CSE351 Spring 2010 – Midterm Exam (3 May 2010)

Please read through the entire examination first! We designed this exam so that it can be completed in 50 minutes and, hopefully, this estimate will prove to be reasonable.

There are 3 problems for a total of 100 points. The point value of each problem is indicated in the table below. Write your answer neatly in the spaces provided. If you need more space (you shouldn't), you can write on the back of the sheet where the question is posed, but please make sure that you indicate clearly the problem to which the comments apply. Do NOT use any other paper to hand in your answers. If you have difficulty with part of a problem, move on to the next one. They are mostly independent of each other.

For problem 3, you may find it convenient to separate the page of assembly code from the rest of the exam (you do not need to turn in that page as no answers are to be written on that page) so that you can refer to it more easily for subsequent questions.

The exam is CLOSED book and CLOSED notes. Please do not ask or provide anything to anyone else in the class during the exam. Make sure to ask clarification questions early so that both you and the others may benefit as much as possible from the answers.

Name: _____

ID#: _____

Problem	Max Score	Score
1	35	
2	20	
3	45	
TOTAL	**100**	

1. Number Representation (35 points)

While on a boat, two circuit elements within the boat's 8-bit navigation computer break. Miraculously, the computer still seems to work, except that when it reads or writes any value, the two least significant bits are always 0.

For instance, this computer would erroneously compute:

```
  0000 0001
+ 0000 0011
-----------
  0000 0000
```

Notice that the computer also failed to carry from the two LSBs, since it read them as 0.

a) (5 pts) What two's complement integers will this computer read correctly?

Any positive or negative number that is divisible by 4 (two LSBs are 0).

b) (10 pts) The boat uses a floating point system with 1 sign bit, 3 bits for the exponent, and 4 bits for the significand. What positive denormalized values will it read correctly? Recall that denormalized values will have an exponent of 000 and that the bias for a 3 bit exponent is $2^{3-1}-1 = 3$.

Denormalized numbers have an exponent that is all 0s, implying an exponent equal to 1-bias, or 1-3, or -2 in this case. There is a leading 0 implied before the binary point.
0 000 1000 = $0.1000_2 * 2^{-2} = 1/8$
0 000 0100 = $0.0100_2 * 2^{-2} = 1/16$
0 000 1100 = $0.1100_2 * 2^{-2} = 3/16$

c) (10 pts) What is the smallest normalized value you can represent with this broken system?

By smallest we mean closest to zero. The smallest normalized values, positive and negative, will have the same absolute value (only the sign bit changes). So, we'll stick with the positive fraction. Normalized values have a smallest possible exponent of 1. When we subtract the bias of 3, that yields -2. There is also a leading 1 implied before the binary point: 0 001 0000 = $1.0000_2 * 2^{-2} = 1/4$

d) (10 pts) While on your boat, you discover that one of your guest lives at the south pole. Fortunately, the navigation computer represents latitude with double precision, with 1 bit for the sign, 7 bits for the exponent, and 8 bits for the significand. This means that the double precision value will occupy two bytes (and each of the two bytes will have 0s in the two LSBs – still broken). What is now the closest latitude to the south pole that the computer will represent? The south pole has a latitude of -90. Recall that with a 7 bit exponent the exponent bias will be $2^{7-1}-1=63$.

- What positive exponents can you represent correctly? _____
- What is the value of the sign bit? _____
- What is the biggest significand you can represent correctly? _____
- What is the number closest to -90 that you can represent? _____
- What will be the two byte encoding of this number?
  Sign (1 bit) and Exponent (7 bits) _____
  Significand (8 bits) _____

Exponents will have to be divisible by 4 as they are represented in the 7 LSBs of the first byte of the double precision number (the MSB is the sign bit). Therefore, we can represent 64, 68, 72, 76, etc. corresponding to exponents of 1, 5, 9, 13, etc.

The sign bit is 1 as we want to represent a negative number close to -90.

The significand byte is 11111100 because of our computer's flaw in the 2 LSBs. This corresponds to a significand of 1.11111100.

Thus, the closest number to -90 will be $-1.11111100_2 * 2^5 = -111111.1_2 = -63.5$.
Not very close at all to -90. Our guest will have a long swim to the south pole.

