ECS 50: Programming Assignment #5

Instructor: Aaron Kaloti

Winter 2021

Contents

1 Changelog

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

- v.1: Initial version.

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 Wednesday, 02/24. Gradescope will say 12:30 AM on Thursday, 02/25, 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.*

The autograder will compile your code with the commands used in the examples for each part, so you should use those same compilation commands on a Linux command line, esp. on the CSIF.

3 Grading Breakdown

As stated in the syllabus, this assignment is worth 7% of your final grade. Below is the breakdown of each part.

- Part #1: 1.5%
- Part #2: 2%
- Part #3: 3.5%

4 Submitting on Gradescope

Files to submit:

- `check_bits.s`
- `transpose.s`
- `aaronize.s`

Do not submit the caller code files, i.e. `call_check_bits.s`, etc.

During the 01/12 lecture, I talked about how to manually change the active submission, just in case that is something that you find yourself needing to do.

Once the deadline occurs, whatever score the autograder has for your active submission (your last submission, unless you manually change it) is your *final* score (unless you are penalized for violating a restriction, as mentioned below).

4.1 Regarding Autograder

The reference environment is the CSIF.

Your output must match mine *exactly.*

Once the autograder is released, information about the visible test cases will be inserted here.

5 Programming Problems

5.1 Restrictions

In this assignment, all code must be x86-64 assembly code. You must write this assembly code yourself. That is, you cannot write C++ code and then use `g++ -S` to compile that C++ code into assembly code to submit; we will not accept that, and we will easily be able to tell if you submitted compiler-generated assembly code.

None of your functions can use global variables. For example, within `check_bits.s`, you cannot have a `.data` section containing (global) variables that your `checkBits` function uses in order to operate. If you need such variables, you should instead use registers or stack-based local variables. ***We will manually check if you used global variables.*** The purpose of this is to get you to practice using registers and stack-based local variables more[1].

Your function for each part must preserve any register whose value is changed by the function. In the case of part #1, you should not preserve RAX; doing so would make no sense, because RAX is to contain the return value. There will be certain test cases that check for register preservation.

5.2 General Tips

In this assignment, you will implement three functions. As a reminder, each stack argument will take up 64 bits, since that's how x86-64 prefers things, even if the argument doesn't actually need 64 bits.

You do not have to use the combination of the stack frame and the base pointer (RBP) if you do not want to, but I highly recommend it. I personally found that it made determining the offsets easier. If you do use the stack frame setup, make sure that you are mindful about whether you are using indexed addressing with the RBP vs. the RSP.

5.3 Part #1: Check Bits

File: `check_bits.s`

In a file called `check_bits.s`, you will implement a function called `checkBits()`. This function takes as argument three integers. The second and third integer specify bit numbers (let's call them a and b), with 0 representing the least significant bit. The function should return 1 if the first argument's bit pattern has bits a and b set (to 1). If this is not the case, then the function should return 0. You may assume that $0 \leq a \leq 31$, $0 \leq b \leq 31$, and $a \neq b$.

All three arguments are passed on the stack, with the arguments being pushed in order. The return value must be placed in the RAX.

Hint: In some parts of this part, proper use of bitwise operations can save some time. I did not use a single loop.

Below are examples of how your function should behave. In the first example, the return value is 0, because $6_{10} = 0110_2$ has bit #1 set but not bit #3. In the second example, the return value is 1, because $11_{10} = 1011_2$ has bits #3 and #0 set. (The fact that bit #1 is set is immaterial.) `call_check_bits.s` is provided on Canvas.

[1] For the record, I have nothing against global variables in general. Professor Matloff, a professor in our department, wrote a great argument in defense of global variables that you can access here.

```
$ cat check_bits.s
.text

checkBits:
    # CENSORED: My implementation.
    ret
$ cat call_check_bits.s
.text
.globl _start
_start:
    # First call: Check if 6 has bits #1 and #3 set.
    push $6
    push $1
    push $3
    call checkBits
    add $24, %rsp
    mov %eax, %ebx
    # Second call: Check if 11 has bits #3 and #0 set.
    push $11
    push $3
    push $0
    call checkBits
    add $24, %esp
done:
    nop
.include "check_bits.s"
$ as --gstabs call_check_bits.s -o check_bits.o
$ ld check_bits.o -o check_bits
$ gdb ./check_bits
...
(gdb) b done
Breakpoint 1 at 0x400097: file call_check_bits.s, line 18.
(gdb) r
Starting program: /.../check_bits

Breakpoint 1, done () at call_check_bits.s:18
18      nop
(gdb) p $ebx
$1 = 0
(gdb) p $eax
$2 = 1
```

5.4 Part #2: Matrix Transpose

File: `transpose.s`

In this part, you will implement a function called `transpose()` that performs matrix tranposition. If you are not familiar with matrix transposition, you can find everything you need to know on the Wikipedia page. The short version is that you can find the transpose of a matrix by reflecting its elements along its main diagonal. For example, if you have this matrix:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

The transpose would be:

$$\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

The function takes four arguments and expects them to be pushed onto the stack in order:

1. The starting address of the input matrix (which is assumed to be in row-major order).
2. Number of rows.
3. Number of columns.
4. The starting address of the output matrix (which is assumed to be in row-major order as well).

This function does not return any value. Your function is not allowed to modify the original matrix.

