CSCI210 Computer Architecture and Organization
Lab Assignment

Objectives

- Structure assembly code with procedures
- Branching and Linking
- Use the PC, LR and SP registers correctly
- Dealing with multiple source code files
- Bitwise operations, conditional execution

Sample Procedure Documentation: Assembly documentation needs to be thorough as there aren't nice IDE tools to communicate argument names/types, return types and so on. You do not have to duplicate this sample but do adopt it for your work in this lab and any subsequent labs where you are defining procedures. I expect to see this for every procedure.

```
@ ======================================
@ XOR_Sum sub procedure
@ ======================================
@
@ Purpose:
@ +++++++
@ The XOR_Sum procedure will accept a bit string
@       and return its XOR Sum
@
@ Initial Condition:
@ ++++++++++++
@ The first argument on the stack will be the 32 bit
@       string
@
@ Final Condition:
@ ++++++++++++
@ The XOR Sum calculated from the initial condition
@       will be placed in R0
@
@ Registers Used: R1, R2, R3
@
@ Sample Case:
@ ++++++++++
@ First Argument on stack: 0xF0F0
@ Final Result: XOR Sum of 0 placed into R0
@ =======================================
```

Task One:

- Create a source code file called ***io.s***
- In this file you will define ARM procedures to abstract the **write** and **read** Linux syscalls.
- Define the following procedures

    - **write:**
        - This procedure should contain the code to specify the Linux "write" sys call and the code to switch to supervisor mode
        - The size of the data to be written and the location of the source buffer should be supplied as arguments outside of the procedure in the calling area. The syscall itself can be specified in the procedure definition
        - Add documentation in the form of comments as to what arguments are expected and where they will be expected. IE which registers they will be placed in prior to the procedure call, or on the stack.
        - Move the contents of the link register to the program counter to return from the procedure
        - Add a **global** directive for the procedure label. The linker will need to be able to discover this label to resolve calls from other files.

    - **read:**
        - This procedure should contain the code to specify the Linux "read" sys call and the code to switch to supervisor mode
        - The size of the data to be read and the location of the destination buffer should be supplied as arguments outside of the procedure in the calling area. The syscall itself can be specified in the procedure definition
        - Add documentation in the form of comments as to what arguments are expected and where they will be expected. IE which registers they will be placed in prior to the procedure call.
        - Move the contents of the link register to the program counter to return from the procedure
        - Add a **global** directive for the procedure label. The linker will need to be able to discover this label to resolve calls from other files.

- **Test:** Create a source code file called ***testing_io.s***

    - This file will serve as your "main", so define it like you would the normal source files you've written. Give it a **_start** label and define all the data and buffers that you'll need.
    - Prompt the user for a brief message and echo the message to the terminal. Don't worry about dynamic buffer sizing. Just hard code the number of characters to read and write.
    - Assemble each individual file as normal
    - Linking needs to be updated to include all source files into the executable

        - **ld -o testing_io testing_io.o io.o**

    - Do some ***objdump*** and ***gdb*** tracing for your own edification. It is instructive to see how the assembler combines different source code files into a single executable.

Task Two:

- In the ***io.s*** source file add the procedure ***print_as_hex*** to print a 32 bit value in ASCII hex. This is a procedure-ified version of the ***print reg*** problem from the previous lab. You can decide on how you would like the argument(s) to be handled. Just provide comprehensive documentation. Have this procedure contain a nested call to the ***write*** procedure defined above.
- Note that when you have a nested procedure call you will need to think about how to preserve LR and restore LR. We will be covering this in much more detail, but you should begin thinking about it now. In general, each procedure should preserve LR as the first statement and restore it prior to returning; that way any nested calls can overwrite LR without fear, and each procedure can remember where it was called from.

    - The general approach to this task is to push LR onto the runtime stack upon procedure entry and to pop LR into PC prior to procedure exit. There are ***pseudo instructions*** for this

        -
          ```
          PUSH {lr}
          POP  {lr}
          ```

          But these are aliases for the actual stack operations

        -
          ```
          str lr, [sp, #-4]! @ pre-decrementation {--sp}
          ldr pc, [sp], #4   @ post-incrementation{sp++}
          ```

    - We will learn about why we need to decrement SP when "pushing" and increment SP when "popping" so don't worry too much about it at present.
    - You are required to use and understand the explicit manipulation of the stack pointer register. We will cover the runtime stack in much more detail next week.

- In ***testing_io.s*** prove that this works correctly

Task Three:

- Create the source file called ***bitwise.s***
- Define the procedure ***counts***. This procedure will count the number of 0 and 1 bits in a 32 bit argument.
    - Place the count of 0's in R0
    - Place the count of 1's in R1
- In ***testing_io.s*** prove that this works by calling the procedure and writing the results by preparing the number of counts as an ASCII hex string using ***print_as_hex***. Also include a descriptive message. Do not use ***printf***

Task Four:

- In ***bitwise.s*** define the procedure ***russian_peasant.*** This procedure will multiply two numbers using the Russian peasant multiplication approach.
    - https://www.wikihow.com/Multiply-Using-the-Russian-Peasant-Method

- Implement the halving with a right shift and the doubling with a left shift
- This procedure will accept two arguments
- Normally multiplication requires double the bits of the operands, but you may use smaller values to keep the result within 32 bits and put the result in R0. To extend this problem you could deal with large numbers and place the LO word of the result in R0 and the HO word in R1.
- In ***testing_io.s*** prove that this works by calling the procedure and writing the results by preparing the product as an ASCII hex string using ***print_as_hex***. Also include a descriptive message

- **Submission: *testing_io.s, io.s, bitwise.s, makefile.*** If I left a file off of this list please submit it. Also, please attach a zip file, so I do not have to download each file individually.