Exam 1, CSCI 210, Spring 2004

Name: _____

1. [10 pts] How would the behavior of the following two Unix shell commands differ?

```
unix% grep open f > wc
unix% grep open f | wc
```

2. [5 pts] Suppose we are using an 7-bit floating-point representation with 3 bits for the excess-3 exponent and 3 bits for the mantissa, supporting the denormalized and the nonnumeric cases.

What bit pattern represents $0.125_{(10)}$?

3. [30 pts] Translate the following C function into a subroutine in the x86 assembly language. The entry and exit templates are already provided. (I recommend using the callee-save register ebx to hold n.)

```
int lastOf(int n) {
    int k;

    while(n != 0) {
        scanf("%d", &k);
        n--;
    }
    return k;
}
```

```
.section .data
fmt:        .string ”%d”
.section .text
lastOf:     pushl %ebp                  # entry template
            movl %esp, %ebp

            movl %ebp, %esp             # exit tempate
            popl %ebp
            ret
```

4. [10 pts] Suppose that *gcc* sees the following `while` loop.

```
while(ecx < edx) {
    eax += ecx;
    ecx++;
}
```

In compiling this code for the x86 CPU, *gcc* will choose version (b.) of the following two alternatives, even though it is less intuitive and longer (six versus five instructions).

a.

```
loop:   cmpl %edx, %ecx
        jge done
        addl %ecx, %eax
        incl %ecx
        jmp loop
done:
```

b.

```
        cmpl %edx, %ecx
        jge done
loop:   addl %ecx, %eax
        incl %ecx
        cmpl %edx, %ecx
        jl loop
done:
```

Explain why the compiler prefers (b.).

5. [15 pts] Explain the optimization technique of *strength reduction*. In what situations does it apply? How does a compiler transform code using the technique? Feel free to give an example before and after the optimization; you might write the example in C or in x86 assembly.

6. [10 pts] Consider the following code, using the `findGoldbach()` function you wrote for Lab 1.

a.

```
int main() {
    int n;

    scanf("%d", &n);
    if(findGoldbach(n) != 0) {
        printf("%d %d\n", findGoldbach(n),
            n - findGoldbach(n));
    } else {
        printf("no pair found\n");
    }
    return 0;
}
```

b.

```
int main() {
    int n, p;

    scanf("%d", &n);
    p = findGoldbach(n);
    if(p != 0) {
        printf("%d %d\n", p, n - p);
    } else {
        printf("no pair found\n");
    }
    return 0;
}
```

Both alternatives work the same. But version (a.) is considerably slower, since `findGoldbach()` is quite slow, and (b.) uses `findGoldbach()` only once, whereas (a.) may call it as many as three times.

Given (a.), *gcc* will *not* choose to optimize it by transforming it to (b.). despite the fact that (b.) works identically and is much faster. Explain why it does not perform this optimization.