Laboratory Exercise 2

Using Logic Instructions with the Nios II Processor

Logic instructions are needed in many embedded applications. Logic instructions are useful for manipulation of bit strings and for dealing with data at the bit level where only a few bits may be of special interest. They are essential in dealing with input/output tasks. In this exercise we will consider some typical uses. We will use the Nios II processor in one of the pre-built computer systems that come with the Monitor Program.

Part I

In this part you will implement a Nios II assembly language program that counts the longest string of 1's in a word of data. For example, if the word of data is 0x103fe00f, then the required result is 9.

Perform the following:

1. Create a new folder to hold your Monitor Program project for this part. Create a file called *part1.s*, and type the assembly language code shown in Figure 1 into this file. This code uses an algorithm involving shift and AND operations to find the required result—make sure that you understand how this works.

2. Make a new Monitor Program project in the folder where you stored the *part1.s* file. Depending on your DE-series board, use the corresponding computer system listed in Table 1.

Board	Computer System
DE0-CV	DE0-CV Computer
DE1-SoC	DE1-SoC Computer
DE2-115	DE2-115 Computer
DE10-Lite	DE10-Lite Computer

Table 1: DE-series board computer systems

3. Compile and load the program. Fix any errors that you encounter (if you mistyped some of the code). Once the program is loaded into memory in the computer, single step through the code to see how the program works.

Part II

Perform the following:

1. Make a new folder and make a copy of the file *part1.s* in that new folder. Give the new file a name such as *part2.s*.

2. In the new file *part2.s*, take the code which calculates the number of consecutive 1's and make it into a subroutine called ONES. Have the subroutine use register r4 to receive the input data and register r2 for returning the result.

3. Add more words in memory starting from the label TEST_NUM. You can add as many words as you like, but include at least 10 words. To terminate the list include the word 0 at the end—check for this 0 entry in your main program to determine when all of the items in the list have been processed.

4. In your main program, call the newly-created subroutine in a loop for every word of data that you placed in memory. Keep track of the longest string of 1's in any of the words, and have this result in register r10 when your program completes execution.

5. Make sure to use breakpoints or single-stepping in the Monitor Program to observe what happens each time the ONES subroutine is called.

```
            .text
            .global  _start
_start:
            ldw      r9, TEST_NUM(r0)     /* Load the data into r9 */

            mov      r10, r0              /* r10 will hold the result */
LOOP:       beq      r9, r0, END          /* Loop until r9 contains no more 1s */
            srli     r11, r9, 0x01        /* Count the 1s by shifting the number and */
            and      r9, r9, r11          /* ANDing it with the shifted result */
            addi     r10, r10, 0x01       /* Increment the counter */
            br       LOOP

END:        br       END                  /* Wait here */

TEST_NUM:   .word    0x3fabedef           /* The number to be tested */
            .end
```

Figure 1: Assembly-language program that counts consecutive ones.

Part III

One might be interested in the longest string of 0's, or even the longest string of alternating 1's and 0's. For example, the binary number 101101010001 has a string of 6 alternating 1's and 0's.

Write a new assembly language program that determines the following:

- Longest string of 1's in a word of data—put the result into register r10
- Longest string of 0's in a word of data—put the result into register r11

- Longest string of alternating 1's and 0's in a word of data—put the result into register r12 (Hint: What happens when an n-bit number is XORed with an n-bit string of alternating 0's and 1's?)

Make each calculation in a separate subroutine called ONES, ZEROS, and ALTERNATE. Call each of these subroutines in the loop that you wrote in Part III, and keep track of the largest result for each calculation, from your list of data.

Part IV

In this part you are to extend your code from Part III so that the results produced are shown on the 7-segment displays on your DE-series board. Display the longest string of 1's (r10) on *HEX*1 − 0, the longest string of 0's (r11) on *HEX*3 − 2, and the longest string of alternating 1's and 0's (r12) on *HEX*5 − 4.

