ECSE 324: Lab 3

Theodore Janson, 260868223

Edin Huskic, 260795559

March 13, 2020

I. Basic I/O

A. Problem and Approach

The code for slider switches was provided. It required a subroutine to read slider switches by reading the value (on or off) at the memory location for a certain slider switch and loading R0. The LED program was similar, but required both read and write instructions. The read subroutine was implemented the same way as for the slider switches. The write instruction, however, required storing the value contained in R0 into the address of the LED.

B. Challenges Faced

None

C. Possible Improvements

None

II. HEX Displays

A. Problem and Approach

Three subroutines needed to be implemented: HEX_clear_ASM(HEX_t hex), HEX_write_ASM(HEX_t hex), and HEX_flood_ASM(HEX_t, char val).

The first clear subroutine required storing a bit sequence of 0s at the address of corresponding to the display being cleared. The first step consisted in saving all the contents of registers R1-R11 and LR onto the stack. Two registers were loaded with the addresses corresponding to hex displays 0 to 3 and 4 to 5. Since each register can hold 32 bits and each hex display is represented by 8 bits, two registers are needed. We then loop through each hex display (0-5). During each loop, a bitwise AND is done through a TST instruction between R0 and R3. R3 is initialized to 1 for the first loop and takes on values of 2,4,6,16 for each subsequent loop by left shifting by 1 bit after every loop. These values represent HEX0 to HEX5. By performing a TST, we check if the display to be cleared has the same bit sequence as one of those values - in which case, a clear subroutine is called through a BLNE instruction. The TST is necessary so that the correct parameters are passed to the subroutine for clearing that specific hex display. an implementation specific to that display. To clear the hex, we either store the value of 0 at the memory locations corresponding to displays 0-3 or 4-5. The decision is done by comparing the current loop value (0-5) to 3 and storing accordingly.

Flood was implemented using an identical loop implementation. To flood a display the value of 8 is stored at the memory locations corresponding to displays 0-3 or 4-5. Storing the value 8 was more complicated than storing the value of 0. The hexadecimal 0x0000007F is used because it is the representation of 8 encoded through a 7 segment decoder. To flood a series of hex display, we need to store a sequence of 1s that is 8 * n long where n is the hex number (1-4). During each loop, we need to store the value 7F (11111111) at 8-bit intervals of the 32 bit long address for HEX displays 0-3. This is accomplished by left shifting this value by 8 bits during each loop pass. When the actual flood subroutine is called during the loop pass, we compare the counter to 3 and branch to a specific branch for storing in 0-3,4, or 5. At those branches, an ADD is done with the previous values of the sum of shifted 7Fs and the new shifted 7F. For displays 0-4, we stored the bit sequence of 1s at their corresponding address. For hexes for and 5, hard coded values for the position of 7F as the shifted versions couldn't be used since the bit sequence would be all zeros during those loops. Finally, we branch back to the loop.

The final function to implement was the write. The implementation was pretty similar to the previous two subroutines. The initialization was done similarly, but the displays were cleared by calling the subroutine described in the first section. The implementation of the loop for looping through the hex displays was similar: when the parameter R0 matched the one-hot encoding for the current value of the one-hot encoded value (left shifted number) of the hex-display, a subroutine for writing each hex display is called. This subroutine is initialized by loading a register with the memory address of an array defined at the bottom of the file. This array contains the 7 segment encoded numbers 0-15. Then, a write-loop is entered. We loop from 0-15: during each loop we load a register with the corresponding 7-segment number. If the parameter R1 (value to display) is the same as the value in the counter, we move the 7 corresponding 7 segment value into a register so that it can be stored in the appropriate location. Note that only 1 number between 0-15 will trigger the BEQ to call the store branch. When we enter this branch, we compare the hex to be written (R0) with 3 so that we can branch to the specific storing routines for each hex. To store the write value of Hex 0-3, we store the 7-segment value at the memory location offset by 0-3 for each display. The storing is similar for Hex4 and Hex5.

