Andrew login ID:______________________________________

Full Name:______________________________________

CS 15-213, Fall 2004

Exam 1

Tuesday October 12, 2004

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name and Andrew login ID on the front.
- Write your answers in the space provided below the problem. If you make a mess, clearly indicate your final answer.
- The exam has a maximum score of 70 points.
- The problems are of varying difficulty. The point value of each problem is indicated. Pile up the easy points quickly and then come back to the harder problems.
- This exam is OPEN BOOK. You may use any books or notes you like. No electronic devices are allowed. Good luck!

1 (12):	
2 (10):	
3 (15):	
4 (8):	
5 (10):	
6 (8):	
7 (7):	
TOTAL (70):	

Problem 1. (12 points):

Consider a 9-bit variant of the IEEE floating point format as follows:

- Sign bit
- 4-bit exponent with a bias of -7.
- 4-bit significand

All of the rules for IEEE (normalized, denormalized, special numbers, etc.) apply.
Fill in the numeric value represented by the following bit patterns. Write your numbers in either fractional form (e.g., $-11/8$) or decimal form (e.g., 1.375).

Bit Pattern	Numeric Value
`1 0000 0000`	
`0 0000 1111`	
`1 0001 0001`	
`0 1010 1010`	
`1 1110 1111`	
`0 1111 0000`	

Problem 2. (10 points):

You are given the following C code to compute integer absolute value:

```
int abs(int x)
{
  return x < 0 ? -x : x;
}
```

You've concerned, however, that mispredicted branches cause your machine to run slowly. So, knowing that your machine uses a two's complement representation, you try the following (recall that `sizeof(int)` returns the number of bytes in an `int`):

```
int opt_abs(int x)
{
  int mask = x >> (sizeof(int)*8-1);
  int comp = x ^ mask;
  return comp;
}
```

A. What bit pattern does `mask` have, as a function of `x`?

B. What numeric value does `mask` have, as a function of `x`?

C. For what values of `x` do functions `abs` and `opt_abs` return identical results?

D. For the cases where they produce different results, how are the two results related?

.

E. Show that with the addition of just one single arithmetic operation (any C operation is allowed) that you can fix `opt_abs`. Show your modifications on the original code.

F. Are there any values of `x` such that `abs` returns a value that is *not* greater than 0? Which value(s)?

Problem 3. (15 points):

This question will test your ability to reconstruct C code from the assembled output. On the opposing page, there is asm code for a routine called `bunny`. It comes from a C routine with the following outline.
Don't fill in the outline yet.

```
static int bunny(int l, int r, int *A) {
    int x = ______;
    int i = _______;
    int j = _______;
    while(_______) {
        do  j--;  while(_______);
        do  i++;  while(_______);
        if(_______) {
            int t = A[i];
            A[i] = A[j];
            A[j] = t;
        }
    }
    return ______;
}
```

A. (3 points): Fill in the following table of register usage. Use the variable names from the outline. If a register gets used to store two different things, just list both of them. I've filled in two blanks to show examples. This will help you understand the code; do this before part C.

Register	Variable
`%eax`	
`%ebx`	
`%ecx`	
`%edx`	
`%esi`	
`%edi`	
`%esp`	
`%ebp`	

B. (3 points): Why does `bunny` push `%edi`, `%esi`, and `%ebx` on to the stack?

```
bunny:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %edi
        pushl   %esi
        pushl   %ebx
        movl    8(%ebp), %eax
        movl    16(%ebp), %esi
        movl    (%esi,%eax,4), %edi
        leal    -1(%eax), %ecx
        movl    12(%ebp), %ebx
        incl    %ebx
        cmpl    %ebx, %ecx
        jge     .L3
.L16:
        decl    %ebx
        cmpl    %edi, (%esi,%ebx,4)
        jg      .L16
.L7:
        incl    %ecx
        cmpl    %edi, (%esi,%ecx,4)
        jl      .L7
        cmpl    %ebx, %ecx
        jge     .L3
        movl    (%esi,%ecx,4), %edx
        movl    (%esi,%ebx,4), %eax
        movl    %eax, (%esi,%ecx,4)
        movl    %edx, (%esi,%ebx,4)
        jmp     .L16
.L3:
        movl    %ebx, %eax
        popl    %ebx
        popl    %esi
        popl    %edi
        popl    %ebp
        ret
```

C. (5 points): Fill in the blanks on the outline (on the previous page).

D. (4 points): Look at `draft_horse` and write out the control flow structure. As an example, the control flow structure of `bunny` would be:

```
bunny() {
    while() {
        while() { }
        while() { }
        if() { }
    }
    return;
}
```

