CS 15-213, Spring 2009
Exam 1
Tuesday, 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!

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(16):</td>
</tr>
<tr>
<td>2</td>
<td>(22):</td>
</tr>
<tr>
<td>3</td>
<td>(13):</td>
</tr>
<tr>
<td>4</td>
<td>(13):</td>
</tr>
<tr>
<td>5</td>
<td>(22):</td>
</tr>
<tr>
<td>6</td>
<td>(14):</td>
</tr>
</tbody>
</table>

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. If, 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.

<table>
<thead>
<tr>
<th>Description</th>
<th>Decimal</th>
<th>Binary Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bias</td>
<td></td>
<td>— — — —</td>
</tr>
<tr>
<td>Smallest positive number</td>
<td></td>
<td>— — — —</td>
</tr>
<tr>
<td>Lowest finite</td>
<td></td>
<td>— — — —</td>
</tr>
<tr>
<td>Smallest positive normalized</td>
<td>— — — —</td>
<td>(-\frac{7}{16})</td>
</tr>
<tr>
<td></td>
<td>— — — —</td>
<td>(\frac{5}{4})</td>
</tr>
<tr>
<td></td>
<td>— — — —</td>
<td>1 010 01</td>
</tr>
<tr>
<td></td>
<td>— — — —</td>
<td>13</td>
</tr>
</tbody>
</table>
Problem 2. (22 points):
Consider the C code written below and compiled on a 32-bit Linux system using GCC.

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

struct s2
{
  struct s1 a;
  struct s1 *b;
  int x;
  char c;
  int y;
  char e[3];
  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 \texttt{fun1}, \texttt{fun2}, \texttt{fun3}, and \texttt{fun4}?

```
push %ebp
mov %esp,%ebp
mov 0x8(%ebp),%eax
add $0x1c,%eax
pop %ebp
ret

ANSWER: ________

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

ANSWER: ________

push %ebp
mov %esp,%ebp
mov 0x8(%ebp),%eax
mov 0x1c(%eax),%eax
pop %ebp
ret

ANSWER: ________

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

ANSWER: ________
```
d) Assume a variable is declared as `struct s2 myS2;` and the storage for this variable begins at address `0xbfb2ffc0`.

```
(gdb) x/20w &myS2
0xbfb2ffc0: 0x0000000f 0x000000d5 0xbfb2ffe8 0x00000000
0xbfb2ff0: 0xb7f173ff 0x0000012c 0xbf030102 0x0000000c
0xbfb2ffe0: 0xb7e2dfe5 0xb7f1ff4 0xbfb30012 0x000000f3
0xbfb2ff0: 0xb7e2eb9 0xb7f1ff4 0xbfb30058 0xb7e1adce
0xbfb30000: 0x00000001 0xbfb30084 0xbfb3008c 0xbfb30010
```

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):

Given the memory dump and disassembly from GDB on the next page, fill in the C skeleton of the function switchfn:

```c
int switchfn(int a, long b) {
    int y = 0, x = ____________;
    switch (a * b) {
        case 1:
            return 24;
        case 6:
            a = ____________;
            return a;
        case 0:
            return a + b;
        case 4:
            x = a;
            y *= b;
            break;
        case ____:
            a = y == x;
        case 3:
            b = y ____ x;
        case 5:
            return a ____ b;
    }
    return x == y;
}
```
There may be a few instructions you haven’t seen before in this assembly dump. \texttt{data16} is functionally equivalent to \texttt{nop}. \texttt{setcc} functions similarly to \texttt{jcc} except it will set its operand to 1 or 0 instead of jumping or not jumping, respectively. \texttt{cqto} is the 64-bit equivalent of \texttt{cltd}.


gdb) x/7xg 0x4005c0
0x4005c0 <_IO_stdin_used+8>:     0x00000000004004a1  0x0000000000400494
0x4005d0 <_IO_stdin_used+24>:    0x000000000004004ac  0x00000000004004b4
0x4005e0 <_IO_stdin_used+40>:    0x00000000004004a5  0x00000000004004bc
0x4005f0 <_IO_stdin_used+56>:    0x000000000040049a

