# Laboratory Exercise 2

Logic Instructions and Memory-Mapped Output

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 ARM* processor in the DE1-SoC Computer. As in Lab 1, you may want to develop, and debug, your ARM programs on your home computer using CPUlator. However, marking will be done by running your programs in the lab at the University using the Monitor Program and the DE1-SoC board.

**Part I**

In this part you will implement an ARM assembly language program that counts the longest string of 1's in a word (32 bits) 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 store your solution for this part of the exercise. 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. Use the DE1-SoC Computer for your project. (For CPUlator, no project is used—you would just load the file *part1.s* into CPUlator, as you learned in Lab 1).
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 your DE1-SoC board (or the *simulated* memory in CPUlator), single step through the code to observe the program's operation.

**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 R1 to receive the input data and register R0 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 R5 when your program completes execution.
5. Make sure to use breakpoints and/or single-stepping in the Monitor Program (or in CPUlator, if debugging on your home computer without a DE1-SoC board) to observe what happens each time the ONES subroutine is called.

```
/* Program that counts consecutive 1's */

          .text                   // executable code follows
          .global _start
_start:
          MOV     R1, #TEST_NUM   // load the data word ...
          LDR     R1, [R1]        // into R1

          MOV     R0, #0          // R0 will hold the result
LOOP:     CMP     R1, #0          // loop until the data contains no more 1's
          BEQ     END
          LSR     R2, R1, #1      // perform SHIFT, followed by AND
          AND     R1, R1, R2
          ADD     R0, #1          // count the string length so far
          B       LOOP

END:      B       END

TEST_NUM: .word   0x103fe00f

          .end
```

Figure 1: Assembly-language program that finds the largest string of 1's.

**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 R5
- Longest string of 0's in a word of data—put the result into register R6
- Longest string of alternating 1's and 0's in a word of data—put the result into register R7 (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 (R5) on *HEX*$1-0$, the longest string of 0's (R6) on *HEX*$3-2$, and the longest string of alternating 1's and 0's (R7) on *HEX*$5-4$.

Each result should be displayed as a two-digit decimal number. Use the approach that you learned in previous lab exercises to convert the numbers in registers R5, R6, and R7 from binary to decimal.

The parallel port in the DE1-SoC Computer connected to the 7-segment displays *HEX*$3-0$ is memory mapped at the address 0xFF200020, and the port connected to *HEX*$5-4$ is at the address 0xFF200030. Figure 2 shows how the display segments are connected to the parallel port bits. 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 0xFF200020 to show this result. A subroutine that produces such bit patterns is given in Figure 3.

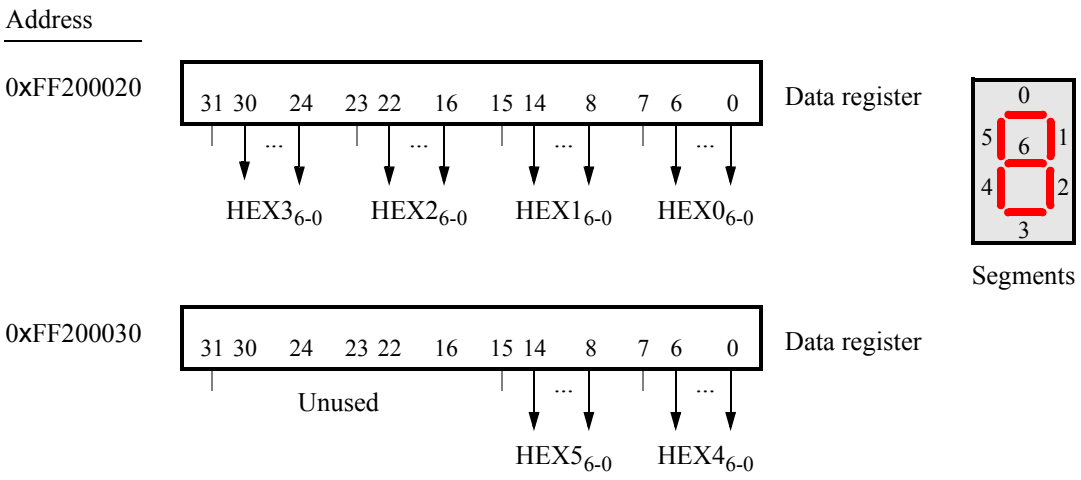


Figure 2: The parallel ports connected to the seven-segment displays *HEX*$5-0$.

An example of code that shows the content of registers on the 7-segment displays is illustrated in Figure 4. Note that this code uses the DIVIDE subroutine that was introduced in Lab Exercise 1. The code in the figure shows only the steps needed for register R5. You will need to extend the code to display all three 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: R0 = the decimal value of the digit to be displayed
 *    Returns: R0 = bit patterm to be written to the HEX display
 */

SEG7_CODE:  MOV     R1, #BIT_CODES
            ADD     R1, R0          // index into the BIT_CODES "array"
            LDRB    R0, [R1]        // load the bit pattern (to be returned)
            MOV     PC, LR

BIT_CODES:  .byte   0b00111111, 0b00000110, 0b01011011, 0b01001111, 0b01100110
            .byte   0b01101101, 0b01111101, 0b00000111, 0b01111111, 0b01100111
```

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

```
/* code for Part III (not shown) */

/* Display R5 on HEX1-0, R6 on HEX3-2 and R7 on HEX5-4 */
DISPLAY:    LDR     R8, =0xFF200020 // base address of HEX3-HEX0
            MOV     R0, R5          // display R5 on HEX1-0
            BL      DIVIDE          // ones digit will be in R0; tens
                                    // digit in R1
            MOV     R9, R1          // save the tens digit
            BL      SEG7_CODE
            MOV     R4, R0          // save bit code
            MOV     R0, R9          // retrieve the tens digit, get bit
                                    // code
            BL      SEG7_CODE
            LSL     R0, #8
            ORR     R4, R0
            ...
            code for R6 (not shown)
            ...
            STR     R4, [R8]        // display the numbers from R6 and R5
            LDR     R8, =0xFF200030 // base address of HEX5-HEX4
            ...
            code for R7 (not shown)
            ...
            STR     R4, [R8]        // display the number from R7
```

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