Andrew login ID: _____

Full Name: _____

Recitation Section: _____

CS 15-213, Spring 2009

Exam 1

Tues., February 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 100 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 calculators or other electronic devices are allowed.
- Good luck!

1 (16):
2 (22):
3 (13):
4 (13):
5 (22):
6 (14):
TOTAL (100):

Problem 1. (16 points):

Consider a new floating point format that follows the IEEE spec you should be familiar, except with 3 exponent bits and 2 fraction bits (and 1 sign bit). Fill in all blank cells in the table below. *I*f, in the process of converting a decimal number to a float, you have to round, write the rounded value next to the original decimal as well.

Description	Decimal	Binary Representation
Bias		`-----`
Smallest positive number		
Lowest finite		
Smallest positive normalized		
`-----`	$-\frac{6}{16}$	
`-----`	$\frac{6}{4}$	
`-----`		`1 010 10`
`-----`	11	

Problem 2. (22 points):

Consider the C code written below and compiled on a 32-bit Linux system using GCC.

```
struct s1
{
    int y;
    short x;
};

struct s2
{
    struct s1 a;
    struct s1 *b;
    int x;
    char c;
    int y;
    char e[5];
    int z;
};

short fun1(struct s2 *s)
{
    return s->a.x;
}

void *fun2(struct s2 *s)
{
    return &s->z;
}

int fun3(struct s2 *s)
{
    return s->z;
}

short fun4(struct s2 *s)
{
    return s->b->x;
}
```

a) What is the size of `struct s2`?

b) How many bytes are wasted for padding?

You may use the rest of the space on this page for scratch space to help with the rest of this problem. Nothing written below this line will be graded.

c) Which of the following correspond to functions `fun1`, `fun2`, `fun3`, and `fun4`?

```
push   %ebp
mov    %esp,%ebp
mov    0x8(%ebp),%eax
movswl 0x4(%eax),%eax
pop    %ebp
ret
```

ANSWER: _____

```
push   %ebp
mov    %esp,%ebp
mov    0x8(%ebp),%eax
mov    0x8(%eax),%eax
movswl 0x4(%eax),%eax
pop    %ebp
ret
```

ANSWER: _____

```
push   %ebp
mov    %esp,%ebp
mov    0x8(%ebp),%eax
mov    0x20(%eax),%eax
pop    %ebp
ret
```

ANSWER: _____

```
push   %ebp
mov    %esp,%ebp
mov    0x8(%ebp),%eax
add    $0x20,%eax
pop    %ebp
ret
```

ANSWER: _____

d) Assume a variable is declared as `struct s2 myS2;` and the storage for this variable begins at address `0xbfbdc300`.

```
(gdb) x/20w &myS2
0xbfbdc300:     0x000000d5      0x0000000f      0xbfbdc338      0x00000000
0xbfbdc310:     0x000000ff      0x0000012c      0x01020501      0xb7f0a603
0xbfbdc320:     0x0000000c      0x080496a0      0xbfbdc338      0x0804828d
0xbfbdc330:     0xb7ed9fd5      0xb7fc1ff4      0x000000f3      0x08040012
0xbfbdc340:     0xb7eda0b9      0xb7fc1ff4      0xbfbdc3a8      0xb7ec6dce
```

Fill in all the blanks below.

HINTS: Label the fields. Not all 20 words are used. Remember endianness!

What would be returned by:

```
fun1(&myS2) = 0x___________

fun2(&myS2) = 0x___________

fun3(&myS2) = 0x___________

fun4(&myS2) = 0x___________
```

What is the value of:

```
myS2.b->y = 0x___________

myS2.a.y  = 0x___________

myS2.z    = 0x___________

myS2.e[1] = 0x___________
```

Problem 3. (13 points):

This problem concerns assembly code generated by GCC for a function containing a switch statement on an x86-64 machine.

Below is the entire assembly dump of the function selector, whose C skeleton is given subsequently.

```
0000000000400470 <selector>:
400470:   8d 46 01                lea    0x1(%rsi),%eax
400473:   83 f8 06                cmp    $0x6,%eax
400476:   77 1a                   ja     400492 <selector+0x22>
400478:   89 c0                   mov    %eax,%eax
40047a:   ff 24 c5 a0 05 40 00    jmpq   *0x4005a0(,%rax,8)
400481:   48 83 c7 04             add    $0x4,%rdi
400485:   eb 0e                   jmp    400495 <selector+0x25>
400487:   8b 37                   mov    (%rdi),%esi
400489:   eb 0a                   jmp    400495 <selector+0x25>
40048b:   d1 fe                   sar    %esi
40048d:   83 c6 05                add    $0x5,%esi
400490:   eb 03                   jmp    400495 <selector+0x25>
400492:   8d 34 f6                lea    (%rsi,%rsi,8),%esi
400495:   48 63 c6                movslq %esi,%rax
400498:   8b 04 87                mov    (%rdi,%rax,4),%eax
40049b:   c3                      retq
```