The main idea for implementing the main function for this part of the lab was to first flood hex 4 and 5 and clear the rest so that they could be written on. An if statement was needed to clear display 0-3 when the number 512 was read from the slider switches (switch 9). To write on displays 0-3, we needed 'if' statement to check if the data read from push buttons corresponded to the display being written on. In these cases, we wrote the value of the slider switches onto the hex display.

B. Challenges Faced

The main challenge was debugging the entire process. We had problems in our loop, as the display would freeze after being written on. The problem proved to be due to forgetting to save the subroutine on the stack when calling nested subroutines. Our method for debugging was to analyze the problem and to check the section of code that would seem to most obviously be problematic.

Debugging the program was very challenging as the debugging breakpoint capabilities of the IDE were not very useful, nor was the disassembly feature. This required going through the code very carefully and to identify bugs by keeping track of loops and parameters being passed to subroutines.

C. Possible Improvements

A possible improvement for flooding the hex displays with 7F would be to rotate the bit sequence, instead of using a left shift for displays 0-3 and hardcoding the values to reset 7F to the LSBs for 4 and 5. This would avoid having separate storing routines for Hex 0-3, 4 and 5.

Similarly, for the write instruction, we could avoid having separate storing instruction by implementing a rotating offset.

III. TIMER

A. Problem and Approach

This section involved HPS timers and the creation of a stopwatch. In order to create a functional stopwatch, one HPS timer was used for milliseconds, seconds, minutes and hours to work. Three functions needed to be implemented in order for the timer to work. An additional timer is used in order to take into account the polling of the pushbuttons. The first function involves configuring the HPS timer in order for the input to be read correctly. This involved interrupting all timers with the help of a register counter and configuring all its parameters, such as the M bit, the interrupt mask bit I and the enable bit E, with subroutines. Once done, all three parameters are combined together and re-added onto the register. The second function involves interrupting the timer. The function checks for the interrupt bit S inside the HPS timer memory location, and reads whether the bit is active or not (either 1 or 0). The third function involves clearing the timer. As given in the De1-SoC instructions manual, reading the content of the End-of-Interrupt register is the equivalent of clearing content inside the HPS timer. The function then iterates through each timer for the reading of this bit. The result is that the display is now filled with zeros. The rest of the timer is implemented in the main function. Since C uses microseconds, it was taken into account when programming each timer. Conditions were implemented in order for timers to display the right numbers, for example switching the numbers of hours by 1 when the timer reaches 60 minutes.

B. Challenges Faced

The biggest challenge in this section was to implement the timers themselves, as implementing the main function was relatively simple once the timers functioned correctly. In order for the timer to work, all the parameters need to be set, which requires a thorough reading of the De1-Soc manual related to the board. Once the exact memory locations and its values were identified, coding the parameters at the right emplacements was difficult.

C. Possible Improvements

Instead of using a counter to iterate through the different timers in the code, there could be a way to implement a change of the parameters for all timers at once, in order to save time and reduce lines of code.

IV. INTERRUPTS

A. Problem and Approach

With the help of the previous section, a stopwatch with interrupts was to be implemented for this section. In order for this to work, the code from the timer and its files are reused, but slightly modified. The polling method is replaced for a method using interrupts. The new main code is changed to include parameters necessary for interrupts to work. In order to implement interrupts, the drivers given were installed into their respective folders inside the project folder. These drivers are responsible for handling interrupt events inside the code, such as the pushbuttons events. Once one of these interrupt events is activated, the processor halts execution in order to process the new information received. The ISR file, used to scan for interrupts, checks for the pushbutton pressed and prints a flag in return if the result is positive. Since the code is halted, this means that we are able to remove the second HPS timer implemented for polling pushbuttons, which free processor time and use. The rest of the code used for the stopwatch mostly remains unchanged, except for the new drivers with the ISR file, and the main function.

B. Challenges Faced

The issue for the section was understanding what subroutines needed to be implemented in the ISR file in order for hardware devices to handle the interrupts as intended. Laboratory documentation greatly helped during this process.

C. Possible Improvements

Since most the code is based on given drivers online and the reused files from the previous section, there is not much room for improvements in this case. Most of the improvements should be concerning the reused code of the previous section.