Below is an example (really the example above) of how your function should behave. `call_transpose.s` is provided on Canvas.

```
$ cat transpose.s
.text

transpose:
    # CENSORED: My implementation.
    ret
$ cat call_transpose.s
.data

input:
    .long 1
    .long 2
    .long 3
    .long 4
    .long 5
    .long 6

output:
    .rept 6
    .long -1
    .endr

.text
.globl _start
_start:
    push $input
    push $3
    push $2
    push $output
    call transpose
    add $32, %rsp
done:
    nop
.include "transpose.s"
$ as --gstabs call_transpose.s -o transpose.o
$ ld transpose.o -o transpose
$ gdb ./transpose
...
(gdb) b done
...
(gdb) r
...
(gdb) p (int[6]) input
$1 = {1, 2, 3, 4, 5, 6}
(gdb) p (int[6]) output
$2 = {1, 3, 5, 2, 4, 6}
```

5.5 Part #3: Aaronization

File: `aaronize.s`

We[2] say that to *aaronize* a one-dimensional array *arr* of integers is to apply the following transformation to each element $arr[i]$ of the array, creating a new array *newArr*.

```
newArr[i] = arr[i]
If i == 0:  # if first element
    newArr[i] += arr[i + 1]
Else if i == len(arr) - 1:  # if last element
    newArr[i] += arr[i - 1]
Else:  # neither first nor last element
    newArr[i] += arr[i - 1]
    newArr[i] += arr[i + 1]
```

Again, this transformation is applied to *each element* of the array *arr*, producing a new array of integers (of the same length) that we call *newArr* in the above pseudocode.

Here is a concrete example. Suppose our original array is $arr = [7, 3, 1, 10]$. After one *aaronization*, we get $newArr = [10, 11, 14, 11]$. Here is the breakdown of the calculations:

- $newArr[0] = arr[0] + arr[1] = 7 + 3 = 10$.
- $newArr[1] = arr[0] + arr[1] + arr[2] = 7 + 3 + 1 = 11$.

[2] By which I mean "nobody".

- $newArr[2] = arr[1] + arr[2] + arr[3] = 3 + 1 + 10 = 14$.
- $newArr[3] = arr[2] + arr[3] = 1 + 10 = 11$.

If we *aaronize* the array again, we go from $arr = [10, 11, 14, 11]$ to $newArr = [21, 35, 36, 25]$. Below is a breakdown of the calculations:

- $newArr[0] = arr[0] + arr[1] = 10 + 11 = 21$.
- $newArr[1] = arr[0] + arr[1] + arr[2] = 10 + 11 + 14 = 35$.
- $newArr[2] = arr[1] + arr[2] + arr[3] = 11 + 14 + 11 = 36$.
- $newArr[3] = arr[2] + arr[3] = 14 + 11 = 25$.

If we *aaronize* one more time, we go from $arr = [21, 35, 36, 25]$ to $newArr = [56, 92, 96, 61]$. Below is a breakdown of the calculations.

- $newArr[0] = arr[0] + arr[1] = 21 + 35 = 56$.
- $newArr[1] = arr[0] + arr[1] + arr[2] = 21 + 35 + 36 = 92$.
- $newArr[2] = arr[1] + arr[2] + arr[3] = 35 + 36 + 25 = 96$.
- $newArr[3] = arr[2] + arr[3] = 36 + 25 = 61$.

In this part, you will implement a function called `aaronize()` that performs *aaronization* on an array of integers a given number of times.

The function takes the following arguments and expects them to be pushed onto the stack in order.

1. The starting address of the input array.
2. The length of this array.
    - You may assume the length is at least 3.
3. The number of times to *aaronize*.
    - You may assume this is at least 1.
4. The starting address of the output array.

This function does not return any value. Your function is not allowed to modify the original array of integers.

Below is an example (really the example above) of how your function should behave. `call_aaronize.s` is provided on Canvas.

```
$ cat aaronize.s
.text

aaronize:
    # CENSORED: My implementation.
    ret
$ cat call_aaronize.s
.data

input:
    .long 7
    .long 3
    .long 1
    .long 10

output:
    .rept 4
    .long -1
    .endr

.text
.globl _start
_start:
    push $input
    push $4
    push $3
    push $output
    call aaronize
    add $32, %rsp
done:
    nop
.include "aaronize.s"
```

```
$ as --gstabs call_aaronize.s -o aaronize.o
$ ld aaronize.o -o aaronize
$ gdb ./aaronize
...
(gdb) b done
...
(gdb) r
...
(gdb) p (int[4]) input
$1 = {7, 3, 1, 10}
(gdb) p (int[4]) output
$2 = {56, 92, 96, 61}
```

Tip #1: It is helpful to verify that what you have written so far works as you go along. In this part, I made a copy of the input array at one point and wanted to verify that the contents of this copy were correct. Below, I show how to do this with GDB. The starting address of the array copy is in the R13 register. I first determine what the starting address is, then I print out the four words[3] from that starting address onwards. I can see that the values 7_{16}, 3_{16}, 1_{16}, and A_{16} match the values of the input array.

```
(gdb) p/x $r13
$4 = 0x7fffffffdcf0
(gdb) x/4w 0x7fffffffdcf0
0x7fffffffdcf0: 0x00000007  0x00000003  0x00000001  0x0000000a
```

Tip #2: I regretted not making a helper function to copy the contents of one array into another array. You may find such a helper function useful, depending on your approach. I would definitely recommend a helper function for any operation (taking a decent number of lines) that you have in at least two or three different locations in your `aaronize()` code. Use your judgement.

UCDAVIS
COMPUTER SCIENCE

[3] Don't forget that in GDB, a word is 32 bits and *not* 16 bits like it is in x86.