I want to know about any `if`, `while`, function calls, and returns, but I don't care about anything else. Do not use `goto`.

```
draft_horse() {




}
```

E. (bragging rights): What algorithm is this code implementing?

```
draft_horse:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $28, %esp
        movl    %ebx, -12(%ebp)
        movl    %esi, -8(%ebp)
        movl    %edi, -4(%ebp)
        movl    8(%ebp), %ebx
        movl    12(%ebp), %esi
        movl    16(%ebp), %edi
        cmpl    %esi, %ebx
        jge     .L17
        movl    %edi, 8(%esp)
        movl    %esi, 4(%esp)
        movl    %ebx, (%esp)
        call    bunny
        movl    %eax, -16(%ebp)
        movl    %edi, 8(%esp)
        movl    %eax, 4(%esp)
        movl    %ebx, (%esp)
        call    draft_horse
        movl    %edi, 8(%esp)
        movl    %esi, 4(%esp)
        movl    -16(%ebp), %eax
        incl    %eax
        movl    %eax, (%esp)
        call    draft_horse
.L17:
        movl    -12(%ebp), %ebx
        movl    -8(%ebp), %esi
        movl    -4(%ebp), %edi
        movl    %ebp, %esp
        popl    %ebp
        ret
```

Problem 4. (8 points):

Given the following code:

```
int
calcHash(char *str) {
    unsigned int i;
    int hash = 0;
    for(i = 0; i < strlen(str); i++) {
        hash += str[i] * 32 + i;
    }
    return hash;
}
```

A savvy programmer has re-written it to read as follows:

```
int
calcHash(char *str) {
    unsigned int i, len = strlen(str);
    int hashA = 0, hashB = 0;
    for(i = 0; i < len - 1; i += 2) {
        hashA += (str[i] << 5) + i;
        hashB += (str[i + 1] << 5) + i + 1;
    }
    if(i == len - 1) {
        hashA += (str[i] << 5) + i;
    }
    return hashA + hashB;
}
```

Answer the questions about this code on the following page.

A. Explain in one or two sentences how moving strlen(str) from line 6 to line 3 improves the performance of this code.

Would this transformation would preserve the exact functionality of the original code? Explain.

B. Explain in one or two sentences how creating two separate add instructions on lines 7 and 8 improves the performance of this code.

Is this an optimization that a compiler could perform? Why or why not?

C. Point out one other optimization that was added to this code and explain how it improves performance.

Problem 5. (10 points):

This problem will test your knowledge of stack discipline and byte ordering. As in Lab 3, you will perform a buffer overflow attack on the following C code. Your goal is to call `secret` and make the program execute the infinite loop.

```
int read_string() {
  char buf[8];
  scanf("%s", &buf);
  return buf[1];
}

int main() {
  printf("0x%x\n", read_string());
  return 0;
}

void secret(int arg) {
  if(arg == 0x15213)
    while(1);
  exit(-1);
}
```

Things to keep in mind while working on this problem.

- `scanf("%s", buf)` reads an input string from stdin and stores it at address `buf` (including the terminating '`\0`' character). It does **not** check the size of the destination buffer.
- Linux/x86 machines are Little Endian.

A. Suppose we gave the program the codes `69 6c 75 76 32 31 33`. Using the stack template below, indicate where `%ebp` points to, and fill in the stack with the values that were just read in **after** the call to `scanf`. **Addresses increase from left to right**.

```
          |<---------------- buf ---------------->|
     +----+----+----+----+----+----+----+----+----+----+----+----+----+
     |    |    |    |    |    |    |    |    |    |    |    |    |    |
     +----+----+----+----+----+----+----+----+----+----+----+----+----+
```

B. Using the assembly code on the next page, fill in the stack template below with codes that will cause the program to execute `secret` and make it believe that `arg` has the value 0x15213. **Addresses increase from left to right and from top to bottom.**

```
          |<---------------- buf ---------------->|
     +----+----+----+----+----+----+----+----+----+----+----+----+----+
     |    |    |    |    |    |    |    |    |    |    |    |    |    |
     +----+----+----+----+----+----+----+----+----+----+----+----+----+

     +----+----+----+----+----+----+----+----+----+----+----+----+----+
     |    |    |    |    |    |    |    |    |    |    |    |    |    |
     +----+----+----+----+----+----+----+----+----+----+----+----+----+
```