0x0000000000040476 <switchfn+0>: mov $0x0, %ecx
0x000000000004047b <switchfn+5>: mov $0xdeadbeef, %edx
0x0000000000040480 <switchfn+10>: movslq %edi, %rax
0x0000000000040483 <switchfn+13>: imul %rsi, %rax
0x0000000000040487 <switchfn+17>: cmp $0x6, %rax
0x000000000004048b <switchfn+21>: ja 0x4004c5 <switchfn+79>
0x000000000004048d <switchfn+23>: jmpq *0x4005c0, (%rax, 8)
0x0000000000040494 <switchfn+30>: mov $0x18, %eax
0x0000000000040499 <switchfn+35>: retq
0x000000000004049a <switchfn+36>: lea (%rdx, %rsi, 4), %eax
0x000000000004049d <switchfn+39>: data16
0x000000000004049e <switchfn+40>: data16
0x000000000004049f <switchfn+41>: nop
0x00000000000404a0 <switchfn+42>: retq
0x00000000000404a1 <switchfn+43>: lea (%rdi, %rsi, 1), %eax
0x00000000000404a4 <switchfn+46>: retq
0x00000000000404a5 <switchfn+47>: mov %edi, %edx
0x00000000000404a7 <switchfn+49>: imul %esi, %ecx
0x00000000000404aa <switchfn+52>: jmp 0x4004c5 <switchfn+79>
0x00000000000404ac <switchfn+54>: cmp %edx, %ecx
0x00000000000404ae <switchfn+56>: sete %al
0x00000000000404b1 <switchfn+59>: movzbl %al, %edi
0x00000000000404b4 <switchfn+62>: cmp %edx, %ecx
0x00000000000404b6 <switchfn+64>: setl %al
0x00000000000404b9 <switchfn+67>: movzbl %al, %esi
0x00000000000404bc <switchfn+70>: movslq %edi, %rax
0x00000000000404bf <switchfn+73>: cqto
0x00000000000404c1 <switchfn+75>: idiv %rsi
0x00000000000404c4 <switchfn+78>: retq
0x00000000000404c5 <switchfn+79>: cmp %ecx, %edx
0x00000000000404c7 <switchfn+81>: sete %al
0x00000000000404ca <switchfn+84>: movzbl %al, %eax
0x00000000000404cd <switchfn+87>: retq
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.

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

mystery_sort:
    jmp  loop1_check

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

loop2:
    mov  (%rdi, %rcx, 8), %rax
    cmp  %rax, (%rdi, %rdx, 8)
    jg   loop2_check
    mov  %rcx, %rdx

loop2_check:
    dec  %rcx
    test  %rcx, %rcx
    jnz  loop2

    dec  %rsi
    mov  (%rdi, %rsi, 8), %rax
    mov  (%rdi, %rdx, 8), %rcx
    mov  %rcx, (%rdi, %rsi, 8)
    mov  %rax, (%rdi, %rdx, 8)

loop1_check:
    test  %rsi, %rsi
    jnz  loop1

    ret
```

Page 9 of 17
void mystery_sort (long* array, long len)
{
    long a, b, tmp;

    while (_____ > _____)
    {
        a = _____;

        for (b = _____; b > _____; b--)
        {
            if (array[_____] > array{_____.})
            {
                _____ = _____;
            }
        }

        len--;  
        tmp = array[_____.];
        array[_____.] = array[_____.];
        array[_____.] = tmp;
    }
}
Problem 5. (22 points):
Circle the correct answer.

1. What sequence of operations does the leave instruction execute?
   (a) mov %ebp,%esp
       pop %ebp
   (b) pop %ebp
       mov %ebp,%esp
   (c) pop %esp
       mov %ebp,%esp
   (d) push %ebp
       mov %esp,%ebp

2. Who is responsible for storing the return address of a function call?
   (a) the caller
   (b) the callee
   (c) the kernel
   (d) the CPU

3. On what variable types does C perform logical right shifts?
   (a) signed types
   (b) unsigned types
   (c) signed and unsigned types
   (d) C does not perform logical right shifts

4. What is the difference between the rbx and the ebx register on an x86_64 machine?
   (a) nothing, they are the same register
   (b) ebx refers to only the low order 32 bits of the rbx register
   (c) they are totally different registers
   (d) ebx refers to only the high order 32 bits of the rbx register

5. Which of the following is the name for the optimization performed when you pull code outside of a loop?
   (a) code motion
   (b) loop expansion
   (c) dynamic programming
   (d) loop unrolling
6. On 32-bit x86 systems, where is the value of %ebp saved in relation to the current value of %ebp?
(Assume a pointer size of 32 bits.)
   (a) there is no relation between where the current base pointer and old base pointer are saved.
   (b) old ebp = (ebp - 4)
   (c) old ebp = (ebp + 4)
   (d) old ebp = (ebp)

7. Which of the following mov instructions is invalid?
   (a) mov %esp, %ebp
   (b) mov $0xdeadbeef, %eax
   (c) mov (0xdeadbeef), %esp
   (d) mov $0xdeadbeef, 0x08048c5f
   (e) mov %ebx, 0x08048c5f

8. In C, the result of shifting a value by greater than its type’s width is:
   (a) illegal
   (b) undefined
   (c) 0
   (d) Encouraged by the C1x standard.

9. Extending the stack can be done by
   (a) swapping the base pointer and the stack pointer
   (b) subtracting a value from your stack pointer
   (c) adding a value to your stack pointer
   (d) executing the ret instruction

10. 64-bit systems can support 32-bit assembly code
    (a) TRUE
    (b) FALSE

11. Assuming the register %rbx contains the value 0xfaaafbbfccccfddd, which instruction would cause
    the register %rdi to contain the value 0x00000000fcffddd?
    (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:

```c
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?)