cs154: Introduction to Computer Systems
Spring 2020

Homework 3
(Assigned Apr 20)
Due Apr 27 11:59pm

Submit your work by adding and committing one file into the `hw3` directory of `CNETID-cs154-spr-20` svn repository. The file should be named either `hw3.txt` or `hw3.pdf` for answers written in a plain ASCII text file or PDF file, respectively. PDFs of scanned hand-written pages must not exceed 6 megabytes. No other file formats or filenames are acceptable, and no files besides `hw3.txt` or `hw3.pdf` will be graded. Not following directions will result in losing points.

(Example) (0 points)

From HW TA: This question is serving as an example. This question does not require you to answer anything. How to interpret each line of assembly code is written next to it.

Manually decompile the following assembly code into two short C functions `funcQ()` and `funcP()`, the prototypes of which are included as comments. You can ignore the `.globl` directives. Your code should not use local variables (new variables declared inside the functions); the original C code (before compilation) did not have any.

You decompile the assembly by using **your brain**, powered by your understanding of assembly (from lectures and Chapter 3). If asked a similar question during Exam 1 you will not be able to use a computer. There is no single correct representation of a C function in assembly.

```
        .globl  _funcQ
_funcQ:  # long funcQ(long x, long y), x is in %rdi as 1st argument, y is in %rsi as 2nd
        imulq   $3, %rsi        # Multiply %rsi by 3, so %rsi becomes 3*y
        imulq   $2, %rdi        # Multiply %rdi by 2, so %rdi becomes 2*x
        addq    %rdi, %rax      # Add %rdi, which is 2*x, to %rax
        addq    %rsi, %rax      # Add %rsi, which is 3*y, to %rax
        ret                     # Return, value in %rax is returned
                                # so here funcQ essentially is "return 2*x+3*y"
        .globl  _funcP
_funcP:  # long funcP(long r, long s, long t), r is in %rdi, s is in %rsi, t is in %rdx
        testq   %rsi, %rsi      # Testq %rsi, %rsi does not change %rsi at all, it just
                                # places %rsi for the next jle judgement
        jle     foo             # Jump to foo, if %rsi is <= 0, or s <= 0
                                # The code below is executed when s <= 0 is not fulfilled
        movq    %rdx, %rax      # put %rdx in %rax, so %rax stores t now
        movq    %rdi, %rdx      # put %rdi in %rdx, so %rdx stores r now
        movq    %rax, %rdi      # put %rax in %rdi, so %rdi stores t now
        callq   _funcQ          # call funcQ, and pass in %rdi (t) and %rsi (s)
                                # so essentially here we call funcQ(t,s)
                                # and once funcQ returns, the return value is stored in %rax
        addq    %rdx, %rax      # add %rdx (r) to %rax (funcQ(t,s))
        jmp     bar             # Jump to bar

foo:                            # The code below is executed when "jle foo" is fulfilled
                                # if (s <= 0)
        movq    %rdi, %rax      # put %rdi in %rax, so %rax stores r now
        movq    %rsi, %rdi      # put %rsi in %rdi, so %rdi stores s now
        movq    %rax, %rsi      # put %rax in %rsi, so %rsi stores r now
        callq   _funcQ          # call funcQ, and pass in %rdi(s) and %rsi(r)
                                # so essentially here we call funcQ(s,r)
                                # and once funcQ returns, the return value is stored in %rax
        addq    %rdx, %rax      # add %rdx (t) to %rax (funcQ(s,r))
bar:
        ret                     # return %rax
```

ANSWER:

The original code was:

```
long funcQ(long x, long y) {
  return 2*x + 3*y;
}

long funcP(long r, long s, long t) {
  if (s <= 0) {
    return t + funcQ(s, r);
  } else {
    return funcQ(t, s) + r;
  }
}
```

(1) (10 points + 2 bonus points)

Consider the following assembly code:

```
1          .globl   _loop
2  _loop:
3          xorq     %rax, %rax
4          movq     $5, %rdx
5  foo:
6          movq     %rax, %rcx
7          movq     %rdx, %rax
8          andq     %rdi, %rax
9          orq      %rcx, %rax
10         shlq     %rsi, %rdx
11         testq    %rdx, %rdx
12         jne      foo
13         ret
```

The assembly code was generated by compiling C code with the following overall form:

```
long loop(long x, long n) {
  long result = ___1___;
  long mask;
  for (mask = ___2___; mask ___3___; mask = ___4___) {
    result ___5___;
  }
  return result;
}
```

Your task is to fill in the missing parts of the C code to get a program equivalent to the generated assembly code. Recall that the result of the function is returned in register `%rax`. You will find it helpful to examine the assembly code before, during, and after the loop to form a consistent mapping between the registers and the program variables. The clarity of your answers below may be improved by mentioning assembly line numbers.

A. Which registers hold program values `x`, `n`, `result`, and `mask`?
B. What are the initial values of `result` and `mask`?
C. What is the test condition for `mask`?
D. How does `mask` get updated?
E. How does `result` get updated?
(Bonus) F. Fill in all the missing parts of the C code, by providing the entire contents of the `__1__`, `__2__`, etc blanks.

(2) (5 points + 5 bonus points)

Consider the following C source code, in which the constants `R`, `S`, and `T` have already been declared through `#defines` (e.g. "`#define R 2`"):

```
int A[R][S][T];
long lkup(long i, long j, long k, int *dest) {
  *dest = A[i][j][k];
  return sizeof(A);
}
```

A. Generalize Equation (3.1) of the textbook (page 236 in Ed. 2, page 258 in Ed. 3) to give an expression for the *address* `&(A[i][j][k])` of element `A[i][j][k]` in terms of $x_{\mathtt{A}} = \&(\mathtt{A}[0][0][0])$, $L =$ `sizeof(int)`, indices i, j, k, and array sizes R, S, T. Your answer may not require all these variables ($x_{\mathtt{A}}$, i, R, etc.), but it must include L.

(Bonus) B. When compiling the above C code to assembly, the result includes:

```
1           .globl  lkup
2   lkup:
3           movq    %rsi, %rax
4           leaq    (%rax,%rax,5), %r8
5           imulq   $60, %rdi, %rax
6           addq    %r8, %rax
7           addq    %rdx, %rax
8           movl    A(,%rax,4), %edx
9           movq    %rcx, %rax
10          movl    %edx, (%rax)
11          movq    $1440, %rax
12          ret
13          .comm   A,1440,64
```

From the assembly code, determine the values of R, S, and T. To receive full credit you must explain your answer with reference to the assembly line numbers. Be concise; you should not need more than roughly 100 words.

The three-operand form of `imul` (line 5) multiplies the value of the two source operands `$60` and `%rdi` and stores it in the destination operand `%rax`. It can be used if the first operand is a constant.

The "`A`" in "`movl A(,%rax,4), %edx`" on line 8 should be read as an immediate that has a symbolic rather than an absolute value. Just like the targets of jump instructions have symbolic names that are turned into numeric values later (e.g. "`jle foo`"), the address of array `A` can appear in the assembly. The last "`.comm`" line is an assembler directive (rather than an assembly instruction) which indicates the size (1440) and alignment restrictions (64) of `A`.