Each result should be displayed as a two-digit decimal number. You may want to use the approach discussed in Part IV of Exercise 1 to convert the numbers in registers r10, r11, and r12 from binary to decimal.

The parallel ports in the computer systems that are connected to the 7-segment displays are memory mapped devices. Figure 2 shows how the display segments are connected to the parallel ports and the ports' respective memory mapped addresses.

To show each of the numbers from 0 to 9 it is necessary to light up the appropriate display segments. For example, to show 0 on *HEX*0 you have to turn on all of the segments except for the middle one (segment 6). Hence, you would store the bit-pattern $(00111111)_2$ into the address corresponding to the *HEX*3 − 0 parallel port to show this result. A subroutine that produces such bit patterns in given in Figure 3.

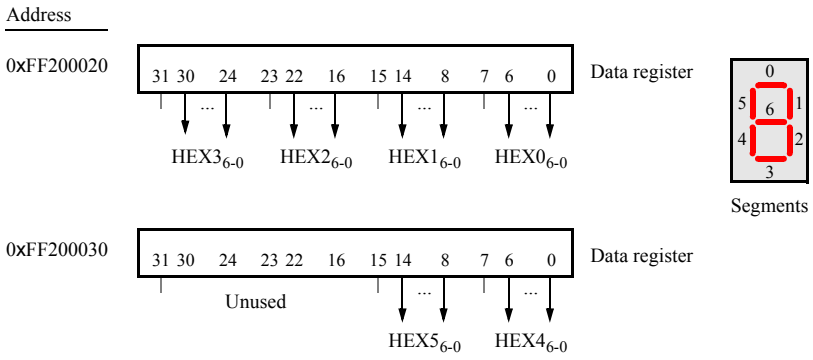

Figure 2: The parallel ports connected to the seven-segment displays, *HEX*5 − 0, of the DE-series board.

An example of code that shows the contents of registers on the 7-segment displays is illustrated in Figure 4. Note that this code uses the DIVIDE subroutine that was discussed in Part IV of Exercise 1. The code in the figure shows only the steps needed for register r10. Extend the code to display all three result registers on the 7-segment displays as described above.

```
/* Subroutine to convert the digits from 0 to 9 to be shown on a HEX display.
 *    Parameters: r4 = the decimal value of the digit to be displayed
 *    Returns: r2 = bit pattern to be written to the HEX display
 */
SEG7_CODE:  movia    r15, BIT_CODES      # starting address of bit codes
            add      r15, r15, r4        # index into the bit codes
            ldb      r2, (r15)           # read the bit code needed for our digit
            ret

BIT_CODES:  .byte    0b00111111, 0b00000110, 0b01011011, 0b01001111, 0b01100110
            .byte    0b01101101, 0b01111101, 0b00000111, 0b01111111, 0b01100111
            .skip    2      # pad with 2 bytes to maintain word alignment
```

Figure 3: A subroutine that produces bit patterns for 7-segment displays.

```
. . .
code for Part III (not shown)
. . .
/* now display r10 on HEX1−0, r11 on HEX3−2 and r12 on HEX5−4 */
DISPLAY:    movia    r8, 0xFF200020
            mov      r4, r10             # display r10 on HEX1−0
            call     DIVIDE              # ones digit will be in r2; tens digit in r3
            mov      r4, r2              # pass ones digit to SEG7_CODE
            call     SEG7_CODE
            mov      r14, r2             # save bit code
            mov      r4, r3              # retrieve tens digit, pass to SEG7_CODE
            call     SEG7_CODE
            slli     r2, r2, 8
            or       r14, r14, r2        # bit code for tens | bit code for ones
            . . .
            code for r11 (not shown)
            . . .
            stw      r14, (r8)           # display the results from r10 and r11

            code for r12 (not shown)
            . . .
```

Figure 4: A code fragment for showing registers in decimal on 7-segment displays.