AUTUMN 2018 CSE 2421 LAB 4

DUE: Tuesday, November 27th, by 11:30 p.m.

Objectives:

- Assembly language programs in X86-64
- Using assembler directives in X86-64
- Writing code for functions in X86-64; passing parameters in X86-64
- Move, ALU, jump, call, and return instructions in X86-64
- Instructions for two byte ("word") operands in X86-64
- Calling C library function printf to print output in X86-64

REMINDERS:

– Every lab requires a README file. The required name of this file for this lab is **lab4Readme** [Case matters!]. Do not change the name, or your lab submission will be penalized 20%. The file should have EXACTLY this name, with no extensions, such as .txt, etc. The Linux OS can determine that the file is a text file without any extension. Just create a text file with the text editor of your choice on stdlinux, and save it with the name specified. This file should include the following:
    - Student name;
    - Total amount of time (hours) to complete the entire lab;
    - Short description of any concerns, interesting problems or discoveries encountered, or comments in general about the contents of the lab.

– You should always aim to hand an assignment in on time. If you are late (even by a minute – or heaven forbid, less than a minute late), you will receive 75% of your earned points for the designated grade as long as the assignment is submitted by 11:30 pm the following day, based on the due date shown at the top of this document. If you are more than 24 hours late, you will receive a zero for the assignment and your assignment will not be graded at all.

– Any lab submitted that does not assemble (using the command indicated below) and run without crashing WILL RECEIVE AN AUTOMATIC GRADE OF ZERO for the lab. No exceptions will be made for this rule - to achieve even a single point on a lab your code must minimally assemble and execute without crashing. To assemble your code, you must use the following command (this is the command the graders will use also):

At the Linux command line prompt:

```
%gcc –m64 -o lab4 lab4.s
```

- You are welcome to do more than what is required by the assignment as long as it is clear what you are doing and it does not interfere with the mandatory requirements.

- You are responsible for making sure that your lab submits correctly to Carmen Canvas.

– You are required to comment your assembly language code, and the quality and clarity of the comments will be given weight in the grade for the lab, as usual.

GRADING CRITERIA (approximate percentages listed)

–(20%) The code and algorithm are well commented.
  - A comment should be included in the main program including the programmer name as well as explaining the nature of the problem and an overall method of the solution (what you are doing, not how).
  - A short comment should be included for each logical or syntactic block of statements, including each function.
  - A comment should be included for values in registers, and stack set up and clean up instructions in each function; these comments should help the reader of your program to understand the algorithm being used, and where various values in the program are being stored.

–(20%) The program should be appropriate to the assignment, well-structured and easy to understand without complicated and confusing flow of control.

–(60%) The results are correct, verifiable, and well-formatted. The program correctly performs as assigned.

LAB DESCRIPTION

Assembly language program in X86-64 assembly language. Mandatory file name: **lab4.s**

For this lab, you will write an X86-64 assembly language program which has **5 functions,** with the C language function prototypes shown below:

```
int main();

void multInts(int size, int *array1, int *array2);

void addShorts(short size, short *array1, short *array2);

void invertArray(int size, int *array1);

void printArray(int size, int *array1);
```

See below for a description of what each of these functions should do.

PROBLEM:

Write an assembly language program with five procedures or functions using X86-64. The program should have 4 static arrays, defined in the data section of the program (that is, stored on the heap), including two int arrays, intArray1 and intArray2, both of the same size, and two short arrays, shortArray1 and shortArray2, both of the same size. There should also be two other variables defined in the data section, one of which is an int, sizeIntArrays, and will hold the size of the int arrays, and the other of which is a short, sizeShortArrays, and will hold the size of the short arrays. **Do not change any of these labels in the data section** (case matters!).

Required data section:

```
.data
sizeIntArrays:
        .long 5
sizeShortArrays:
        .word 4
intArray1:
        .long 10
        .long 25
        .long 33
        .long 48
        .long 52
intArray2:
        .long 20
        .long -37
        .long 42
        .long -61
        .long -10
shortArray1:
        .word 69
        .word 95
        .word 107
        .word 12
        .word 332
shortArray2:
        .word 27
        .word -87
        .word 331
        .word -49
        .word -88
```

CONSTRAINTS:

• Your program must have five procedures/functions, with the labels main, multInts, addShorts, invertArray, and printArray (**Do not change these:** case matters! We will not penalize you for any changes made by Carmen, though.).

