

Lab D: Assembly Language "do at home" lab.

Lab Goals

- Assembly language primer: improving proficiency in assembly language features.
- Interfacing C to assembly code-continued
- Using dynamically allocated memory
- Multi-precision addition.
- Pseudo-random number generation.

This lab may be done in pairs!

As usual, you should read and understand the reading material and complete part 0 before attempting to do the lab assignment.

For this lab, unlike the previous lab (lab 3) you **are** supposed to use stdlib functions. Make sure you compile and link with the CDECL conventions, as otherwise the C to assembly interface you have used before will not work!

Part 0: Basic Command-line Arguments Printing using stdlib

Part 0 is crucial for the successful completion of this lab! make sure you finish it and understand it before implementing your program to be submitted.

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 CDECL calling convention. Be careful not to mess up your stack!

In this preliminary 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[i]** to stdout using puts, for all i from 0 to argc-1

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

The lab assignment: Multi-Precision Integer IO and Adder

We have partitioned the lab work into parts, suggesting the order of implementation and testing. Nevertheless, you are supposed to submit a single program that ties it all together as stated in part 4 below.

Part 1: Structs and Multi-precision Integer Hexadecimal Printing and Reading

Read about the difference between little endian and big endian [little vs. big endian](#).

Part 1.A: Printing a Multi-precision Integer

Implement `print_multi(struct multi *p)`: gets a pointer to struct `multi{unsigned char size; unsigned char num []}` where `size` is the number of bytes in the `num` array (always greater than 0), 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("%02hhx")` once for every word in the array. If the number contains leading zeros, you may wish to remove them in the output, but this is not a requirement in the assignment.

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`:

```
x_struct: dw 5
x_num: dw 0xaa, 1, 2, 0x44, 0x4f
```

The output in this case should be (with a linefeed at the end):

```
4f440201aa
```

Part 1.B: Reading a Multi-precision Integer

After you implement and test the printing, you should implement a function `getmulti` that reads a line from `stdin` using `fgets`, containing only a sequence of hexadecimal digits, and stores it in the above type of structure. You may assume that the input contains no leading zeros. Use your printing function to see that your input is correct. You may assume that the input line contains less than 600 characters. Note that your code will be simpler if you process hexadecimal characters in **pairs**.

Think: how do you **very simply** make sure you always need to process an even number of hex digits?

Part 2: 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 $1 + \max(\text{len1}, \text{len2})$.

Input:

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

For example:

```
x_struct: dw 5
x_num: dw 0xaa, 1, 2, 0x44, 0x4f
y_struct: dw 6
y_num: dw 0xaa, 1, 2, 3, 0x44, 0x4f
```

Output:

Without loss of generality, assume that $\text{len1} > \text{len2}$. Therefore

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

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

- **result_array[i]=array1[i]+array2[i]+cy** for $0 \leq i < \text{min_len}$.
- **result_array[i]=array1[i]+cy** for $\text{min_len} \leq i < \text{max_len}$.

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

Part 2.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**.

Part 2.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.

Part 3: Pseudo-Random Number Generator (PRNG)

Implement a function name **rand_num** that uses basic assembly instructions 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. Note: we recommend, but not require, that you use the parity flag!
- Shift the bits of the (non-masked) STATE variable one position to the right, with the MSB determined by the parity you just computed.

First, test your function by printing some generated pseudo-random numbers in hexadecimal using printf. Once you have done that, write a function PRmulti: uses the PRNG to create a pseudo-random Multi-precision Integer as follows: the first 8 bits generated by the PRNG determine the length n in bytes of the number (generate a new random byte instead if this is zero!), and then 8*n PRNG bits determine the actual value to be inserted into the appropriate struct.

If done properly, you should be able to use your printing function from part 1.A to print the resulting numbers, do so and thoroughly test your code.

Part 4: Putting it all together

Your final program "multi" should combine all the above as follows. The program should print to stdout the numbers to be added (in hexadecimal), and then their sum. Then the program exists normally. The source of the numbers is to be determined as follows:

- By default (no command-line arguments), the program operates (i.e. prints and adds) on the numbers encoded by x_struct and y_struct.
- If argv[1] is "-I", the program operates on numbers obtained from stdin, one number per line (as in part 1.B)
- If argv[1] is "-R" the program operates on numbers obtained from the Pseudo-Random number generator, as in part 3.

Some example of program runs follow. First, the default with no command line arguments:

```
$ ./multi
4f440201aa
4f44030201aa
4f9347040354
```

In the examples with the "-I" flag below, the first two lines are input lines. You may have an extra leading zero in each number's output.

```
$ ./multi -I
ffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffff
```

