**Andrew login ID:** _____

**Full Name:** _____

**Recitation Section:** _____

# CS 15-213, Fall 2009
# Exam 1

Thursday, September 24, 2009

**Instructions:**

- Make sure that your exam is not missing any sheets, then write your full name, Andrew login ID, and recitation section (A–J) on the front.
- Write your answers in the space provided for the problem. If you make a mess, clearly indicate your final answer.
- The exam has a maximum score of 76 points.
- The problems are of varying difficulty. The point value of each problem is indicated (instructors reserve the right to change these values). 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 calculators or other electronic devices are allowed.
- QUESTIONS: If you have a question, write it (clearly) on an index card and raise your hand. We will take the card and write a reply.
- Good luck!

| | |
|---|---|
| 1 (10): | |
| 2 (8): | |
| 3 (12): | |
| 4 (11): | |
| 5 (8): | |
| 6 (12): | |
| 7 (9): | |
| 8 (6): | |
| Extra (4): | |
| TOTAL (76): | |

## Problem 1. (10 points):

### Part A

Fill in the blanks in the table below with the number described in the first column of each row. You can give your answers as unexpanded simple arithmetic expressions (such as $15^{213} + 42$); you should not have trouble fitting your answers into the space provided.

Remember that 32-bit floats have 8 bits of exponent and 23 bits of mantissa.

| Description | Number |
|---|---|
| `int x=0; float *f = (float *)&x;` What is the value of `*f`? | |
| `int x=-1; float *f = (float *)&x;` What is the value of `*f`? | |
| Smallest positive, non-zero denormalized 32-bit `float` | |

### Part B

Assume we are running code on an IA32 machine, which has a 32-bit word size and uses two's complement arithmetic for signed integers. Consider the following definition:

```
int x = foo();
unsigned int ux = x;
int y = bar();
```

Fill in the empty boxes in the table below. For each of the C expressions in the first column, either:

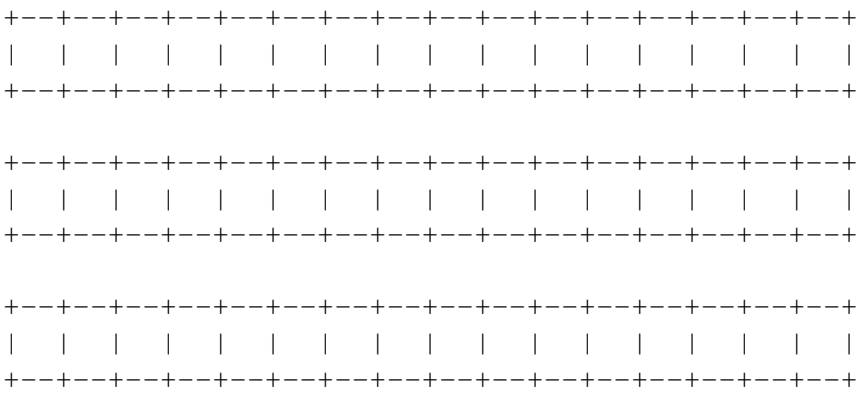
- State that it is true of all argument values, or
- Give an example where it is not true.

| Puzzle | True / Counterexample |
|---|---|
| `(x >> 31) ^ ((-x) >> 31) == 0` | |
| `x ^ ~(x >> 31) < 0` | |
| `x ^ y ^ (~x) - y == y ^ x ^ (~y) - x` | |
| `(((!!ux)) << 31) >> 31) == (((!!x) << 31) >> 31)` | |

