Fall '22 CIS 314 Assignment 4 – 100/100 points – Due Friday, 10/28, 11:59 PM

Please submit individual source files for coding exercises (see naming conventions below). Your code and answers need to be documented to the point that the graders can understand your thought process. Full credit will not be awarded if sufficient work is not shown.

1. [40] Consider the following C code:

```
long sum(long from, long to) {
    long result = 0;
    do {
        result += from;
        ++from;
    } while (from <= to);
    return result;
}
```

a. (10) Convert the above C code into a *goto* version by replacing the loop with a *goto* statement and label. Write your answer in a comment in code above your *sum* implementation for part b below. See B&O'H section 3.6.5 for examples.

b. (30) Implement your code from part a above in x86-64. Copy/paste the following code into https://www.onlinegdb.com/ (and ensure that the "Language" selector is set to "C"):

```
#include <stdio.h>

long sum(long from, long to) {
     // Declare and initialize result var - *do not modify*.
     long result = 0;

     // Ensure that argument *from* is in %rdi,
     // argument *to* is in %rsi, *result* is in %rax - *do not
     // modify*.
     __asm__ ("movq %0, %%rdi # from in rdi;" :: "r" ( from ));
     __asm__ ("movq %0, %%rsi # to in rsi;" :: "r" ( to ));
     __asm__ ("movq %0, %%rax # result in rax;" :: "r" ( result ));

     // Your x86-64 code goes below - comment each instruction...
     __asm__(
          // TODO - Replace the instruction below with add, compare,
          // jump instructions, labels, etc as necessary to implement
          // the loop.
          "movq %rdi, %rax;" // # For example, this sets result=from
     );

     // Ensure that *result* is in %rax for return - *do not modify*.
     __asm__ ("movq %%rax, %0 #result in rax;" : "=r" ( result ));
     return result;
}
```

```
int main() {
      printf("sum(1, 6): %ld\n", sum(1, 6));
      printf("sum(3, 5): %ld\n", sum(3, 5));
      printf("sum(5, 3): %ld\n", sum(5, 3));
}
```

First, run the above code to ensure that you get the following output:

```
sum(1, 6): 1
sum(3, 5): 3
sum(5, 3): 5
```

Note that I recommend using onlinegdb.com for this problem because this may or may not work in your VM depending on a few factors including host architecture and gcc version.

Next, notice that the following instruction simply sets the `result` variable equal to the value of the `from` parameter:

```
"movq %rdi, %rax;" // # For example, this sets result=from
```

Your task is to replace this instruction with a sequence of x86-64 instructions that will implement a loop summing the integers from `from` up to (and including) `to` in `result`. *DO NOT* modify any of the C code or any of the other x86-64 instructions!

Hint: I was able to accomplish this using 1 label and 4 instructions (2 *addq*, 1 *cmpq*, 1 *jle*).

Add a comment describing the purpose of each of your x86-64 instructions. Your x86-64 code must follow the register usage conventions outlined in B&O'H section 3.7.5.

Here are some test runs with the intended functionality:

```
sum(1, 6): 21
sum(3, 5): 12
sum(5, 3): 5
```

Copy/paste your solution into a C file named 4-1.c.

2. [20] Consider the following C code:

```
long fact(long x) {
  if (x <= 1) {
    return 1;
  }

  return x * fact(x - 1);
}
```

Copy/paste the above C code into a C file and write a main() function to test your code – for example:

```
fact(1): 1
fact(3): 6
fact(5): 120
```

Next, compile the fact() function to x86-64 using https://godbolt.org/ with x86-64 gcc 8.3 and the -Wall -Og -masm=att arguments.

In a code comment for the fact() function (in your C file), list all x86-64 instructions that modify the stack pointer. Also state the size of the stack frame.

Hint: don't forget about the return address!

Name your source file 4-2.c.

