Fall ’19 CIS 314 Assignment 4 – 100/100 points – Due Friday, 11/1, 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. [30] Consider the following x86-64 code:

```
loop:
     movq  %rsi, %rcx
     movl  $1, %eax
     movl  $0, %edx
.L2:
     testq %rax, %rax
     je    .L4
     movq  %rax, %r8
     andq  %rdi, %r8
     orq   %r8, %rdx
     salq  %cl, %rax
     jmp   .L2
.L4:
     movq  %rdx, %rax
     ret
```

The code above was generated by compiling C code (with Arch gcc) that has the following overall form:

```
long loop(long a, long b) {
  long result = 0;
  for (long mask = ?; mask != ?; mask <<= ?) {
    result |= (? & ?);
  }
  return result;
}
```

Copy the above x86-64 code into a C file as a comment. Annotate each line of the x86-64 code in terms of *a*, *b*, *result*, and *mask*. Assume that this x86-64 code follows the register usage conventions outlined in B&O’H section 3.7.5 (it does). Then implement the above C function by filling in the blanks so that it’s functionally equivalent to the x86-64 code.

Here are some test runs:

```
loop(1, 5): 1
loop(2, 4): 0
loop(3, 3): 1
loop(4, 2): 4
loop(5, 1): 5
```

Also write a main() function to test your loop function.

Hint: the register conventions in B&O'H section 3.7.5 will inform which registers are holding *a* and *b*.

Hint: try compiling your C code to x86-64 using Arch gcc with the -S, -Og flags.

Name your source file 4-1.c.

2. [40] Consider the following C code:

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

Implement the *do-while* loop above in x86-64. Use the following as a framework:

```
long sum(long from, long to) {
     // Declar 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 two lines below with add, compare,
          // jump instructions, labels, etc as necessary to implement
          // the loop.
          "movq %rdi, %rax;" // # For example, this sets result=from
          "movq %rdi, %rdi;" // # This line is a no-op
     );

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

To be clear, you should only be modifying the x86-64 instructions in the *__asm__* call, just below the "TODO" comment.

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. FWIW, I was able to get this working with 4 instructions and 1 label.

Here are some test runs:

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

Also write a main() function to test your sum function. Compile and test your code using Arch gcc. Name your source file 4-2.c.

3. [30] 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;
        }
    }
}
```

This C code is quite inefficient, primarily due to the array lookups. Many of these array lookups can be removed by rewriting the loops to use pointer dereferencing and arithmetic (e.g., *a, a += 8 rather than a[i][j] - see B&O'H section 3.8.4 for examples).

When compiled with -O1, gcc (on Arch) generates the following x86-64 code for the **inner loop** of the function, which uses pointer dereferencing and arithmetic rather than array lookups:

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

Copy the x86-64 code above into a C file as a comment. Annotate each line of the x86-64code in terms of *N*, *a*, *i*, *j*, *t1*, and *t2*. Write a new C version of transpose that uses these

optimizations.

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}}
```

Also write a main() function to test your procedure. Name your source file 4-3.c.

Hint: declare a pointer to the *i*th row and a pointer to the *i*th column in the outer loop; use pointer arithmetic in the inner loop.

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).