The code at line 40047a uses an indirect jump to index into a jump table:

```
40047a:   ff 24 c5 a0 05 40 00    jmpq   *0x4005a0(,%rax,8)
```

In GDB, we examine the memory dump at address `0x4005a0` which produces the following output:

```
(gdb) x /8g 0x4005a0
0x4005a0:    0x0000000000400487      0x0000000000400481
0x4005b0:    0x0000000000400492      0x000000000040048d
0x4005c0:    0x000000000040048b      0x0000000000400492
0x4005d0:    0x0000000000400481      0x0000002c3b031b01
```

Fill in the blank portions of C code below to reproduce the function corresponding to this object code.

```
int selector (int *x, int y) {
  switch (y) {
  case ____:
  case ____:
    ____________;
    break;
  case ____:
    y = ________;
    break;
  case ____:
    y = ________;
  case ____:
    y = ________;
    break;
  default:
    y = ________;
  }
  return ________;
}
```

Problem 4. (13 points):

The function below is hand-written assembly code for a sorting algorithm. Fill in the blanks on the next page by converting this assembly to C code.

```
.globl mystery_sort     # exports the symbol so other .c files
                        # can call the function

mystery_sort:
        dec     %rsi

        xor     %rdx, %rdx
        inc     %rdx
        jmp     loop1_check

loop1:
        xor     %rdx, %rdx
        mov     %rsi, %rcx
        jmp     loop2_check

loop2:
        dec     %rcx
        mov     8(%rdi, %rcx, 8), %r8
        mov     (%rdi, %rcx, 8), %r9
        cmp     %r8, %r9
        jle     loop2_check
        mov     %r8, (%rdi, %rcx, 8)
        mov     %r9, 8(%rdi, %rcx, 8)
        inc     %rdx

loop2_check:
        test    %rcx, %rcx
        jnz     loop2

loop1_check:
        test    %rdx, %rdx
        jnz     loop1

        ret
```

```
void mystery_sort (long* array, long len)
{
  long a, b, tmp;

  do
  {
    a = _____;

    for (b = _____; b > _____; b--)
    {

      if (array[_____] > array{_____])
      {

        tmp = array[_____];

        array[_____] = array[_____];

        array[_____] = tmp;

        a++;

      }

    }

  } while (a > _____);

}
```

Problem 5. (22 points):

Circle the correct answer.

1. What register is the return value from a function stored in? (Assume 32 bit return value)
   (a) eip
   (b) ebp
   (c) eax
   (d) esp
2. The pushl instruction does what to the stack pointer?
   (a) decrements the stack pointer by 4 bytes
   (b) decrements the stack pointer by 1 byte
   (c) increments the stack pointer by 4 bytes
   (d) increments the stack pointer by 1 byte
3. What does the test instruction do before setting condition flags?
   (a) bitwise and
   (b) subtraction
   (c) bitwise xor
   (d) bitwise bang
4. On the x86_64 fish machines, what is the size of an int?
   (a) 1 byte
   (b) 32 bytes
   (c) 4 bytes
   (d) 8 bytes
5. Which of the following represents the order of the registers that store the first four arguments to a function in x86_64?
   (a) rdi, rsi, rcx, rdx
   (b) rax, rbx, rsi, rdi
   (c) rsi, rdi, rdx, rbx
   (d) rdi, rsi, rdx, rcx

6. The ~ operator performs what operation on a value?

   (a) bitwise complement
   (b) logical complement
   (c) reverses the order of the bits
   (d) determines if the number is zero

7. What byte ordering system do the fish machines use?

   (a) Little endian
   (b) Big endian
   (c) Intel x86_64
   (d) at&t syntax

8. In the Intel IA32 architecture, function arguments are passed

   (a) on the stack
   (b) in registers
   (c) on the hard drive
   (d) on the heap

9. Placing a breakpoint on an instruction with GDB halts program execution

   (a) before the specified instruction is executed
   (b) immediatly after the specified instruction is executed
   (c) while the specified instruction is executed
   (d) GDB cannot place breakpoints

10. 32-bit systems can support 64-bit assembly code

    (a) TRUE
    (b) FALSE

11. Assuming the register %rbx contains the value 0xf123f234f345f456, which instruction would cause the register %rdi to contain the value 0xffffffff345f456?

