ECS 50: Programming Assignment #5

Instructor: Aaron Kaloti

Fall Quarter 2021

Contents

1 Changelog

You should always refer to the latest version of this document.

- v.1: Initial version.
- v.2: Updated tip in part #4 about 16-byte alignment to direct you to the 12/01 Canvas announcement.
- v.3: Added submission details.

2 General Submission Details

Partnering on this assignment is prohibited. If you have not already, you should read the section on academic misconduct in the syllabus.

This assignment is due the night of Saturday, 12/04. Gradescope will say 12:30 AM on Sunday, 12/05, due to the "grace period" (as described in the syllabus). *Be careful about relying on the grace period for extra time; this could be risky.*

3 Programming Problems

Throughout these directions, any use of the term "integer" refers to a 32-bit, signed integer.

3.1 File Organization and Compilation

You will ONLY submit the files `count_target.s`, `sum_within_range.s`, and `functions.s`. Skeleton versions of these files are provided on Canvas.

The CSIF is the reference environment. You should make sure your code can be compiled and behave properly on the CSIF.

Needless to say, you should use the compilation comamnds used in the examples. These are the compilation commmands that will be used when grading your submission. **If your code for a certain part cannot be compiled on the CSIF, then you will automatically get a zero for that part. Since part #3 and part #4 expect you to implement two functions in the file `functions.s`, if anything in that file causes compilation to fail, then you will earn a zero on both parts #3 and #4.** In these compilation commands, I use `-g`, but you don't need to do that unless you plan on using a debugger such as GDB.

3.2 Restrictions

In this assignment, all code that you write must be x64 (x86-64) assembly code (using the AT&T syntax) for an x64 computer that has a Linux operating system. ***You must write this assembly code yourself; you cannot use a tool, such as a compiler[1], to generate assembly code. For instance, you cannot write C++ code and then use a compiler to convert it to x64 assembly code. We will not accept that, and we will easily be able to tell if you submitted tool-generated assembly code.***

You are not allowed to use any global variables (e.g. variables/arrays in the `.data` section) other than the ones in the versions of the files provided on Canvas.

You may be penalized if any of your code modifies global variables, e.g. the input array in part #1.

3.3 General Advice

- Pay close attention to when operand sizes should be 32 bits vs. 64 bits. As talked about in lecture, failure to do so can lead to tricky bugs. If you want to use instruction suffixes, please do.
- I wouldn't modify the stack pointer in parts #1 or #2.

3.4 Part #1: Count Target

File: `count_target.s`

In this part, you are writing the program you wrote for part #1 of P2, except for an x64 processor instead of for an RV32EM one.

Write a program that counts the number of times that a target integer (given in a variable `target`) appears in an array of integers called `arr` whose length is given in a variable called `arrlen`.

As you can see in the `count_target.s` file provided on Canvas, there is already code at the bottom to print out the final count. You just need to make sure that the final count ends up in the RSI register.

Below is an example of how your program should behave. `input_count_target.s` is provided on Canvas.

```
$ cat input_count_target.s
.data

.global arr, arrlen, target
arr:
    .long 5, 13, -8, 19, 20, 13, 5, 13
arrlen:
    .long 8
target:
    .long 13
$ gcc -no-pie -Wall -Werror -g count_target.s input_count_target.s -o count_target
$ ./count_target
Count: 3
```

I imagine some of you may try to use example inputs from part #1 of P2 when testing this program. If you do so, make sure to keep in mind that in x64, `.word` is 16 bits, not 32 bits.

3.5 Part #2: Sum Within Range

File: `sum_within_range.s`

[1]Here, by "compiler", we mean the program that does the step of converting source code to assembly code.

In this part, you are writing the program you wrote for part #2 of P2, except for an x64 processor instead of for an RV32EM one.

Write a program that iterates through an array of integers called `arr` and calculates the sum of all integers in the array that are between (inclusive) a lower boundary (given by the variable `low`) and an upper boundary (given by `high`). (You may assume that `low` is always less than or equal to `high`.)

As you can see in the `sum_within_range.s` file provided on Canvas, there is already code at the bottom to print out the final sum. You just need to make sure that the final sum ends up in the RSI register.