The encoding will be: 1 1000100 11111100

2. Stack Frames (20 points)

Given the following disassembly of a short function:

```
0x00001fb6 <func+0>:    push   %ebp
0x00001fb7 <func+1>:    movl   %esp,%ebp
0x00001fb9 <func+3>:    subl   $0x8,%esp
0x00001fbc <func+6>:    movl   0x8(%ebp),%eax
0x00001fbf <func+9>:    addl   %eax,%eax
0x00001fc1 <func+11>:   leave
0x00001fc2 <func+12>:   ret
```

and the following stack, `%ebp`, and `%esp` *right after* executing the instruction, `<func+3>`, at address `0x00001fb9`:

```
%esp = 0x300
%ebp = 0x308

0x300: 0x00000000
0x304: 0x00000000
0x308: 0x00000328
0x30c: 0x00001fdc
0x310: 0x00000003
```

a) (8 pts) What is the value in `%eax` (the return value of the function) after we return and leave the function (after instruction `<func+12>`)?

The value of `%eax` *is 6. The last instruction that touches* `%eax` *in this function is* `<func+9>`*, the add instruction. It basically doubles what was in* `%eax` *before. What was in* `%eax` *before was loaded from memory (the stack), 8 bytes after* `%ebp`*, which is* `0x308 + 8 = 0x310`*. The value 3 is at that location, so 3 is doubled to yield 6.*

b) (6 pts) What was the old value of the stack pointer (`%esp`) before we executed the first instruction of this function (at `<func+0>`)?

The value of `%esp` *was* `0x30c`*. At instruction* `<func+3>`*, we subtracted 8 bytes from the stack pointer (*`%esp`*) for temporary scratch space. That accounts for the two* `0x00000000` *values at the top of the stack. The instruction at* `<func+0>` *pushed the old value of* `%ebp` *onto the stack. Therefore we subtracted another 4 bytes from the stack pointer. This means that the old* `%esp` *was 12 bytes more than the current location.* `0x300 + 8 + 4 = 0x30c`

c) (6 pts) What is the old value of the base pointer (`%ebp`) before we entered this function?

The value of `%ebp` *was* `0x328`. *At instruction* `<func+0>` *(*`push %ebp`*), we pushed the old* `%ebp` *to the top of the stack by first decrementing the stack pointer to point to* `0x308` *(recall, it started at* `0x30c`*). The value stored at* `0x308` *is the old value of* `%ebp`.

3. Analyzing and Extending Assembly Code (45 points)

For this problem, you'll examine the assembly code for the procedure "walk", an interpreter of instructions for a simple robot. The robot keeps track of its current (x, y) position on the Cartesian plane, and its current direction:

- north (encoded as 0; facing towards higher y coordinates),
- east (encoded as 1; facing towards higher x coordinates),
- south (encoded as 2; facing towards lower y coordinates), or
- west (encoded as 3; facing towards lower x coordinates).

The "walk" function (shown on the next page) takes as arguments:

- an initial x coordinate,
- an initial y coordinate,
- an initial direction, and
- an array of instructions (encoded as unsigned integers, ending in a 0)

that tell it how to move.

The procedure reads and decodes each instruction in turn, branching to the relevant code for each instruction and updating the robot's position and direction accordingly. The instruction set is very limited:

- HALT (encoded as 0) tells the robot to stop reading instructions and print its ending position.
- RIGHT (encoded as 1) tells the robot to stay in place and turn 90 degrees to the right. For example, if the robot reads a RIGHT instruction when it is at (2, 5), headed north, it will remain at position (2, 5) and turn to face east.
- STEP (encoded as 2) tells the robot to take one step in its current direction. For example, if the robot reads a STEP instruction when it is at (2, 4), headed north, it will move to (2, 5), still facing north.

