CSCE 230L – Lab 4: Introduction to I/O

February, 2014

Objectives

- Gain a basic understanding of how memory-mapped I/O works
- Practice writing programs to do simple tasks with I/O

Useful References

Appendix B in the textbook (sections 10)

List of instructions for the NIOS II Processor:
http://www.altera.com/literature/hb/nios2/n2cpu_nii51017.pdf

Information about I/O on the board:
ftp://ftp.altera.com/up/pub/Altera_Material/11.0/Examples/DE1/NiosII_Computer_Systems/DE1_Basic_Computer.pdf

What to Turn In

- Pre-lab questions (due at the beginning of lab; in paper)
    - All questions total to 20 points
- Code and questions from lab (submit on handin)
    - Task 1 (15 points)
    - Task 2 (25 points)
    - Part 2 Question (10 points)
    - Task 3 (30 points)

Make separate code files for each task and answer all the questions in a single text file. For the code, all you need to submit are the assembly files (.s files). Other files from a monitor program project are not needed.

Name (1 point):

Pre-lab (19 points)

1. Read sections 3.1, 3.2, B.9, and B.10.1 from textbook.
2. Read section 2.3, 3, and 3.1 from the DE1 Computer Information PDF.

In a system with memory-mapped I/O, each device has a pre-determined memory location to use. At this memory location are bits used to keep status of, or store settings for the device. If we look up information about our board (link is in useful references section) and go to section 2.3.1, we see information about the LEDs on the board. We can see that the memory locations specified for LEDR and LEDG are 0x10000000, and 0x10000010. Also given is the format for the data stored at that memory location. So if we wanted to turn on LEDR0, LEDR2 and LEDR3, our data would be 0x00001101, and we need to store that data at address 0x10000000. The code example below shows how we would do this. Note that for I/O we use the instructions ldwio and stwio.

```
movia   r2, 0x10000000          /* load address of LEDR into r2 */
movi    r3, 0x00001101          /* load data for LEDR into r3 */
stwio   r3, 0(r2)               /* store the data at the memory location for LEDR */

/*here we use 'movi' to pass a 32 bits long immediate value to a register*/
```

3. What is the memory address for the 7-segment displays parallel port on the board?

4. If we wanted to display "1234" and "5678" on the 7-segment displays. In each case, what would the data register look like? Give answers as hex values.

5. Why is the data for the slider switch read-only?

6. Download *prelab4_1.s* and look at the code. ('.equ' is used to substitute one for another)
    a. Look at the first instruction in MAIN_LOOP, where does the new number come from?

b. In MAIN_LOOP, what does the instruction *stwio r17, 0(r9)* do?

c. What does LOOP2 do?

d. What would we change if we wanted this program to use the red LEDs instead of the green LEDs?

Lab

Part 1

Task 1(15 Points)

From the pre-lab you should now understand that I/O devices are at a specified memory location. Now you get to practice writing a program to use some of the I/O on your board. Let's begin with something easy. **The first task is reading from slider switches, and displaying it on LEDR**. For example, if your SW9 and your SW4 are on, and all others are off, your LEDR 9 and LEDR 4 need to light up, and other red LED are off.

Part 2

The second thing we want to do is make a rotating pattern on the red LEDs, meaning that one LED is on at a time and we cycle through each LED and then start over. Rotate right instructions are useful here, as we can use the instruction to cycle through LEDs.

```
addi    r3, r0, 1       /* r3 has the value of 1 for rotation amount
addi    r2, r0, 2       /* r2 has the value 0x00…010
ror     r2, r2, r3      /* r2 has the value 0x00…001
ror     r2, r2, r3      /* r2 has the value 0x10…000
```

1. Why do we want to use rotate instructions instead of shift instructions? (2 points)

2. To light up the LEDR 9, we need to give value '0x00000200' to LEDR's data register, to light up the LEDR 0, the value is '0x00000001'. After LEDR 0 is lighted up, to rotate light to the right, we want to light up LEDR 9, if we use 'ror' to rotate the bit '1' in the value, we will get '0x10000000', which cannot light up LEDR 9. What can we do to light up LEDR 9 right after LEDR 0? (4 points)

3. If we really wanted to use shift instructions instead of rotate instructions, what condition would our code need to check? (Which LED turns on after the last LED turns off) (4 points)

Since we want the LEDs to rotate slowly enough that we can observe it easily with our eyes, we need some kind of delay in between each rotation. You can use the following code to add a delay in your program.

```
MAIN:       …
            br DELAY
```

```
DELAY:      movia   r10, 1650000        /* initialize counter */
LOOP:       addi    r10, r10, -1        /* decrement counter */
            bne     r10, r0, LOOP       /* stay in loop is counter is not zero */
            br      MAIN                /* delay is finished, branch somewhere else */
```

Task 2(25 points)

Using the delay code above, write a program that cycles through the red LEDs, with a delay before changing which LED is on.

For example, first LEDR9 is on, then a delay, then LEDR9 turns off and LEDR8 turn on, then a delay, and so on. Your program could have the following structure. Remember to light up the first light after turn off the last light (Light up LEDR 9 after turning off LEDR 0)!

```
START:      /* initialize addresses */
            …
NEXT:       /* turn on next LED then goto DELAY */
            …
DELAY:      /* delay for a time, then goto NEXT */
            …
```

Part 3

Now we want to get some experience using the 7-segment displays and the push buttons. What we want to do here is to have the segments display **either** “Add” or “Sub”, with the push button alternating between the two. Looking in the DE1 Computer Information PDF, you should be able to find the address of the push buttons and which bit in the data register corresponds to each button. We will use button **KEY2** for this lab. When the button is not pressed, the segments will display “Add” and when the button is pressed, the segments will display “Sub”. Now we run into the issue of waiting for a button to be pressed. A simple solution is to use a concept called **busy-wait**. Our program will continuously check the status of the push-button’s data registers until the bit for KEY2 changes. Once it does, then we change the word on the 7-segment display. In a high-level view, our program looks as follows:

```
/* assume button is initially not pressed */
status = 0

while(TRUE)
{
Add:    displayAdd()
        while (status == 0) {       /* while KEY2 is not pressed */
                /* update status of KEY2 */
                status = readKEY2value()
        }
Sub:    displaySub()
        while (status == 1) {       /* while KEY2 is pressed */
                /* update status of KEY2 */
                status = read KEY2value()
        }
        /* when we get here, KEY2 is not pressed, so go back to display ADD */
}
```

When we read the data register of the push buttons, we're only concerned with one bit. So how do we check just that one bit? Using logic instructions, we can clear every other bit (make them equal to 0), and then check if the resulting value is equal to zero or not. The following code will demonstrate.

```
/* assume r2 has address for push buttons */
ldwio   r2, 0(r2)               /* load data register of push buttons */
movia   r3, 0x00000004          /* only bit location of KEY2 is a 1 */
and     r2, r2, r3              /* clear all bits but location of bit for KEY2 */
bne     r2, r0, KEY2_PRESSED    /* if r2 != 0, then KEY2 is pressed */
```

Task 3 (30 points)

Using the 7-segment displays and the push buttons, write a program that will display "Add" on the displays when KEY2 is not pressed, and "Sub" when KEY2 is pressed.