There will always be at least one value in the array that is between the lower boundary and the upper boundary.

Below is an example of how your program should behave. `input_sum_within_range.s` is provided on Canvas.

```
$ cat input_sum_within_range.s
.data

.global arr, arrlen, low, high
arr:
    .long 7, -8, 12, 5, 100, 4
arrlen:
    .long 6
low:
    .long 5
high:
    .long 15
$ gcc -no-pie -Wall -Werror -g sum_within_range.s input_sum_within_range.s -o sum_within_range
$ ./sum_within_range
Sum: 24
```

I imagine some of you may try to use example inputs from part #2 of P2 when testing this program. If you do so, make sure to keep in mind that in x64, `.word` is 16 bits, not 32 bits.

3.6 Part #3: Add

File: `functions.s`

At the top of `call_add.c` (which is provided on Canvas), you can see the following declaration:

```c
int add(int a, int b);
```

In `functions.s` (you should start with the version of this file provided on Canvas), you will implement this function in x64 assembly code. The function simply returns the sum of its two arguments.

Because `add()` is called by C code (that will become compiler-generated assembly code), your function must follow the System V AMD64 ABI in regards to the following:

- Where `add()` expects the arguments.
- Which registers `add()` needs to preserve.
- Where `add()` places the return value.

Below is an example of how your function should behave.

```
$ cat call_add.c
#include <stdio.h>

int add(int a, int b);

int main()
{
    printf("%d\n", add(5, 11));
    printf("%d\n", add(3, -1));
}
$ gcc -no-pie -Wall -Werror -g call_add.c functions.s -o call_add
$ ./call_add
16
2
```

3.7 Part #4: Sum Inputs

File: `functions.s`

At the top of `call_sum_inputs.c` (which is provided on Canvas), you can see the following declaration:

```c
int sumInputs(int targetNum, int targetVal);
```

In `functions.s`, you will implement this function in x64 assembly code. This function keeps prompting the user to enter an integer until one of the following occurs:

- The user enters a number of integers equal to `targetNum`.
- The user enters `targetVal`.

After the function is done getting input from the user, the function should return the sum of all integers entered by the user. (If the user enters `targetVal`, then that should not contribute to the sum.)

Because `sumInputs()` is called by C code (that will become compiler-generated assembly code), your function must follow the System V AMD64 ABI in regards to the following:

- Where `sumInputs()` expects the arguments.
- Which registers `sumInputs()` needs to preserve.
- Where `sumInputs()` places the return value.

Below are examples of how your function should behave.

```
$ cat call_sumInputs.c
#include <stdio.h>

int sumInputs(int targetNum, int targetVal);

int main()
{
    printf("sumInputs() returned: %d\n", sumInputs(5, 800));
}
$ gcc -no-pie -Wall -Werror -g call_sumInputs.c functions.s -o call_sumInputs
$ ./call_sumInputs
2
3
6
2
1
sumInputs() returned: 14
$ ./call_sumInputs
100
30
20
-10
-42
sumInputs() returned: 98
$ ./call_sumInputs
30
50
800
sumInputs() returned: 80
$ ./call_sumInputs
800
sumInputs() returned: 0
```

Here are some tips specific to part #4:

- In the version of `functions.s` provided to you on Canvas, there is a global variable called `siScanfFormatStr`. This variable contains the format string that you should use when you call `scanf()` to get the user's input.
- See the 12/01 Canvas announcement about 16-byte alignment for the stack.

4 Grading Breakdown

As stated in the updated syllabus, this assignment is worth 6% of your final grade. Below is the approximate worth of each individual part of this assignment.

- Part #1: 1.5%
- Part #2: 1.5%
- Part #3: 0.75%
- Part #4: 2.25%

5 Submission Details

As with the previous programming assignment, there is no autograder for this assignment.

Submitting on Gradescope: There is a place on Gradescope that you can submit to. **You will ONLY submit the files `count_target.s`, `sum_within_range.s`, and `functions.s`.** You may be penalized for submitting other files.

Note that only your active submission (i.e. your latest submission, unless you manually activate a different submission) will count. I'll only see the file that you submitted in your active submission.