UCLA Computer Science 33 (Spring 2015)
Midterm
108 minutes total, open book, open notes
Questions are equally weighted (12 minutes each)

Name:_____ Student ID:_____

1	2	3	4	5	6	7	8	9	sum

1 (12 minutes). Which integer constants can a single x86 leal instruction can multiply an arbitrary integer N by? The idea is that one puts N into a register, executes the leal instruction, and the bottom 32 bits of N*K will be put into some other register, where K is a constant. For which values of K can this be done? For each such value, show an leal instruction that implement that value.

With “leal (, N, c), D” where N is a register and c is 1, 2, 4, 8. you can produce 1*N, 2*N, 4*N, and 8*N. With “leal(N, N, c), D”, you can also produce 3*N, 5*N, and 9*N when c = 2, 4, and 8, respectively. K = 1, 2, 3, 4, 5, 8, 9

2 (12 minutes). On the x86-64, what's the fastest way to reverse each 8-bit byte in a 64-bit unsigned integer? For example, given the input integer 0x0123456789abcdef, we want to compute 0x80c4a2e691d5b3f7; this is because 0x01 is binary 00000001 and reversing it yields binary 10000000 which is 0x80, and similarly the bit-reverse of 0x23 is 0xc4, and so forth until the bit-reverse of 0xef is 0xf7. Write the code in C, and estimate how many machine instructions will be generated (justify your estimate).

```
long long reverseBits(long long x)
{
  unsigned long long m1 = 0xF0F0F0F0F0F0F0F0;
  unsigned long long m2 = 0xCCCCCCCCCCCCCCCC;
  unsigned long long m3 = 0xAAAAAAAAAAAAAAAA;
  unsigned long long l = x;
  l = (l&m1)>>4 | (l<<4)&m1;
  l = (l&m2)>>2 | (l<<2)&m2;
  l = (l&m3)>>1 | (l<<1)&m3;
  return l;
}
```

Approx 15. operations.

3 (12 minutes). On the x86, there is no 'pushl %eip' instruction. Suppose you want to push the instruction pointer onto the stack anyway. What's the best way to do it? If your method takes three instructions A, B, C, the value pushed onto the stack should be the address of D, the next instruction after C.

“pushl %eip” can be implemented by using the “call” instruction where the operand is the address of the next instruction. Ex:

```
0x10: call 0x15
0x15: <next_instruction>
```

In this example, “call 0x15” is equivalent to:

```
%esp = %esp – 4
(%esp) = 0x15 (0x15, the address of the next instruction)
%eip = %eip + 5 (0x10 + 0x5)
```

4 (12 minutes). Explain two different methods that GDB can implement its 'fin' command (which finishes execution of the current function), one method with hardware breakpoints and one without. For each method, say what happens if the current function calls another function via tail recursion.

The basic idea was to recall the methods by which breakpoints are implemented in gdb.

Software: Replace the instruction that we would like to break at with an INT or INT3 (if the replaced instruction is only 1 byte).

Hardware: Registers DR0-DR4 are special registers for which if the the instruction at the address in DRX is read/written/executed (depending on the status of DR7), break.

Using these methods, we can set a breakpoint for when the function completes. Since we want to break AFTER the function completes ("finishes execution of the current function", setting the breakpoint at "ret" is incorrect. We will need to set the breakpoint at the return address of the current stack frame (generally accessible via 4(%ebp)). In the case of tail recursion where the current function calls another function, the new function will essentially take over the current stack frame, which means its return value will be the same. As a result, using 4(%ebp) will work in this case as well.

5 (12 minutes). Consider the following C functions and their translations to x86 code.

```
int f (int *p, long *q) {
  ++*p;
  ++*q;
  return *p;
}

int g (int *p, char *q) {
  ++*p;
  ++*q;
  return *p;
}
```

```
 f: movl  4(%esp), %ecx
    movl  8(%esp), %edx
    movl  (%ecx), %eax
    addl  $1, %eax
    movl  %eax, (%ecx)
    addl  $1, (%edx)
    ret
```

```
g: movl  4(%esp), %ecx
   movl  8(%esp), %edx
   movl  (%ecx), %eax
   addb  $1, (%edx)
   addl  $1, %eax
   movl  %eax, (%ecx)
   ret
```

