroutine if there was a 0 passed or if we checked all 4 bits. This was to ensure that nothing would be overwritten. In config, we first disabled the timer by comparing the enable bit in the control register with a string of 1s and with a LSB of 0, effectively setting it to 0. We then initialized the load value to the value passed in the struct pointer. We multiplied that value by 25 for the 25 MHz timers, and then increased that to 100 if we found out that it was the 100Mhz timers, as per the manual. We then setup the load bit, the interrupt bit and the enable bit accordingly. Reading the timer was much simpler, and we simply traversed the timer bases to find which one to read, after which we just returned it in R0. Similarly, clear simply involved writing in a 0 at each of those steps.

Finally, we implemented 2 simple timers with this driver, with a 1 second and a 5ms timeout respectively. The actual stopwatch updated at every 10 milliseconds and rewrote the displays then too. We could maybe have improved the logic by only rewriting if there was a change to the value, but this was the most straightforward and error free way to implement it. This functionality was governed by the polling of the pushbuttons, where we either started, stopped or reset the timer depending on the pushbuttons pressed. This could have been improved by using the last pushbutton to implement increased functionality, such as loading in a time to start from using the switches from the previous section, perhaps even counting backwards from there. However, the conversion of the switch values into time values would have been daunting to even consider, and we did not have time to think about it.