- You need to write the main procedure to call the other four procedures at appropriate points, and with the appropriate parameters in the correct registers to pass to the procedures, so that they do what is described below. You are allowed to do other work in main to get parameter values to pass to the procedures before you call them, but NO OUTPUT SHOULD BE PRINTED FROM main.
- The multInts procedure should first call printf to print "Products" on one line, and then multiply the first value in the first int array, intArray1, times the first value in the second int array, intArray2, and call printf to write the result to output on a separate line, then multiply the second value in the first int array times the second value in the second int array, and write the result to output on a separate line, and so on, for each pair of values consisting of one value from the first int array, and the second value in the same position in the second int array. A blank line should be printed at the end of all of the products.
- The addShorts procedure should first call printf to print "Sums", and then add the first value in the first short array, shortArray1, to **the last value** in the second short array, shortArray2, and call printf to write the sum to output, then add the second value in the first short array to **the second from the last** value in the second short array, and call printf to write the sum to output, and so on, for each of the n pairs of values in the two arrays consisting of one value at index position i in the first short array, and the second value at index position n – 1 – i in the second short array. A blank line should be printed at the end of all of the sums.
- The printArray procedure will be called twice, the first time before the call to invertArray, and the second time after. It should first call printf to print "Elements in intArray1", and then, on the following lines, print the values in the array it is passed in order, from the first value to the last value, one value per line. printArray should be called with a pointer to intArray1, as well as the size of the array it is to print. It should also print a blank line following all of the values in the array.
- The invertArray procedure should invert, or reverse, the elements in intArray1, but prints no output. The elements in intArray1, after they are placed in inverse order by invertArray, will be printed out by procedure printArray, below, which will be called from main after the invertArray procedure returns to main.
- The printArray procedure should be called the second time after the call to, and return from, invertArray. It first calls printf to print "Elements in intArray1", and then, on the following lines, print the values in the array it is passed in order, from the first to the last, one value per line. printArray should be called the second time with a pointer to intArray1 after the elements are inverted, or reversed, by invertArray, to print out the elements in the opposite order which they were originally in. It will also be passed the size of the array it is to print. It should also print a blank line following all of the values in the array.
- To pass values to procedures, you must use the appropriate registers, as discussed in class and shown in the class slides.

- You should also follow the X86-64 conventions discussed in class, and covered in the class slides, related to caller and callee save registers, and returning values from procedures in register rax or an appropriate subregister.
- Remember that you need to put format strings in read only memory in order to call printf to print output, as well as passing the values of any variables to be printed in the appropriate register(s). See the first X86-64 program in the class slides for examples of how to do this.

CAREFUL: Be sure to pay attention to the conventions in X86-64 for parameter registers, and also for caller save/callee save registers. If you fail to do this, you will get segfaults almost certainly when you run your program!

NOTE: Until you are ready to write the code for a function, you can use a "stub" function in assembly language. To use a stub function, you use assembler directives as for any function, but the code in the function should only be a ret instruction. This will cause the function to return immediately when it is called, and it will do nothing. When you are ready to write the rest of the code for the function, you can add it, resave and reassemble, and then run the program to test the function.

OUTPUT:

- For the sample input given above, the output should be as follows:

```
Products
200
-925
1386
-2928
-520

Sums [NOTE CHANGE FROM ORIGINAL LAB INSTRUCTIONS!]
20
426
20
39

Elements in intArray1
10
25
33
48
52

Elements in intArray1
52
48
33
25
```

LAB SUBMISSION

You should submit all your lab assignments electronically to Carmen Canvas. See the instructions for Lab 1 and posts on Piazza for guidance.

NOTE:

- Your programs MUST be submitted in source code form. Make sure that you submit the program code as a .s file only. Do NOT submit the object files and/or the executable. If you only submit the executable file for the program, you will receive NO CREDIT, and no exceptions will be made for this rule!

- It is YOUR responsibility to make sure your code can assemble and run without causing runtime errors on the CSE department server stdlinux.cse.ohio-state.edu using gcc –m64 -o lab4 lab4.s as specified earlier in this lab description.