There's a compiler bug: one of these functions is translated incorrectly, and the other one is OK. Identify the bug, and explain why the other function is translated correctly even though one might naively think that its translation has a similar bug.

g() has a bug with when there is aliasing. If p == q, the result of g() should be that *p = *p + 2. However, g() will first pull the value of *p into register %eax. Then, it increments *q by one (which is the same as p in the aliasing case) and saves the value to memory. Then, it increments *p which is stored in %eax. At this point, %eax contains *p + 1. Finally, the value in %eax is stored back into memory, which means that the resulting value that is stored into memory *p + 1 instead of *p + 2. Additionally, %eax is used as the return value, which is also *p + 1 instead of *p + 2.

f() also has a bug, in that it doesn’t return the correct value. In the case where p == q, the data in memory is modified correctly to become *p + 2, but the value returned is only *p + 1.

6 (12 minutes). Suppose we have allocated memory locations 0xffff0000 through 0xffffffff for the stack, and we are worried that our x86 program might overflow the stack. We decide to institute the ironclad rule that if a function ever attempts to grow the stack past the allocated bounds, the function immediately stops what it's doing and returns 0, thus shrinking the stack. Explain the problems you see in implementing this rule. Don't worry about the effects of this rule on the user program; worry only about implementing the rule correctly.

The first key point is that we can't really implement this strictly in C code. Recall that the maintenance of the stack can occur between lines of C code (for example, simply calling a function will allocate space for the return address, the old %ebp, and possibly for the new stack frame). We need the compiler's help. First, for each instruction that changes the stack, the compiler will have to insert a check. Ex:

```
subl 0x10, %esp
```

becomes:

```
movl %esp, %ebx
subl 0x10, %ebx
cmpl 0xFFFF0000, %ebx
jb <STACK EXCEEDED>
movl %ebp, %esp
```

Additionally, there are two other special cases that would need to be checked:

- Subtraction underflow: If you attempt to allocate so much space on the stack that %esp underflows back to an allowable address.
- Stack allocation that is not known during compile time.

Also another thing that was worth points: pointing out that it would be slow.

7 (12 minutes). Give C source code that corresponds to the following x86-64 assembly language code. Explain briefly and at a high level what useful thing the function does.

```
[1]  sub: movq   %rdi, %rdx
[2]       subq   %rsi, %rdx
[3]       xorq   %rdi, %rsi
[4]       xorq   %rdi, %rdx
[5]       movq   %rdx, %rax
[6]       andq   %rsi, %rax
[7]       shrq   $63, %rax
          ret
```

%rdi has arg1, %rsi has arg2
1. %rdx = arg1
2. %rdx = arg1 - arg2

3. %rsi = arg1 ^ arg2
4. %rdx = arg1 ^ (arg1 - arg2)
5. %rax = arg1 ^ (arg1 - arg2)
6. %rax = (arg1 ^ (arg1 – arg2)) & (arg1 ^ arg2)
7. %rax = MSB ((arg1 ^ (arg1 – arg2)) & (arg1 ^ arg2))

```
long long sub(long long arg1, long long arg2)
{
  return ((arg1 ^ (arg1 - arg2)) & (arg1 ^ arg2)) < 0;
}
```

arg1 ^ arg2 is a number whose MSB is 1 (aka a number that is < 0) if the arguments are of different sign. arg1 ^ (arg1 - arg2) is a number whose MSB is 1 if arg1 and (arg1 - arg2) are of a different sign. The return value retuns these two MSBs and'd together.

Ex. arg1 = 0x8000000000000000, arg2 = 0x1.
MSB(0x80...00 ^ 0x1) = 1
MSB(0x80...00 ^ (0x80...00 - 1)) = MSB(0x80...00 ^ (0x7F...FF)) = 1
sub returns 1.

This is another way of checking for signed overflow.

