CS33 Homework Assignment 4 Solutions

Fall 2020

1. C currently does not support a 128-bit integer data type. In this problem, you're going to do some of the work to implement such a type. We'll stick with unsigned integers for now.
   a. We need an appropriate *typedef*. Define a type, *ulong128_t*, that allows us to easily access the low-order 64 bits and the high-order 64 bits. Keep in mind that x86-64 is a little-endian architecture.

      Answer:

```
typedef struct {
        unsigned long low;
        unsigned long high;
} ulong128_t;
```

   b. If we're going to make use of this type, we need, among many other things, an implementation of multiplication. Produce an implementation, in x86-64 assembler, of Mult128:

```
void Mult128(ulong128_t *op1, ulong128_t *op2, ulong128_t *res);
```

      On return, *res* should point to a *ulong128_t* containing the product of **op1* and **op2*. You should expect your answer to use around 12 instructions, including the *ret* at the end. Some hints:

      i. You might first write an approximate version of Mult128 in C, compile it with the –S (which tells gcc to produce assembler code) and –O1 flags, and work with the gcc-produced assembler code (which will be in a *.s* file)
      ii. The product of (a + b) and (c + d) is ac + ad + bc + bd. (You probably knew this!)
      iii. The portion of the result that's greater than or equal to 2^{128} can be ignored, since we're concerned only with the low-order 128 bits of the product.
      iv. The unsigned multiply instruction, *mulq*, produces a 128-bit result from two 64-bit operands. The multiplicand is in %rax (and thus isn't mentioned explicitly as an operand). The multiplier is given as the only operand to the instruction. The high-order 64 bits of the result will be put in %rdx (caution, this register also holds the third argument to the function!); the low-order 64 bits of the result will be put in %rax.

      Answer:

```
Mult128:
      movq  %rdx, %r10        # save address of third param, since rdx is needed for
                              # 128-bit mult
      movq  (%rdi), %rax      # get low-order 64 bits of op1
```

```
mulq  (%rsi)            # multiply times low order 64 bits of op2
movq  %rax, (%r10)      # save low-order 64 bits of result
movq  %rdx, 8(%r10)     # save high-order 64 bits of result
movq  8(%rdi), %rax     # get high-order 64 bits of op1
mulq  (%rsi)            # multiply times low-order 64 bits of op2
addq  %rax, 8(%r10)     # add low-order 64 bits of result to high-order 64 bits
                        # of prev result
movq  (%rdi), %rax      # get low-order 64 bits of op1
mulq  8(%rsi)           # multiply times high-order 64 bits of op2
addq  %rax, 8(%r10)     # add low-order 64 bits of result to high-order 64 bits
                        # of prev result
ret                     # return to caller
```