Lab 4: Assembly

Lab Goals

- Assembly language primer: basic instructions.
- C calling convention: accessing the function's arguments.
- Calling C library functions from Assembly code.

This lab can be done in pairs!

As usual, you should read and understand the reading material and complete task 0 before attending the lab.
For this lab you are supposed to use stdlib functions.

Task 0: Basic Command-line Arguments Printing

Task 0 is crucial for the successful completion of this lab! make sure you finish it and understand it before your lab session.
Read the Assembly lecture Assembly Language Primer. For this task you must understand the arguments of main( ), how to access the arguments of a function in assembly language (discussed in class), and how to pass arguments to a function in the C calling convention. Be careful not to mess up your stack!

In this task you need to write function starting with the (global) label "main" in assembly language which performs the following:

- print **argc** in decimal format to stdout using printf
- print **argv[0]** (the first argument to the main function) to stdout using puts

Now, write a makefile to compile the assembly code you wrote, and to link the resulting object file with te C standart library (gcc myfile.o). This makefile will be useful throughout the lab.

Task 1: Extended Arguments Print

Task 1 extends Task 0. In this task you need to print **argc** and all the elements **argv**. That is you need to write a program that iterates over the array.
As in task 0, print argc in decimal format using printf and the elements of argv print using puts.

Task 2: Structs and Multi-precision Integer Hexadecimal Printing

Read about the difference between little endian and big endian little vs. big endian.

Implement print_multi(struct multi *p)): gets a pointer to struct multi{unsigned int size; unsigned char num [ ]} where size is the number of bytes in the num array, and the num array is a multi-precision unsigned integer in **little endian**. The

function should print the value of the **entire** number in hexadecimal by calling printf("%x") once for every byte in the array.
Warning: please note that C library functions do not maintain the value of all your registers!

Test this by initializing a global struct, as in the following lines, and call print_multi from main with a pointer to the struct
x_struct: dd 5
    x_num: db 0xaa, 1,2,0x44,0x4f

Task 3: Addition of Multi-Precision integers

Overview

In this task you need to implement the function **struct multi* add_multi(struct multi *p, *q)**;
The function should perform an addition between two such numbers represented as structs, creating a third number represented the same way. This is done by byte-wise addition between the two arrays defined in the given structs while maintaining the carry between additions. The result should be placed in a newly allocated array in a new allocated struct of size 4+max(len1, len2).

Input:
Two arrays **array1, array2** (defined as variables in the code), of size **len1, len2** respectively.
For example:

x_struct: dd 5
    x_num: db 0xaa, 1,2,0x44,0x4f

y_struct: dd 6
    y_num: db 0xaa, 1,2,3,0x44,0x4f

Output:
Without loss of generality, assume that len1 > len2. Therefore

- **max_len = max(len1,len2)=len1**
- **min_len = min(len1,len2)=len2**

The function will return an array, dynmically allocated using malloc, **result_array**, of size **max_len** such that:

- **result_array[i]=array1[i]+array2[i]+cy** for 0 <= i < min_len.
- **result_array[i]=array1[i]+cy** for min_len <= i < max_len.

cy is the result of the carry from the previous addition.

If you have difficulty doing this with dynamic memory allocation, which uses malloc(size) as in C, reserve the array in the data section, to earn 80% of the credit on this task.

Task 3.A: Get MaxMin

Implement this assembly language function **not** in the C calling convention. Given pointers to number structures in eax and ebx, return the pointer to the one with the higher length field in eax, and the other pointer in ebx.

Task 3.B: add_multi Implementation

Use the MaxMin function and Print_multi you wrote to implement and test the element-wise addition, and print each number to be added and the result in separate lines to stdout.

Test your function by defining appropriate initialized number structs and printing the resulting array.

Example

For the following structs:
x_struct: dd 6
    x_val: db 1,0xf0,1,2,0x44,0x4f
y_struct: dd 5
    y_val: db 1,1,2,0x44,1
Printing the result array should produce:
2
F1 (which is 0xf1)
3
46 (which is 0x46)
45 (which is 0x45)
4F (which is 0x4f)

Task 4: Pseudo-Random Number Generator

Implement a function name **rand_num** that uses basic assembly insturction in order to generate a random number using a "linear-feedback shift register". See LFSR in Wikipedia The function uses a global initialized (not to zero!) unsigned 16-bit (word) STATE variable, and a constant MASK variable. Use the mask for the Fibonacci LFSR for 16 bits. Each pseudo-random operation does:

- Use the MASK to get just the relevant bits of the STATE variable.
- Compute the parity of the above relevant bits.
- Shift the bits of the (non-maked) STATE variable one position to the right, with the MSB determined by the parity you just computed.

Test your function by printing 20 consecutive generated pseudo-random numbers in hexadecimal using printf.

Submission

Task 1-3 are mandatory and task 4 can be completed later on in the completion lab. You are required to submit a zip file in the format [your_id].zip that will conatain a folder for each task and each folder will contain the assembly code file and the corresponding makefile.