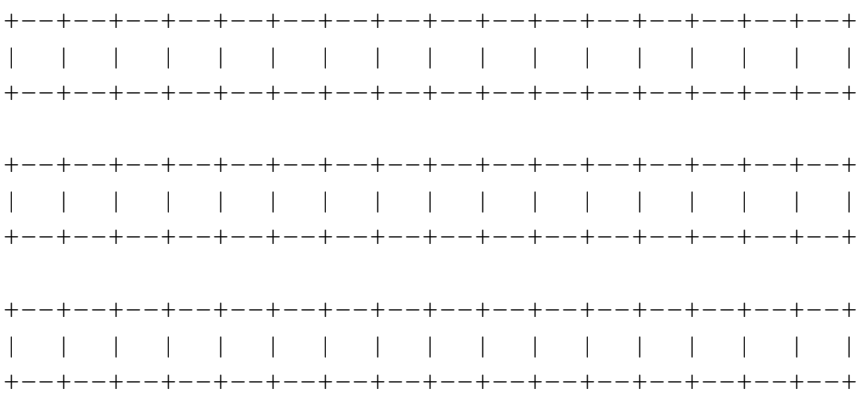
## Problem 2. (8 points):

```
struct {
    char a[9];
    short b[3];
    float c;
    char d;
    int e;
    char *f;
    short g;
} foo;
```

A. Show how the struct above would appear on a 64-bit ("x86_64") Windows machine (primitives of size $k$ are $k$-byte aligned). Label the bytes that belong to the various fields with their names and clearly mark the end of the struct. Use hatch marks or x's to indicate bytes that are allocated in the struct but are not used.



B. Rearrange the above fields in `foo` to conserve the most space in the memory below. Label the bytes that belong to the various fields with their names and clearly mark the end of the struct. Use hatch marks or x's to indicate bytes that are allocated in the struct but are not used.



C. How many bytes are wasted in part A, inside and after the struct, if the next memory value is a pointer?

D. How many bytes are wasted in part B, inside and after the struct, if the next memory value is a pointer?

## Problem 3. (12 points):

Consider the following two 8-bit floating point representations based on the IEEE floating point format. Neither has a sign bit—they can only represent nonnegative numbers.

1. Format A
   - There are $k = 4$ exponent bits. The exponent bias is 7.
   - There are $n = 4$ fraction bits.
2. Format B
   - There are $k = 6$ exponent bits. The exponent bias is 31.
   - There are $n = 2$ fraction bits.

Fill in the blanks in the table below by converting the given values in each format to the closest possible value in the other format. Express values as whole numbers (e.g., 17) or as fractions (e.g., 17/64). If necessary, you should apply the round-to-even rounding rule. If conversion would cause an overflow, follow the IEEE standard convention for representing +Infinity. You should also assume IEEE conventions for representing denormalized values.

| Format A | | Format B | |
|---|---|---|---|
| Bits | Value | Bits | Value |
| `0111 0000` | 1 | `011111 00` | 1 |
| | | | 112 |
| | $\frac{23}{32}$ | | |
| | | `110111 10` | |
| `0000 0101` | | | |

## Problem 4. (11 points):

Your friend Harry Bovik, who hasn't taken 15-213, is in need of your help. He was writing a function to do strange arithmetic for a project of his, but accidentally deleted his source code file, and also spilled his drink across the sheet of paper with his scratch work on it, leaving him with only half-legible code and an executable file that he compiled just recently. Being the clever student that you are, you ask to see his scratchwork and executable file.

```
int foo(_____________ a)
{
    _____________ b = 0;

    switch (_____________) {
        case 0:

            b = _____________;

            _____________;
        case 1:

            b = _____________;

            _____________;
        case 2:

            b = _____________;

            _____________;
        case 3:

            b = _____________;

            _____________;
        case 4:

            b = _____________;

            _____________;
    }
    return b;
}
```

Feeding the executable to your trusty debugger, you find the following relevant information:

```
(gdb) disassemble foo
Dump of assembler code for function foo:
0x0000000000400508 <foo+0>:     mov    $0x0,%edx
0x000000000040050d <foo+5>:     lea    0x1(%rdi),%eax     # %rdi: first argument
0x0000000000400510 <foo+8>:     cmp    $0x4,%eax
0x0000000000400513 <foo+11>:    ja     0x40052c <foo+36>
0x0000000000400515 <foo+13>:    mov    %eax,%eax
0x0000000000400517 <foo+15>:    jmpq   *0x4006d0(,%rax,8)
0x000000000040051e <foo+22>:    mov    %edi,%edx
0x0000000000400520 <foo+24>:    shr    %edx
0x0000000000400522 <foo+26>:    not    %edx
0x0000000000400524 <foo+28>:    neg    %edx
0x0000000000400526 <foo+30>:    jmp    0x40052c <foo+36>
0x0000000000400528 <foo+32>:    mov    %edi,%edx
0x000000000040052a <foo+34>:    xor    %edi,%edx
0x000000000040052c <foo+36>:    mov    %edx,%eax
0x000000000040052e <foo+38>:    retq
End of assembler dump.
(gdb) x/8g 0x4006d0
0x4006d0:       0x000000000040051e      0x0000000000400522
0x4006e0:       0x0000000000400524      0x0000000000400528
0x4006f0:       0x000000000040052a      0x3b031b01000a6425
0x400700:       0x0000000400000028      0x00000044fffffe0c
```