3. [40] The following C code transposes the elements of an N × N array:

```
#define N 4
typedef long array_t[N][N];

void transpose(array_t a) {
    for (long i = 0; i < N; ++i) {
        for (long j = 0; j < i; ++j) {
            long t1 = a[i][j];
            long t2 = a[j][i];
            a[i][j] = t2;
            a[j][i] = t1;
        }
    }
}
```

For example, the input:

```
{{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}, {13, 14, 15, 16}}
```

should be transposed as:

```
{{1, 5, 9, 13}, {2, 6, 10, 14}, {3, 7, 11, 15}, {4, 8, 12, 16}}
```

This C code is quite inefficient, primarily due to cost of the two-dimensional array lookups.

We can see this inefficiency in the assembly code; compiling the code to x86-64 using https://godbolt.org/ with x86-64 gcc 8.3 and the -Wall -Og -masm=att arguments generates the following x86-64 code for the **inner loop** of the procedure:

```
.L4:
        cmpq    %rax, %rdx
        jle     .L7
        movq    %rdx, %rsi
        salq    $5, %rsi
        addq    %rdi, %rsi
        movq    (%rsi,%rax,8), %r8
        movq    %rax, %rcx
        salq    $5, %rcx
        addq    %rdi, %rcx
        movq    (%rcx,%rdx,8), %r9
        movq    %r9, (%rsi,%rax,8)
        movq    %r8, (%rcx,%rdx,8)
        addq    $1, %rax
        jmp     .L4
```

Note the complex memory addressing (e.g., `(%rsi,%rax,8)` requires computing reg[%rsi] + 8 * reg[%rax]) and large number of arithmetic computations for a simple loop.

Hint: try compiling this yourself on godbolt!

In contrast, when compiled using https://godbolt.org/ with x86-64 gcc 8.3 and the -Wall -O1 -masm=att arguments, gcc generates the following x86-64 code for the **inner loop** of the function (the difference being the more-aggressive -O1 optimization setting); this code is significantly more performant due to its use of pointer arithmetic and dereferencing rather than array lookups to access the elements of the array:

```
.L3:
        movq    (%rax), %rcx
        movq    (%rdx), %rsi
        movq    %rsi, (%rax)
        movq    %rcx, (%rdx)
        addq    $8, %rax
        addq    $32, %rdx
        cmpq    %r9, %rax
        jne     .L3
```

Hint: try compiling this yourself on godbolt!

Copy the C code for the transpose() procedure above into a C file. Copy the optimized x86-64 code above (compiled with -O1) into a C file as a comment. Annotate each line of the x86-64 code in terms of *N*, *a*, *i*, *j*, *t1*, and *t2* from the original C code.

Next, write a new C procedure transposeOpt() that implements these optimizations using pointer arithmetic and dereferencing as demonstrated by the optimized assembly (e.g., *a, a += 8 rather than a[i][j] - see examples from lecture and B&O'H section 3.8.4). In other words, your

transposeOpt() should produce x86-64 code similar to the optimized x86-64 code above when compiled on https://godbolt.org/ with x86-64 gcc 8.3 and the -Wall -Og -masm=att arguments (*NOT* -O1). The point here is that we can ask the machine to do less work by writing efficient C code – we don't need to require the compiler to do this and that may not even be possible for a given software project!

Hint: declare a pointer to the *i*th row and a pointer to the *i*th column in the outer loop (e.g., *int *rp = &a[i][0]*); use pointer arithmetic to advance both pointers in the inner loop (again see examples from lecture and B&O'H section 3.8.4).

Also write an int main() function to test your transpose() and transposeOpt() procedures. Name your source file 4-3.c.

Hint: you can allocate the test array using:

```
array_t a = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}, {13, 14, 15, 16}};
```

Zip the source files and solution document (if applicable), name the .zip file <Your Full Name>Assignment4.zip (e.g., EricWillsAssignment4.zip), and upload the .zip file to Canvas (see Assignments section for submission link).