Andrew login ID:_______________________________

Full Name:_______________________________

CS 15-213, Spring 2004

Exam 1

February 26, 2004

Instructions:

- Make sure that your exam is not missing any sheets (there should be 15), 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 80 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 (9):	
2 (14):	
3 (12):	
4 (8):	
5 (12):	
6 (12):	
7 (6):	
8 (7):	
TOTAL (80):	

Problem 1. (9 points):

Assume we are running code on a 10-bit machine using two's complement arithmetic for signed integers. Short integers are encoded using 5 bits. Sign extension is performed whenever a short is casted to an int. For this problem, assume that all shift operations are arithmetic. Fill in the empty boxes in the table below.

```
int i = -42;
unsigned ui = i;
short s = -7;
unsigned short us = s;
```

Note: You need not fill in entries marked with "–". TMax denotes the largest positive two's complement number and TMin denotes the minimum negative two's complement number. Finally, you must use hexidecimal notation for your answers in the "Hex Representation" column, failure to do so will result in being marked incorrect on that portion of the question.

Expression	Decimal Representation	Hex Representation
Zero	0	**000**
–	-9	**3f7**
i	**-42**	**3d6**
i >> 5	**-2**	**3fe**
ui	**982**	**3d6**
(int) s	**-7**	**3f9**
(int)(s ^ -12)	**13**	**00d**
(int) us	**25**	**019**
TMax	**511**	**1ff**
TMin	**-512**	**200**

Problem 2. (14 points):

Consider the following 11-bit floating point representation based on the IEEE floating point format:

- There is a sign bit in the most significant bit.
- The next $k = 4$ bits are the exponent. The exponent bias is 7.
- The last $n = 6$ bits are the significand.

Numeric values are encoded in this format as a value of the form $V = (-1)^s \times M \times 2^E$, where s is the sign bit, E is exponent after biasing, and M is the significand.

Part I

How many FP numbers are in the following intervals $[a, b)$?

For each interval $[a, b)$, count the number of x such that $a \leq x < b$.

A. Interval $[1, 2)$: 2^6**=64**

B. Interval $[2, 2.5)$: 2^4**=16**

Part II

Answer the following problems using either decimal (e.g., 1.375) or fractional (e.g., $11/8$) representations for numbers that are not integers.

A. For denormalized numbers:

   (a) What is the smallest value E of the exponent after biasing? **-6**

   (b) What is the largest value M of the significand? **63/64**

B. For normalized numbers:

   (a) What is the smallest value E of the exponent after biasing? **-6**

   (b) What is the largest value E of the exponent after biasing? **7**

   (c) What is the smallest value M of the significand? **1**

   (d) What is the largest value M of the significand? **127/64**

Part III

Fill in the blank entries in the following table giving the encodings for some interesting numbers.

Description	E	M	V	Binary Encoding
Zero	-6	0	0	`0 0000 000000`
Smallest Positive (nonzero)	**-6**	**1/64**	**1/4096**	**0 0000 000001**
Largest denormalized	**-6**	**63/64**	**63/4096**	**0 0000 111111**
Smallest positive normalized	**-6**	**1**	**1/64**	**0 0001 000000**
Positive Infinity	—	—	—	**0 1111 000000**
Negative Infinity	—	—	—	**1 1111 000000**

Problem 3. (12 points):

Consider the following assembly code:

```
08048333 <func>:
 8048333:       55                      push   %ebp
 8048334:       89 e5                   mov    %esp,%ebp
 8048336:       83 ec 0c                sub    $0xc,%esp
 8048339:       c7 45 fc 00 00 00 00    movl   $0x0,0xfffffffc(%ebp)
 8048340:       8b 45 08                mov    0x8(%ebp),%eax
 8048343:       0f af 45 0c             imul   0xc(%ebp),%eax
 8048347:       39 45 fc                cmp    %eax,0xfffffffc(%ebp)
 804834a:       72 02                   jb     804834e <func+0x1b>
 804834c:       eb 2f                   jmp    804837d <func+0x4a>
 804834e:       8b 45 08                mov    0x8(%ebp),%eax
 8048351:       89 c2                   mov    %eax,%edx
 8048353:       0f af 55 0c             imul   0xc(%ebp),%edx
 8048357:       8d 45 fc                lea    0xfffffffc(%ebp),%eax
 804835a:       01 10                   add    %edx,(%eax)
 804835c:       8b 45 08                mov    0x8(%ebp),%eax
 804835f:       48                      dec    %eax
 8048360:       89 44 24 04             mov    %eax,0x4(%esp,1)
 8048364:       8b 45 0c                mov    0xc(%ebp),%eax
 8048367:       03 45 08                add    0x8(%ebp),%eax
 804836a:       d1 e8                   shr    $0x1,%eax
 804836c:       89 04 24                mov    %eax,(%esp,1)
 804836f:       e8 bf ff ff ff          call   8048333 <func>
 8048374:       89 c2                   mov    %eax,%edx
 8048376:       8d 45 fc                lea    0xfffffffc(%ebp),%eax
 8048379:       29 10                   sub    %edx,(%eax)
 804837b:       eb c3                   jmp    8048340 <func+0xd>
 804837d:       8b 45 fc                mov    0xfffffffc(%ebp),%eax
 8048380:       c9                      leave
 8048381:       c3                      ret
```