1. Unfortunately Harry's scratch work has `break` statements hastily scribbled in and crossed out again in every case, and he can't remember which cases are supposed to have them. Using the assembly dump of his function, figure out which cases had `break`s at the end of them. (Write either "`break`" or nothing at all in the last blank of each case block.)

2. The scratch work you were handed also failed to note what types `a` and `b` are, but fortunately some of the opcodes give it away. Figure out what types Harry meant for his variables to be.

3. Using the disassembly of `foo` and the jump table you found, reconstruct the rest of the switch statement.

4. What values will `foo` return for each possible input `a`?

## Problem 5. (8 points):

Consider the following data structure declarations:

```
struct node {
  unsigned uid;
  union data d;
  struct node *next;
};
```

```
union data {
  int x[3];
  long y[3];
};
```

Below are given four C functions and five x86_64 code blocks, compiled on Linux using GCC.

```
int odin(struct node *ptr) {
  return (ptr->d.x[2]);
}
```

| | |
|---|---|
| A | `mov 0x20(%rdi),%rax`<br>`mov 0x8(%rax),%rax` |

```
unsigned dva(struct node *ptr) {
  return (ptr->uid = (long)ptr->next);
}
```

| | |
|---|---|
| B | `mov 0x10(%rdi),%eax` |

| | |
|---|---|
| C | `mov 0xc(%rdi),%rax` |

```
long tri(struct node *ptr) {
  union data *dptr =
    (union data *)ptr->next;
   return dptr->y[1];
}
```

| | |
|---|---|
| D | `mov 0x20(%rdi),%rax`<br>`add $0x8,%rax` |

```
long *chetyre(struct node *ptr) {
  return &ptr->next->d.y[0];
}
```

| | |
|---|---|
| E | `mov 0x20(%rdi),%rax`<br>`mov %eax,(%rdi)` |

In the following table, next to the name of each C function, write the name of the x86_64 block that implements it.

| Function Name | Code Block |
|---|---|
| odin | |
| dva | |
| tri | |
| chetyre | |

## Problem 6. (12 points):

Below is some assembly code to a famous algorithm. Please briefly read the code then answer the questions on the following page.

```
0000000000400498 <mystery>:
  400498:  41 b8 00 00 00 00  mov    $0x0,%r8d
  40049e:  eb 22              jmp    4004c2 <mystery+0x2a>
  4004a0:  89 c8              mov    %ecx,%eax
  4004a2:  c1 e8 1f           shr    $0x1f,%eax
  4004a5:  01 c8              add    %ecx,%eax
  4004a7:  d1 f8              sar    %eax          ; arith shift right 1 bit
  4004a9:  42 8d 0c 00        lea    (%rax,%r8,1),%ecx
  4004ad:  48 63 c1           movslq %ecx,%rax
  4004b0:  8b 04 87           mov    (%rdi,%rax,4),%eax
  4004b3:  39 d0              cmp    %edx,%eax
  4004b5:  7d 05              jge    4004bc <mystery+0x24>
  4004b7:  41 89 c8           mov    %ecx,%r8d
  4004ba:  eb 06              jmp    4004c2 <mystery+0x2a>
  4004bc:  39 d0              cmp    %edx,%eax
  4004be:  7e 10              jle    4004d0 <mystery+0x38>
  4004c0:  89 ce              mov    %ecx,%esi
  4004c2:  89 f1              mov    %esi,%ecx
  4004c4:  44 29 c1           sub    %r8d,%ecx
  4004c7:  85 c9              test   %ecx,%ecx
  4004c9:  7f d5              jg     4004a0 <mystery+0x8>
  4004cb:  b9 ff ff ff ff     mov    $0xffffffff,%ecx
  4004d0:  89 c8              mov    %ecx,%eax
  4004d2:  c3                 retq
```

**a)** Please write a single line of C code to represent the instruction `lea (%rax,%r8,1),%ecx` (Use C variables named rax,r8, and ecx, you can ignore types).

