CSCI210 Assembly Language and Computer Systems
Lab Assignment

Lab Description:
Write an ARM assembly program that will toggle the case of multiple characters in an ASCII string. The string will be entered by the user using Linux SYS (System) calls to read a value from the keyboard. The modified string will then be written to the console window.

In order to read a value from the keyboard you must load appropriate values into registers before switching processor modes to Supervisor. This is analogous to passing arguments to a function. The Linux "read" function needs to know

- What is the source of the read?
- How much data will be read (how many bytes)?
- What is the destination of the read (where will the read data be stored)?

Register	Value	Meaning
R7	3	Linux Sys call number for "read"
R0	0	Where are you reading from? 0 => Keyboard
R2	number	How many characters are to be read? Size in Bytes
R1	address	Address of buffer. What is the destination of the read?

Use Linux SYS calls to write a value to the Linux terminal. Before writing a value to the monitor you must load appropriate values into registers. The Linux "write" function needs to know

- What is the destination of the write (where will the data be written to)?
- How much data will be written?
- What is the source of the write (where is the data that is to be written)?

Register	Value	Meaning
R7	4	Linux Sys call number for "write"
R0	1	Where are you writing to? 1 => Terminal
R2	number	How many characters are to be written? Size in Bytes
R1	address	Address of buffer. What is the source of the write?

<u>LAB TASKS:</u>

Throughout this lab you must comment each line of your source with a descriptive message

Task One:

In this task you will be reading string data from the keyboard and writing string data to the console.

1. Boot the Pi and create a new directory in your CSCI210 folder called Lab_Three.

2. Using nano create a new source file called **ascii_lab.s** Save your source file into the directory you created in step 1.
    1. Add the start label as has been done in previous labs. You will always be doing this.
    2. Also add an **exit:** label that marks the position of the exit routine we have been employing. This routine includes moving 0 into R7 and switching to supervisor mode with an argument of 0. This essentially calls the "exit" routine that is part of the Linux kernel.
    3. All other instructions will go between these two labels.

3. Under the **start:** label, add the label **_display:** and add the code to write a string to the screen using the tables above for program setup. Follow these instructions carefully
    1. At the end of your code section (after the exit routine) add a **.data** directive. You will be placing all of your data values and buffer addresses in this area.
    2. Under the **.data** directive add the following:
        1. **_message: .ascii "Enter a 10 character string\n", 0**
        2. This declares that you are storing a string of ASCII characters in memory. **.ascii** is a directive that communicates instructions to the assembler and it allows you to write a string of ASCII bytes contained within quotes. ASCII strings need to be **null terminated.** The 0 defines the end of the string in memory. sYou can reference the address of this via the label **_message.** The address referenced by **_message** is an **offset to a location within the program segment**; specifically the offset of the first character of the message.
        3. Basic source code structure

```
.text

_start:      @regular program code

_display:    @code to write to the terminal window

_exit:       @exit code goes here

.data        @declare data after the code

_message:    @label for address of a data item
```

        4. You can load the address of **message** into the proper register from the table above.

            1. **LDR R1, = _message**

        5. Save, assemble, link and run the program to ensure that it builds and links and runs correctly. You should see the message appear on the screen

4. After the previous code snippet that displays a message, add the label **read:** and add the code to read a value from the keyboard using the table above. Use the following declaration to set up a buffer to hold your read.
    1. In the **.data** area, under the **_message** add the following

        1. **_buffer: .ascii ""**
        2. This sets aside some memory to store the value that will be read. The **.ascii** directive specifies that string data will be stored there. **Remember:** ASCII data is a single byte per character!! This will be important when you are moving from one character address to the next.

    2. **Do not forget to switch to supervisor mode to execute the sys call**
    3. When you execute this program you should see the message appear on the screen and then the program will pause while it awaits your input. Input 9 characters and hit enter. The new line character will become your 10th character. It is important to have that new line character be a part of your string

5. Write the entered text to the screen by adding another labeled section after the **read:** section

6. Summary

    1. Display the message using a Linux SYS call for write
    2. Read a string using a Linux SYS call for read
    3. Display the entered string using the write SYS call
    4. **Remember:** Supervisor mode is required for each SYS call you want to execute

Task Two:

In this task you will implement code to toggle the case of the entered ASCII string. You will be converting lower case text to upper case text using an AND operation. This operation will be applied to each character in the entered string. Be sure to enter lower case text when you run this program so that you can see the results. **Any spaces or non-non-alphabetic characters must be skipped.** Check the ASCII table for the ranges of non-non-alphabetic characters. I do not expect you to handle everything but I do want you to demonstrate that you can branch past ASCII characters 32 - 47.s

1. Using the **ascii_lab.s** source file add a new labeled section after the **read:** section. Label this section **_toggle:**
    1. This section of code will iterate through the entered character string, 1 character at a time, applying an AND operation to force Bit$_5$ to zero. I would like for you to do also do this using the BIC instruction.
2. **Remember:** ASCII data is a single byte per character. You will need to access a single byte at a time. The normal **LDR** instruction is a **32 bit operation**

   1. You can accomplish reading a single byte using the **LDRB** instruction. **L**oa**DR**egister**B**yte. This will perform an 8 bit read instead of a 32 bit read. The 8 bits will be placed into the LOW ORDER byte of the destination register. The remaining 28 bits will be zero filled.

3. Your loop's first task is to load the first byte of **buffer** into a register so you can manipulate it. Ensure that the address of buffer is stored in a register. You will need a counter to keep track of which character you are currently on. Choose a general purpose register for this.

   1. If R1 contains the address of buffer and R2 is your counter, initially set to zero you can load the first byte like this

      1. **LDRB R0, [R1, R2]** @ this does **implicit addition** of R1 + R2 to get an effective address [address + offset]. R1 is not modified
      2. You can then modify the counter by adding one to R2
      3. You may also do this using any of the ARM addressing modes we discussed in class. Check the documentation.

4. When the byte has been loaded into the register force Bit_5 to zero using a mask and an AND operation. I would also like for you to do this using the BIC instruction; just to see the difference. Comment out the unused instruction(s).

5. Write the byte **back to the buffer** using the **STRB** instruction. **ST**o**R**e**B**yte. Use your address register and counter for this operation just as above

6. Increment your counter (if you are not using one of the auto-increment address modes)

7. Using the **(B)ranch** instruction branch back to the **toggle:** label and continue on. **Don't let this go infinitely.** Use the CMP instruction to test the value in the counter register and execute a conditional branch.

8. When the loop is complete display the modified string to the console window

Things to consider:

   1. How many times do you need to loop?
   2. How will you stop the loop when you have completed?

Task Three: Complete in a separate source code file

ASCII Encryption: A simple encryption method is to perform an XOR operation on an ASCII byte with a "key". A key can be any numeric value. XORing the byte with the key generates a new value. XORing the new value again with the same key will return the original byte value.

Example: Encrypt and Decrypt ASCII ‘A’ using 255 as the key

```
‘A’   =>    ASCII 65   => 0100_0001
Key   =>    255        => 1111_1111
===================================
Encrypted value=> XOR => 1011_1110
                  Key => 1111_1111
===================================
Decrypted value=> XOR => 0100_0001
```

Encrypt an ASCII string by XORing a character with a byte sized encryption key. You choose a “key”. This can be any number between 0 - 255. Use this key to perform an XOR operation **on each byte** of an ASCII string. Display the encrypted string. Reverse this process and display the decrypted string to ensure that the process works correctly. Allow the string to be entered from the console window. You do not need to worry about skipping non-alphabetic characters.

When you have finished submit your source files only