    (a) `movl %ebx, %rdi`
    (b) `movslq %ebx, %rdi`
    (c) `movzlq %ebx, %rdi`
    (d) `lea %ebx, %rdi`

Problem 6. (14 points):

Throughout this question, remember that it might help you to draw a picture. It helps us see what you're thinking when we grade you, and you'll be more likely to get partial credit if your answers are wrong.

Consider the following C code:

```
void foo(int a, int b, int c, int d) {
  int buf[16];
  buf[0] = a;
  buf[1] = b;
  buf[2] = c;
  buf[3] = d;
  return;
}

void bar() {
  foo(0x15213, 0x18243, 0xdeadbeef, 0xcafebabe)
}
```

When compiled with default options (32-bit), it gives the following assembly:

```
00000000 <foo>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   83 ec 40                sub    $0x40,%esp

   6:   8b 45 08                mov    _____(%ebp),%eax //temp = a;
   9:   89 45 c0                mov    %eax,-0x40(%ebp) //buf[0] = temp;

   c:   8b 45 0c                mov    _____(%ebp),%eax //temp = b;
   f:   89 45 c4                mov    %eax,-0x3c(%ebp) //buf[1] = temp;

  12:   8b 45 10                mov    _____(%ebp),%eax //temp = c;
  15:   89 45 c8                mov    %eax,-0x38(%ebp) //buf[2] = temp;

  18:   8b 45 14                mov    _____(%ebp),%eax //temp = d;
  1b:   89 45 cc                mov    %eax,-0x34(%ebp) //buf[3] = temp;
  1e:   c9                      leave
  1f:   c3                      ret

00000020 <bar>:
  20:   55                      push   %ebp
  21:   89 e5                   mov    %esp,%ebp
  23:   83 ec 10                sub    $0x10,%esp
  26:   c7 44 24 0c be ba fe ca movl   $0xcafebabe,0xc(%esp)
  2e:   c7 44 24 08 ef be ad de movl   $0xdeadbeef,0x8(%esp)
  36:   c7 44 24 04 43 82 01 00 movl   $0x18243,0x4(%esp)
  3e:   c7 04 24 13 52 01 00    movl   $0x15213,(%esp)
  45:   e8 fc ff ff ff          call   foo
  4a:   c9                      leave
  4b:   c3                      ret
```

a) Very briefly explain what purpose is served by the first three lines of the disassembly of foo (just repeating the code in words is not sufficient). No more than one sentence should be necessary here.

b) Note that in foo (C version), each of the four arguments are accessed in turn. The assembly dump of foo is commented to show where this is done. Recall that the current `%ebp` value points to where the pushed old base pointer resides, and immediately above that is the return address from the function call. Write into the gaps in the disassembly of foo the offsets from `%ebp` needed to access each of the four arguments `a`, `b`, `c`, and `d`. (Hint: Look at how they are arranged in bar before the call.)

GCC has a compile option called `-fomit-frame-pointer`. When given this flag in addition to the previous flags, the function foo is compiled like this:

```
00000000 <foo>
83 ec 40      sub    $0x40,%esp

8b 44 24 44   mov    ____(%esp),%eax //temp = a;
89 04 24      mov    %eax,(%esp)     //buf[0] = temp;

8b 44 24 48   mov    ____(%esp),%eax //temp = b;
89 44 24 04   mov    %eax,0x4(%esp)  //buf[1] = temp;

8b 44 24 4c   mov    ____(%esp),%eax //temp = c;
89 44 24 08   mov    %eax,0x8(%esp)  //buf[2] = temp;

8b 44 24 50   mov    ____(%esp),%eax //temp = d;
89 44 24 0c   mov    %eax,0xc(%esp)  //buf[3] = temp;
83 c4 40      add    $0x40,%esp
c3            ret
```

c) What is the difference between the first few lines of foo in the first compilation and in this compilation? What does this mean about what the stack frame looks like? (Consider drawing a before/after picture.)

d) Note what has changed in how the arguments `a`, `b`, `c`, `d` and the stack-allocated buffer are accessed: they are now accessed relative to `%esp` instead of `%ebp`. Considering that the arguments are in the same place when foo starts as last time, and recalling what has changed about the stack this time around (note: the pushed return address is still there!), fill in the blanks on the previous page to correctly access the function's arguments.

e) Consider what the compiler has done: `foo` is now using its stack frame without dealing with the base pointer at all... and, in fact, all functions in the program compiled with `-fomit-frame-pointer` also do this. What is a benefit of doing this? (0-point bonus question: What is a drawback?)