**b)** Please write a single line of C code to represent the instruction `mov (%rdi,%rax,4),%eax` (Use C variables named rdi and rax, you can ignore types).

**c)** Commonly found in assembly is the `leave` instruction; why is that instruction not in this code?

**d)** You learned about two different architectures in class, IA32 and x86_64. What architecture is this code written for and what major downside would occur from using the other architecture?

**e)** Now for the fun part! Please fill in the blanks in the following C code to match the assembly above.

```
int mystery (_______ * array, size_t size, int e){
    int a;

    int __________ = size;

    int __________ = 0;

    while (____________________> 0){

        a = ________________________;

        if(_____________________________________________){

            ________ = a;

        }else if (_____________________________________){

            ________ = a;
        }else{

            return _________
        }
    }

    return __________;
}
```

**f)** What famous algorithm is this?

## Problem 7. (9 points):

Circle the correct answer. Assume IA32 unless stated otherwise.

1. Here is a small C program:

```
struct foo { int bar; int baz; };

int get_baz(struct foo *foo_ptr)
{
    return foo_ptr->baz;
}
```

After compiling the code, disassembling `get_baz`, and adding a few comments, we get:

```
get_baz: push   %ebp             ; save old frame base pointer
         mov    %esp,%ebp        ; set frame base pointer
         mov    0x8(%ebp),%eax   ; move foo_ptr to %eax
         –Mystery Instruction Goes Here–
         leave
         ret
```

What is the *Mystery Instruction*?

(a) `mov $baz(%eax),%eax`

(b) `mov 0x4(%eax),%eax`

(c) `lea 0x4(%eax),%eax`

(d) `mov 0xc(%ebp),%eax`

2. The function `bitsy` is declared in C as

```
int bitsy(int x);
```

and the (correctly) compiled IA32 code is:

```
bitsy: push   %ebp
       mov    %esp,%ebp
       sub    $0x8,%esp
       mov    0x8(%ebp),%eax
       not    %eax
       inc    %eax
       leave
       ret
```

What is the result (denoted here by a C expression) returned by bitsy?

(a) `!(x + 1)`

(b) `*(1 - x)`

(c) `-x`

(d) `(x > 0 ? -x : -x + 1)`

3. Which of the following is true:

   (a) There are no IEEE float representations exactly equal to zero.

   (b) There is one IEEE float representation exactly equal to zero.

   (c) There are two IEEE float representations exactly equal to zero.

   (d) There are many IEEE float representations exactly equal to zero.

4. Which one of the following is true:

   (a) Denormalized floats must be normalized before a floating point computation is complete.

   (b) Denormalized floats represent magnitudes smaller than those of normalized floats.

   (c) Denormalized floats signal a computation error or an undefined result.

   (d) Denormalized floats represent magnitudes greater than those of normalized floats.

5. Why does the compiler sometimes generate `xorl %eax,%eax` rather than `movl $0x0,%eax`?

   (a) Using `xorl` allows the binary code to run on both IA32 and x86-64 architectures.

   (b) The `xorl` form is faster and/or uses fewer bytes than `movl`.

   (c) The `movl` form requires a zero to be accessed from memory location 0.

   (d) The `xorl` form stalls the processor until the the result value is stored in %eax and ready for use by the next instruction.

6. On x86-64, `addl %ebx,%eax` has the following effect:

   (a) %eax gets %eax + %ebx, high-order 32 bits of %rax are zeroed

   (b) %eax gets %eax + %ebx, %rax is unchanged

   (c) %eax gets %eax + %ebx, high-order 32 bits of %rax are sign-extended

   (d) %rax gets %eax + %ebx

7. If %esp has the value 0xBFFF0000 before a call instruction, the value immediately after the call instruction (before the first instruction of the called function) is:

   (a) 0xBFFEFFFC

   (b) 0xBFFF0004

   (c) 0xBFFF0000

   (d) The address of the instruction after the call instruction.