C. What is the value of `%ebp` when the instruction at `0x80483e2` is executed?

```
08048398 <read_string>:
 8048398: 55                    push   %ebp
 8048399: 89 e5                 mov    %esp,%ebp
 804839b: 83 ec 18              sub    $0x18,%esp
 804839e: 8d 45 f8              lea    0xfffffff8(%ebp),%eax
 80483a1: 89 44 24 04           mov    %eax,0x4(%esp,1)
 80483a5: c7 04 24 74 84 04 08  movl   $0x8048474,(%esp,1)
 80483ac: e8 ef fe ff ff        call   80482a0 <scanf>
 80483b1: 8b 45 fc              mov    0xfffffffc(%ebp),%eax
 80483b4: c9                    leave
 80483b5: c3                    ret

080483b6 <main>:
 80483b6: 55                    push   %ebp
 80483b7: 89 e5                 mov    %esp,%ebp
 80483b9: 83 ec 08              sub    $0x8,%esp
 80483bc: 83 e4 f0              and    $0xfffffff0,%esp
 80483bf: b8 00 00 00 00        mov    $0x0,%eax
 80483c4: 29 c4                 sub    %eax,%esp
 80483c6: e8 cd ff ff ff        call   8048398 <read_string>
 80483cb: 89 44 24 04           mov    %eax,0x4(%esp,1)
 80483cf: c7 04 24 77 84 04 08  movl   $0x8048477,(%esp,1)
 80483d6: e8 e5 fe ff ff        call   80482c0 <printf>
 80483db: b8 00 00 00 00        mov    $0x0,%eax
 80483e0: c9                    leave
 80483e1: c3                    ret

080483e2 <secret>:
 80483e2: 55                    push   %ebp
 80483e3: 89 e5                 mov    %esp,%ebp
 80483e5: 83 ec 08              sub    $0x8,%esp
 80483e8: 81 7d 08 13 52 01 00  cmpl   $0x15213,0x8(%ebp)
 80483ef: 75 02                 jne    80483f3 <secret+0x11>
 80483f1: eb fe                 jmp    80483f1 <secret+0xf>
 80483f3: c7 04 24 ff ff ff ff  movl   $0xffffffff,(%esp,1)
 80483fa: e8 d1 fe ff ff        call   80482d0 <exit>
 80483ff: 90                    nop
```

Problem 6. (8 points):

Consider the following C declarations:

```
typedef union{              typedef struct {
    char state[3];              short WID;
    char cncode[4];             char name[5];
    int  index;                 Location address;
} Location;                     double balance;
                                char domestic;
                                char *note;
                            } Warehouse;
```

A. Using the templates below (allowing a maximum of 28 bytes), indicate the allocation of data for structs of type `Warehouse`. Mark off and label the areas for each individual element (arrays may be labeled as a single element). **Cross hatch the parts that are allocated, but not used, and be sure to clearly indicate the end of the structure. Assume the Linux alignment rules discussed in class.**

`Warehouse:`

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                                                                   |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

B. How would you define the `Compact` structure to minimize the number of bytes allocated for the structure using the same fields as the `Warehouse` structure?

```
typedef struct {




 } Compact;
```

C. What is the value of `sizeof(Compact)`?

D. Now consider the IA-32 Windows alignment convention. How would you define the `Win_Compact` structure to minimize the number of bytes allocated for the structure using the same fields as the `Warehouse` structure?

```
typedef struct {





} Win_Compact;
```

E. Consider the following C code fragment:

```
Warehouse company1;

strcpy(company1.address.cncode, "CAN"); /* 'C' = 43, 'A' = 41, 'N' = 4e */
```

After this code has been executed,

`company1.address.index = 0x____________`

Assume that this code is running on a little-endian machine such as a Linux/x86 machine. You must give your answer in hexadecimal format.

Problem 7. (7 points):

Answer **true** or **false** for each of the statements below. For full credit your answer must be correct and you must write the entire word (either **true** or **false** in the answer space. You will be given $+1.0$ point for each correct answer, and -0.5 points for each incorrect answer, so wild guessing doesn't pay.

1. `int a[10], x;`
   `x = &(a[5]) - &(a[1]);`
   `x` is always 4.

2. All Intel IA-32 instructions have the same length.

3. Processors with longer pipelines tend to make branch instructions more costly.

4. To swap the values of two variables in C always requires using some kind of temporary storage location.

5. In C, the variable `m` is declared as: `int m[100][100]`. Depending on the CPU architecture, the address for m[9][99] can sometimes be greater than the address for m[10][0].

6. IEEE floating point numbers are evenly distributed for values `0.5 < x < 1.0`.

7. IEEE floating point operations always round toward the nearest FP number.