The assembly on the previous page corresponds to the C code below. Fill in the blanks in the C code to match the operations done in the assembly.

```
unsigned int func(unsigned int a, unsigned int b)
{
  unsigned int result = 0;
  while (result ___< a*b___) {
    result += ___a*b___;
    result -= func(___(a+b)/2___, ___a-1___);
  }
  return result;
}
```

Problem 4. (8 points):

Consider the following C declarations:

```
struct a_struct {
        char             a;
        struct a_strcut *b;
};

struct b_struct {
        char            c;
        int             i;
        double *        d;
        short           e[3];
        struct a_struct m;
};
```

A. Using the templates below (allowing a maximum of 32 bytes), indicate the allocation of data for `struct b_struct`. 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.**

```
struct b_struct:

  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| c  X  X  X| i  i  i  i| d  d  d  d| e  e  e  e|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| e  e  X  X| a  X  X  X| b  b  b  b|           |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

B. How would you define the `struct b2_struct` structure to minimize the number of bytes allocated for the structure using the same fields as the `struct b_struct` structure?

```
struct b2_struct {
        char            c;
        short           e[3];
        int             i;
        double *        d;
        struct a_struct m;
```

(Most solutions with e and c adjacent work.)

```
};
```

C. What is the value of `sizeof(struct b2_struct)`? ______**24**______

Problem 5. (12 points):

Consider the following C code:

```
struct triple
{
    int x;
    char c;
    int y;
};

int mystery1(int x);
int mystery2(int x);
int mystery3(struct triple* t);

int main()
{
    struct triple t = {35, ’q’, 10};

    int result1 = mystery1(42);
    int result2 = mystery2(19);
    int result3 = mystery3(&t);

    printf("result1 = %d\n", result1);
    printf("result2 = %d\n", result2);
    printf("result3 = %d\n", result3);

    return 0;
}
```

Using the assembly code for `mystery1`, `mystery2`, and `mystery3` on the next page, fill in the proper values in this program’s output:

result1 = **3**

result2 = **19**

result3 = **350**

```
080483d0 <mystery1>:
 80483d0:       55                      push   %ebp
 80483d1:       89 e5                   mov    %esp,%ebp
 80483d3:       53                      push   %ebx
 80483d4:       8b 45 08                mov    0x8(%ebp),%eax
 80483d7:       89 c3                   mov    %eax,%ebx
 80483d9:       83 e3 01                and    $0x1,%ebx
 80483dc:       85 c0                   test   %eax,%eax
 80483de:       74 0b                   je     80483eb <mystery1+0x1b>
 80483e0:       c1 f8 01                sar    $0x1,%eax
 80483e3:       50                      push   %eax
 80483e4:       e8 e7 ff ff ff          call   80483d0 <mystery1>
 80483e9:       01 c3                   add    %eax,%ebx
 80483eb:       89 d8                   mov    %ebx,%eax
 80483ed:       8b 5d fc                mov    0xfffffffc(%ebp),%ebx
 80483f0:       c9                      leave
 80483f1:       c3                      ret

080483f4 <mystery2>:
 80483f4:       55                      push   %ebp
 80483f5:       89 e5                   mov    %esp,%ebp
 80483f7:       8b 55 08                mov    0x8(%ebp),%edx
 80483fa:       31 c0                   xor    %eax,%eax
 80483fc:       85 d2                   test   %edx,%edx
 80483fe:       7e 06                   jle    8048406 <mystery2+0x12>
 8048400:       40                      inc    %eax
 8048401:       4a                      dec    %edx
 8048402:       85 d2                   test   %edx,%edx
 8048404:       7f fa                   jg     8048400 <mystery2+0xc>
 8048406:       c9                      leave
 8048407:       c3                      ret

08048408 <mystery3>:
 8048408:       55                      push   %ebp
 8048409:       89 e5                   mov    %esp,%ebp
 804840b:       8b 45 08                mov    0x8(%ebp),%eax
 804840e:       8b 10                   mov    (%eax),%edx
 8048410:       0f af 50 08             imul   0x8(%eax),%edx
 8048414:       89 d0                   mov    %edx,%eax
 8048416:       c9                      leave
 8048417:       c3                      ret
```

Problem 6. (12 points):

This problem tests your understanding of byte ordering and the stack discipline. The following program reads a string from standard input and prints an integer in it's hexadecimal format based on the input it was given.