Starting at (0, 0) headed north and given the 8 encoded instructions: 2, 1, 2, 2, 1, 2, 2, 0; the robot halts at (2, –1), headed south. Use this input/output and the assembly code for walk on the next page to answer the questions on the page after that.

```
void walk(int x, int y, unsigned dir, unsigned instr[])
```

```
walk:
     pushl %ebp
     movl  %esp, %ebp
     subl  $40, %esp
     movl  $0, -12(%ebp)
     jmp   .L15
.L25:
     movl  -12(%ebp), %eax
     movl  20(%ebp,%eax,4), %eax
     cmpl  $1, %eax
     je    .L16
     cmpl  $2, %eax
     je    .L17
```

[AAA:]

```
.L15:
     movl  -12(%ebp), %eax
     movl  20(%ebp,%eax,4), %eax
     testl %eax, %eax
     jne   .L25
     [Print current position]
     leave
     ret
```

```
.L16:
     movl  16(%ebp), %eax
     addl  $1, %eax
     andl  $3, %eax
     movl  %eax, 16(%ebp)
     addl  $1, -12(%ebp)
     jmp   .L15
.L17:
     movl  16(%ebp), %eax
     cmpl  $0, %eax
     je    .L20
     cmpl  $1, %eax
     je    .L21
     cmpl  $2, %eax
     je    .L22
     cmpl  $3, %eax
     je    .L23
     jmp   .L19
.L20:
     addl  $1, 12(%ebp)
     jmp   .L19
.L21:
     addl  $1, 8(%ebp)
     jmp   .L19
.L22:
     subl  $1, 12(%ebp)
     jmp   .L19
.L23:
     subl  $1, 8(%ebp)
.L19:
     addl  $1, -12(%ebp)
     jmp   .L15
```

[BBB:]

3. Analyzing and Extending Assembly Code (continued) (45 points)

a) (8 pts) For each parameter to walk, what is the corresponding memory address used in the assembly code (in terms of `%ebp`)?

x: _____

y: _____

dir: _____

instr: _____

x: `8(%ebp)`*; y:* `12(%ebp)`*; dir:* `16(%ebp)`*; instr:* `20(%ebp)`*;*

b) (20 pts) Describe what each of these sections of code does, in 1-2 sentences each:

- `.L25` to through "`je .L17`"

  This code reads the next instruction from the instructions array (which starts at `20(%ebp)` *and is indexed by the variable* `i` *stored in* `-12(%ebp)`*), and switches on this instruction code, branching to the case to handle that instruction.*

- `.L15` through "`jne .L25`"

  This is the loop test that checks to see if the next is a HALT (0) that will cause the procedure to return after doing some printing.

- `.L16` to `.L17`

  This implements the RIGHT command by first incrementing `dir` *(modulo 4 – implemented using the* `andl` *instruction) and then incrementing* `i` *(stored in* `-12(%ebp)`*) for indexing the next instruction in the array.*

- `.L17` through "`jmp .L15`"

  This part of the code implements the STEP instruction. It must first check `dir` *to determine whether to inc/dec* `x` *or* `y`*. It also concludes by incrementing* `i` *(stored in* `-12(%ebp)`*) for indexing the next instruction in the array.*

c) (17 pts) Implement code for a new IFNORTH instruction for the robot (encoded as 3). When the robot reads IFNORTH at index `i` in the instruction array:

- if the robot is facing north, it replaces its current instruction index with the integer stored at index `i + 1` in the instruction array and continues processing instructions starting at this new index; else

- if the robot is not facing north, it continues processing instructions starting at index `i + 2` in the instruction array.

For example, the instruction array IFNORTH(3), 3, STEP(2), HALT(0) should cause the robot to halt in its current position if it is facing north (the next instruction executed is the HALT at `i = 3`) or, if it is not facing north, take one step forward (the next instruction executed is the STEP at `i = 2`) and then halt.

We'll insert the following instructions at the location marked AAA in the assembly code:

```
cmpl  $3, %eax
je    .L30
```

Write the assembly code to implement the IFNORTH instruction (show the instructions to be inserted in the assembly code at the location marked BBB in the space provided below). Of course, they will start at the label `.L30`.

```
.L30:
     movl  16(%ebp), %eax              // eax = dir
     cmpl  $0, %eax                    // if dir != 0
     jne   .L31                        // goto .L31
     movl  -12(%ebp), %eax             // eax = i
     addl  $1, %eax                    // eax++
     movl  20(%ebp,%eax,4), %eax       // eax = instr[eax]
     movl  %eax, -12(%ebp)             // i = eax
     jmp   .L15                        // goto .L15
.L31
     movl  -12(%ebp), %eax             // eax = i
     addl  $2, %eax                    // eax = eax + 2
     movl  %eax, -12(%ebp)             // i = eax
     jmp   .L15                        // goto .L15
```