8. Which of the following is true:

   (a) A function can immediately clear *any* "callee save" registers.

   (b) The caller must always save *all* "caller save" registers before calling a function.

   (c) The called function must immediately save *all* "callee save" registers on the stack and restore them before returning.

   (d) A function can always ignore the initial values of *all* "caller save" registers.

9. Which of the following is true:

   (a) A 32-bit IEEE float can represent any 32-bit integer to within 0.5.

   (b) All 32-bit IEEE floats with integer values are encoded with the binary point at the rightmost bit, so E (the exponent) is 0 and exp (the 8-bit exponent field) is E + bias = 127.

   (c) No decimal integer has an exact representation in IEEE floating point (10 is not a power of 2).

   (d) There is no exact representation in IEEE floating point of most decimal fractions.

## Problem 8. (6 points):

Consider the C code below, where `H` and `J` are constants declared with `#define`.

```
int array1[H][J];
int array2[J][H];

int copy_array(int x, int y) {
    array2[y][x] = array1[x][y];

    return 1;
}
```

Suppose the above C code generates the following x86-64 assembly code:

```
# On entry:
#    %edi = x
#    %esi = y
#
copy_array:
  movslq  %edi,%rdi
  movslq  %esi,%rsi
  movq    %rdi, %rax
  leaq    0(,%rsi,8), %rdx
  salq    $4, %rax     ; arith shift left
  subq    %rdi, %rax
  subq    %rsi, %rdx
  addq    %rsi, %rax
  leaq    (%rdi,%rdx,4), %rdx
  movl    array1(,%rax,4), %eax
  movl    %eax, array2(,%rdx,4)
  movl    $1, %eax
  ret
```

What are the values of `H` and `J`?

H =

J =

## Extra Credit (4 points)

**This problem is Extra Credit; do not attempt it until you have finished all other questions on the exam. This question is based on knowledge this class does not cover, and you are not expected to know how to solve it.**

This problem deals with a tricky problem with GCC when run with high levels of optimization. This code in particular is compiled with

```
$ gcc -O3 input.c
```

One of your friends who hasn't taken 213 comes to you with a program, wanting your help. They tell you that they have been debugging it for hours, finally removing all their intricate code and just putting a single printf statement inside their loop. They show you this relevant code:

```
short a = 1024;
short b;

for(b=1000;;b++){
    if(a+b < 0){
        printf("Overflow!, stopping\n");
        break;
    }
    printf("%d ",a+b);
}
```

Their code never breaks and runs in an infinite loop. You, being a 213 student of course immediately ask to see the assembly dump:

```
08048380 <main>:
 8048380: 8d 4c 24 04           lea    0x4(%esp),%ecx
 8048384: 83 e4 f0              and    $0xfffffff0,%esp
 8048387: ff 71 fc              pushl  0xfffffffc(%ecx)
 804838a: 55                    push   %ebp
 804838b: 89 e5                 mov    %esp,%ebp
 804838d: 53                    push   %ebx
 804838e: bb e8 07 00 00        mov    $0x7e8,%ebx
 8048393: 51                    push   %ecx
 8048394: 83 ec 10              sub    $0x10,%esp
 8048397: 89 5c 24 04           mov    %ebx,0x4(%esp)
 804839b: 83 c3 01              add    $0x1,%ebx
 804839e: c7 04 24 70 84 04 08  movl   $0x8048470,(%esp)
 80483a5: e8 2e ff ff ff        call   80482d8 <printf@plt>
 80483aa: eb eb                 jmp    8048397 <main+0x17>
 80483ac: 90                    nop
 80483ad: 90                    nop
 80483ae: 90                    nop
 80483af: 90                    nop
```

1. From the programmer's point of view, what is wrong with this assembly code?

2. Why do you think gcc did this? (hint: we never mentioned this in class)

3. Please write the assembly code necessary to achieve the behavior intended by the programmer, and tell us where you would insert the code.