8 (12 minutes). Match each of the following C source functions to each of the following assembly-language functions. A "match" means that the assembly-language code properly implements the C code. For example, if the C function 'f' is implemented by the assembly-language implementation 'B', write "f=B'.

```
int a(int x) { while (x & 1) x>>=1; return x; }
int b(int x) { while (x & 3) x>>=1; return x; }
int c(int x, int y)
  { return x / y - (x % y < 0); }
int d(int x, int y)
  { return x % y + (x % y < 0 ? y : 0); }
int e(unsigned x, unsigned y)
  { return (x + y < x) ^ ((int) y < 0); }
int f(int a, int b, int c) { return a ? b : c; }
int g(int a, int b, int c)
```

```
  { return a ? b | c : b & c; }
int h(unsigned x, unsigned y)
  { return x - x / y * y; }
int i(int x, int y) { return x - x / y * y; }
int j(int x) { return -~x; }
int k(int x) { return ~-x; }
int l(int x) { return x+~x; }
```

(continued on next page)

(continued from previous page)

```
A:    movl  4(%esp), %eax
      testb $3, %al
      je    .L2
.L3:  sarl  %eax
      testb $3, %al
      jne   .L3
.L2:  ret

B:    movl  8(%esp), %eax
      movl  12(%esp), %edx
      movl  %eax, %ecx
      orl   %edx, %ecx
      andl  %eax, %edx
      movl  4(%esp), %eax
      testl %eax, %eax
      movl  %ecx, %eax
      cmove %edx, %eax
      ret

C:    movl  4(%esp), %eax
      testl %eax, %eax
      movl  12(%esp), %eax
      cmovne      8(%esp), %eax
      ret

D:    movl  4(%esp), %eax
      testb $1, %al
      je    .L16
.L17: sarl  %eax
      testb $1, %al
      jne   .L17
```

```
.L16: ret

E:    movl  4(%esp), %eax
      cltd
      idivl 8(%esp)
      shrl  $31, %edx
      subl  %edx, %eax
      ret

F:    movl  4(%esp), %eax
      subl  $1, %eax
      ret

G:    movl  $-1, %eax
      ret

H:    movl  4(%esp), %eax
      xorl  %edx, %edx
      divl  8(%esp)
      movl  %edx, %eax
      ret

I:    movl  4(%esp), %eax
      addl  $1, %eax
      ret

J:    movl  8(%esp), %edx
      movl  %edx, %eax
      addl  4(%esp), %eax
      setc  %al
      shrl  $31, %edx
      xorl  %eax, %edx
      movzbl      %dl, %eax
      ret

K:    movl  4(%esp), %eax
      cltd
      idivl 8(%esp)
      movl  %edx, %eax
      ret

L:    movl  4(%esp), %eax
      movl  8(%esp), %ecx
```

```
        cltd
        idivl %ecx
        movl  $0, %eax
        testl %edx, %edx
        cmovns      %eax, %ecx
        leal  (%edx,%ecx), %eax
        ret
```

a=D
b=A
c=E
d=L
e=J
f=C
g=B
h=H
i=K
j=I
k=F
l=G

9 (12 minutes). Consider the following program:

```
1       unsigned ack (unsigned m, unsigned n) {
2         if (m == 0)
3           return n + 1;
4         if (n == 0)
5           return ack (m - 1, 1);
6         return ack (m - 1,
7                     ack (m,
8                        n - 1));
9       }
```

(continued in next column)

and the following assembly-language implementation as reported by gcc -S:

```
1       ack:
2             pushl %ebx                          1
3             subl  $8, %esp                      1
```

```
 4            movl  16(%esp), %ebx              1
 5            movl  20(%esp), %eax              1
 6            testl %ebx, %ebx                  2
 7            je    .L2                         2
 8            subl  $1, %ebx                    5,6
 9            jmp   .L4                         2,4
10       .L7:
11            testl %ebx, %ebx                  2
12            movl  $1, %eax                    3
13            leal  -1(%ebx), %edx              6
14            je    .L2                         2
15       .L6:
16            movl  %edx, %ebx                  6
17       .L4:
18            testl %eax, %eax                  4
19            leal  1(%ebx), %edx               6
20            je    .L7                         4
21            subl  $8, %esp                    7,8
22            subl  $1, %eax                    8
23            pushl %eax                        8
24            pushl %edx                        7
25            call  ack                         7
26            addl  $16, %esp                   7
27            testl %ebx, %ebx                  2
28            leal  -1(%ebx), %edx              5,6
29            jne   .L6                         2
30       .L2:
31            addl  $8, %esp                    3,5
32            addl  $1, %eax                    3,5
33            popl  %ebx                        3,5
34            ret                               3,5
```

For each instruction in the implementation, identify the corresponding source-code line number. If an instruction corresponds to two or more source-code line numbers, write them all down and explain.

Note: For some of these instructions, there may be alternative lines that could be considered acceptable answers.