```
#include <stdio.h>

int get_key () {
    int key;
    scanf ("%s", &key);
    return key;
}

int main () {
    printf ("0x%8x\n", get_key());
    return 0;
}
```

Here is the corresponding machine code on a Linux/x86 machine:

```
08048414 <get_key>:
 8048414:   55                push   %ebp
 8048415:   89 e5             mov    %esp,%ebp
 8048417:   83 ec 18          sub    $0x18,%esp
 804841a:   83 c4 f8          add    $0xfffffff8,%esp
 804841d:   8d 45 fc          lea    0xfffffffc(%ebp),%eax
 8048420:   50                push   %eax       address arg for scanf
 8048421:   68 b8 84 04 08    push   $0x80484b8  format string for scanf
 8048426:   e8 e1 fe ff ff    call   804830c <_init+0x50>  call scanf
 804842b:   8b 45 fc          mov    0xfffffffc(%ebp),%eax
 804842e:   89 ec             mov    %ebp,%esp
 8048430:   5d                pop    %ebp
 8048431:   c3                ret

08048434 <main>:
 8048434:   55                push   %ebp
 8048435:   89 e5             mov    %esp,%ebp
 8048437:   83 ec 08          sub    $0x8,%esp
 804843a:   83 c4 f8          add    $0xfffffff8,%esp
 804843d:   e8 d2 ff ff ff    call   8048414 <get_key>
 8048442:   50                push   %eax       integer arg of printf
 8048443:   68 bb 84 04 08    push   $0x80484bb  format string for printf
 8048448:   e8 ef fe ff ff    call   804833c <_init+0x80>  call printf
 804844d:   31 c0             xor    %eax,%eax
 804844f:   89 ec             mov    %ebp,%esp
 8048451:   5d                pop    %ebp
 8048452:   c3                ret
```

Here are a few notes to help you with the problem:

- `scanf (''%s'', i)` reads a string from the standard input stream and stores it at stores it at address `i` (including the terminating `'\0'` character. It does **not** check the size of the destination buffer.
- `printf ("0x%8x\n", j)` prints 8 digits of the integer `i` in hexadecimal format as `0x`$xxxxxx$
- Recall that Linux/x86 machines are Little Endian.
- You will need to know the hex values for the following characters:

Character	Hex Value	Character	Hex Value
'E'	0x45	's'	0x73
'v'	0x76	'h'	0x68
'i'	0x69	'!'	0x21
'l'	0x6c	'\0'	0x00

A. Suppose we run this program on a Linux/x86 machine with the input string "`Evil!`".

Here is a template for the stack, showing the location of `key`. Indicate with a labelled arrow 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**).

```
                    |<------ key ------>|
+----+----+----+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    | 45 | 76 | 69 | 6c | 21 | 00 |    |    |    |
+----+----+----+----+----+----+----+----+----+----+----+----+----+
                                        ^
                                      %ebp
```

What is the 4-byte integer (in hex) printed by `printf` inside main?

0x **6c697645**

B. Suppose we instead gave it the input string “`Evillish!`”.

For the remaining problems, each answer should be an unsigned 4-byte integer expressed as 8 hexadecimal digits.

(a) What is the value of `(&key)[1]` just **after** `scanf` returns to `get_key`?

`(&key)[1] = 0x`<u>**6873696c**</u>

(b) What is the value of `%ebp` immediately **before** the execution of the `ret` instruction of `get_key`?

`%ebp = 0x`<u>**6873696c**</u>

You can use the following template of the stack as *scratch space*. This **will not** be considered for credit.

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

Problem 7. (6 points):

Consider the following code for a matrix multiplication function:

```
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<m; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

with matricies `a[n][m]`, `b[m][n]` and `c[n,n]`.

1. Assume that `m` is twice as large as `n`. Is the above loop optimally arranged for preserving locality? If not, state the optimal nesting of these loops.

No,

Nest them as

```
for ( ... i ...)
  for ( ... k ... )
    for ( ... j ... )
```

see class13, slides 30-40

2. Assume that `n` is twice as large as `m`. Is the above loop optimally arranged for preserving locality? If not, state the optimal nesting of these loops.

No,

Nest them as

```
for ( ... i ...)
  for ( ... k ... )
    for ( ... j ... )
```

see class13, slides 30-40

Problem 8. (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.

1. If it takes less time to do either a cache access or a main memory access than to perform a branch, then loop unrolling always improves performance. **false**

2. On the Fish machines the most effective way to time a short routine is to use the cycle counter. **true**

3. Loop invariant code motion can be applied to expressions involving the loop induction variable **false**

4a.
```
typedef int (*a)(int *);
typedef a b[10];
typedef b* (*c)();
c d;
int *(*(*)(int *))[10] (*e)();
```
`d` and `e` have the same type. **either**

4b. `e` and `c` have the same type. **false**

5. In C, the variable `f` is declared as: `int f[12][17]`. The address for f[3][8] can sometimes be greater than the address for f[8][3]. **false**

6. The largest possible finite denormalized IEEE floating number is greater than the smallest possible positive normalized IEEE floating number: **false**