Lab 3 Report

ECSE 324 Nada Marawan (260720514)

Prof. Davis Imad Dodin (260713381)

Demo Date: 15/03/19 **Due Date: 29/03/19**

2 Basic I/O

As described in the Lab Handout, I/O Hardware Components are fairly simply configured for the board. In essence, we must simply store / retrieve information from the designated "interface addresses" for the desired hardware component. Understanding and leveraging the way information is stored (i.e. the format of the data at these addresses), however, is where lies the bulk of the effort for this lab.

2.1 Slider Switches

The Assembly, Header File and Main C code for interfacing with the board's Slider Switches were all provided for this lab. The interface address is read-only and stores a 10 bit one-hot encoding (in reality a 32 bit number with the first 32 bits unused), wherein a hot bit indicates that the slider-switch with the corresponding index is flipped [1]. The assembly code simply reads this information from the address of the interface data register: `0xFF200040` (and returns it in R0, as the calling convention dictates).

2.2 LEDs

This section required us to implement two assembly subroutines from scratch: `read_LEDs_ASM` and `write_LEDs_ASM`, which read and write data from the LED Interface data register, respectively.

We begin by implementing a header file which indicates to the compiler that an implementation for the methods must be found somewhere in the provided source. Writing the header file requires us to begin thinking about the eventual logic of our assembly subroutine, by the way of writing the parameters and return types to our method. We require no parameters to be input when reading from the LEDs and return an integer, as the interface data address stores information as a 10-bit one-hot encoded integer (as was the case for slider switches). [1]. We, similarly, require an integer which is expected to be a one-hot encoding for which LEDs to be activated, as a parameter for writing LEDs, and allow this method to have a void return type (as no information is required to be returned to the user in this case).

For the `read_LEDs_ASM` assembly subroutine, we simply load in the encoding from the interface address of `0xFF200000` [1] and return it in the Register R0. For the `write_LEDs_ASM` assembly subroutine we take the passed one-hot encoding and write it into the above register.

We finally execute our write subroutine in C by first reading the slider switches using the provided subroutine and passing its output as a parameter into the write subroutine. This effectively turns on the LED lights corresponding to each flipped slider switch.

2.2.1 Improvements

This subroutine can be improved by requiring the write subroutine to return a boolean indicating success and performing a check that the provided one hot encoding is in the bounds [0, 1023], returning a boolean of false 0 if this check does not succeed. In order to use the `bool` type in C, we would need to additionally include the <`stdbool.h`> header - as booleans are not primitive in C.

2.3 Hex Displays

This section requires us to write 3 subroutines: `HEX_clear_ASM`, `HEX_flood_ASM` and `HEX_write_ASM` which are responsible for clearing (setting all segments off), flooding (setting all segments on) and writing characters to the selected screen (passed via a one-hot encoded integer. We note that the interface data addresses are readable and writeable and store a 7-bit one-hot encoding for which segments of the display are to be activated [1].

We implement the header file to declare the 3 required methods. We note that all of their return types are void as the methods are solely responsible for writing to the address. The required parameters are a single integer indicating the one-hot encoding of the selected screen to clear or flood, in the case of the first two methods and two integers indicating first the one-hot encoding of the selected screen to write and then which integer value is to be written on the selected screen. As is stipulated by the assignment guidelines, we write a single hexadecimal character to represent the passed value (i.e. a passed value of '11' in the second argument, will write a 'B' to the selected screen).

Before beginning the implementation of our assembly subroutine, we take an important note that the hexadecimal display interface addresses are divided into two: one for the first three screens, and another for the last two. This means we must begin our subroutines by determining which address to use (i.e. which screen we are attempting to write to / clear / flood.) We do this by continuously performing a right shift on the provided one-hot encoding for screen selection, all the while maintaining a counter to dictate which screen has been specified (translating the one-hot encoding). During this loop, if the counter reaches a value of 6, we know that the provided one-hot encoding is either empty or out of bounds (with respect to the number of screens that are available) and we exit our subroutine.

In the case that our counter holds a value of either 4 or 5, we use the interface address for the second set of seven-segment displays, and decrement our counter by 4 to represent which index within the set the screen is at.

We then construct a mask to apply to the contents of the interface address. We either use a 7-bit chunk of 0's that are shifted by the appropriate amount (based on which screen we are clearing) and then ANDed with the current content of the interface address to *clear* the display, or a 7-bit chunk of 1's that are shifted by the approriate amount and then ORed with the current content of the interface address to *flood* the display. In the case of writing a character to the display, we perform a sequential check to determine which integer was provided as input, and then apply a hard-coded mask to turn on the approriate segments of the display for the selected character.

2.3.1 Improvements

Our code eventually became extremely messy and could be improved, however, due to time constraints this was not a realistic goal. This task was definitely the most difficult of the ones in this lab, however, it was also the one which was most instructive in learning the functionality of the interface addresses.

2.4 Push Buttons

This section was, in bulk, a simple exercise of reading from and writing to the appropriate memory addresses for the FPGA's push buttons. We had to implement several subroutines to achieve the following functionality.

- Return whether or not a provided push button is pushed.
- Return the edgecapture status of a provided push button.

- Clear the edgecapture status of a provided push button.
- Toggle on / off interrupt functionality for a provided push button.

We extend the concepts outlined in the previous section to apply masks to select only information for the provided one-hot encoding for which push button to check. We return / write only this information by use of these masks.

3 Timers

This section aims to implement an accurate stopwatch with the use of HPS timers. The stopwatch must have the ability to stop, start and reset.

The HPS timers allows us to poll the board continuously, with the `timer` parameter dictating the accuracy of these polls.

We then use the timer to update each of the hex displays' values for time. Another timer continuously polls for user input to stop, start or reset the stopwatch. The updating of display values was done with simple modular arithmetic, a la ECSE 202.

4 Interrupts

We again try to implement the stopwatch with the same functionality as described previously, with the use of interrupts instead of polling. Code was provided to do a bulk of the work in the `int_setup` and `address_map_ARM` files. The ISR *Interrupt Service Routines* for the FPGA's Push Button Keys were not not yet implemented - wherein lied the work for this task.

We implement the interrupt flag functionality for the pushbutton keys by calling the `read_PB_edgecap_ASM` subroutine, to set the flag for the value read (a one-hot encoding for Push Buttons 0, 1 and 2).

To implement the actual stopwatch functionality, we refactor the code for the previous section - replacing any polls for the HPS timer with Interrupt Flag Checks.

References

[1] Altera Corporation (2014) *DE1-SoC Computer System with ARM Cortex-A9 Manual.* San Jose, CA: Author.