THE UNIVERSITY of NEW ORLEANS
DEPARTMENT OF COMPUTER SCIENCE

CSCI 2467, Spring 2020
Class Activity: Understanding disassembled code
Friday, February 7

CSCI 2467 Staff: staff@2467.cs.uno.edu

1 Introduction

In this activity you will get practice reading assembly language code which has been *disassembled* – taken from an existing, compiled program. This "reverse-engineering" technique is especially useful for folks in the computer security field who are studying malware and software vulnerabilities. It is also an excellent way for anyone to gain a deeper understanding of how their programs are actually compiled and executed. (The questions are from CS:APP3e by Bryant and O'Hallaron, chapter 3.)

Below is a table which should be helpful. This is the "calling convention" for `x86-64` on Linux, which determines which function arguments go into which registers when a function is *called*.

If a function has only one argument, that argument will be stored in `rdi` . If there are two, then the first goes into `rdi` and the second into `rsi`, and so on for functions with more arguments.

`x86-64` calling convention	
Function argument	register
1^{st}	`rdi`
2^{nd}	`rsi`
3^{rd}	`rdx`
4^{th}	`rcx`
5^{th}	`r8`
6^{th}	`r9`
> 6	(stored on stack)

2 Machine-level arithmetic and logical operations

1. In the following C function `arith2()`, four expressions have been replaced with blanks:[1]

```
long arith2(long x, long y, long z)
{

        long t1 = ______________________;

        long t2 = ______________________;

        long t3 = ______________________;

        long t4 = ______________________;

        return t4;
}
```

When compiled using `gcc -S`, this function generates assembly instructions. Below are the instructions implementing the blank expressions: (Intel-style assembly)

```
arith2:
        or      rdi, rsi
        sar     rdi, 3
        not     rdi
        mov     rax, rdx
        sub     rax, rdi
        ret
```

Based on this assembly code, fill in the missing portions of the C code in the blanks above.

(Note: you may only use the symbolic variables `x`, `y`, `z`, `t1`, `t2`, `t3`, `t4` in your expressions above — *do not use register names.*)

[1]You can check your answer to this problem against the solution for Problem 3.10 on page 329 of CS:APP3e.

2. The `lea` instruction stands for "load effective address" and is designed to compute the memory address of an entry in an array or structure. However, `lea` can also be used by a compiler to compute arithmetic operations (addition and multiplication) in a single instruction, so it is often found in unexpected places.

   Consider the following code in which we have omitted the expression being computed:[2]

```
long scale2(long x, long y, long z) {

  long t = ________________________________

  return t;

}
```

   Compiling the actual function with GCC (using `gcc -S scale2.c -masm=intel -Og`) yields the following assembly code:

```
scale2:
        lea     rax, [rdi+rdi*4]
        lea     rax, [rax+rsi*2]
        lea     rax, [rax+rdx*8]
        ret
```

   Based on this assembly code, fill in the missing portion of the C function in the blank above.

   (Note: you may only use the symbolic variables `x`, `y`, and `z` in your expressions above — *do not use register names.*)

[2]You can check your answer to this problem against the solution for Problem 3.7 on page 328 of CS:APP3e.

3 Machine-level control flow

3. Consider the following assembly code:

```
fun1:
        cmp     rdi, rsi
        jge     .L3
        mov     rax, rdi
        ret
.L3:
        mov     rax, rsi
        ret
```

What C function could have been compiled to generate these instructions? (There is more than one correct answer.)

Fill in the three blanks below with valid C code (using variable names `a` and `b`):

```
long fun1(long a, long b) {


    if ( ________________________________________ )

            return ____________________ ;

    else
            return ____________________ ;
}
```