CSE351 Autumn 2010 – Midterm Exam (29 Oct. 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 worth 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.

The last page of the test contains a page of powers-of-two and reminders about some common x86 instructions. Feel free to separate that page from the exam if it is convenient. Pages 4 and 6 contain code that is the subject of the last two questions. These pages can also be detached if convenient.

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: Sample Soution

Problem	Max Score	Score
1	32	
2	28	
3	40	
TOTAL	**100**	

1. Floating-Point Number Representation (32 points)

Suppose we have a 7-bit computer that uses IEEE floating-point arithmetic where a floating point number has 1 sign bit, 3 exponent bits, and 3 fraction bits. All of the bits in the hardware work properly.

Recall that denormalized numbers will have an exponent of 000, and the bias for a 3-bit exponent is $2^{3-1} - 1 = 3$.

a) (18 points) For each of the following, write the binary value and the corresponding decimal value of the 7-bit floating point number that is the closest available representation of the requested number. If rounding is necessary use round-to-nearest. Give the decimal values either as whole numbers or fractions. The first line is filled in for you.

Number	Binary	Decimal
0	0 000 000	0.0
-0.125	**1 000 100**	**-0.125**
Smallest positive normalized number	**0 001 000**	**0.25**
Smallest positive denormalized number > 0	**0 000 001**	**1/32 (= 0.03125)**
Largest positive number < ∞	**0 110 111**	**15.0**
-3.1	**1 100 100**	**-3.0**
12.25	**0 110 100**	**12.0**

1. Floating-Point Number Representation (cont.) (32 points)

b) (8 points) The associative law for addition says that $a + (b + c) = (a + b) + c$. This holds for regular arithmetic, but does not always hold for floating-point numbers. Using the 7-bit floating-point system described above, give an example of three floating-point numbers a, b, and c for which the associative law does not hold, and show why the law does not hold for those three numbers.

There are several possible answers. Here's one.

Let a = 1 110 111, b = 0 110 111, and c = 0 000 001. Then $(a + b) + c = c$, because a and b cancel each other, while $a + (b + c) = 0$, because $b + c = b$ (c is very small relative to b and is lost in the addition).

c) (6 points) One of the novel features of IEEE floating-point arithmetic is the inclusion of denormalized numbers. Most pre-IEEE floating-point systems did not have these. On those systems, if the result of an arithmetic operation was too small to be represented as a normalized number, 0.0 was used instead.

Including denormalized numbers in floating-point arithmetic complicates the implementation, but it was considered important enough to make the extra complexity worthwhile. Why is this so? What advantage is there to including denormalized floating-point numbers instead of requiring all non-zero numbers to be normalized?

Denormalized numbers allow gradual underflow where small non-zero numbers that are nearly zero can be represented. This is particularly useful for results of calculations that are close to zero. It also allows floating-point numbers to represent an evenly spaced set of numbers near zero instead of having a gap around zero.

3. x86 Code and C (28 points)

One of the new interns managed to erase the only remaining copy of a C function. We do have an assembly language version of it, but need your help to reconstruct the original code. Here is the assembly language version:

```
mystery:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        movl    8(%ebp), %ecx
        movl    12(%ebp), %edx
        movl    16(%ebp), %ebx
        movl    %ecx, (%edx,%ebx,4)
        movl    $0, %eax
        cmpl    %ecx, (%edx)
        je      L4
        movl    $0, %eax
L5:
        incl    %eax
        cmpl    %ecx, (%edx,%eax,4)
        jne     L5
L4:
        cmpl    %ebx, %eax
        setl    %al
        movzbl  %al, %eax
        popl    %ebx
        leave
        ret
```

We have managed to reconstruct some of the C code. Your job is to fill in the blanks in the code on the next page to get a program equivalent to the generated code above.

You can detach this page if it is convenient – it does not need to be turned in.

Hints: Remember that the result of an integer-valued function is returned in register eax.

In C, and in the corresponding assembly language code, if a function has an array parameter, the actual argument is a pointer to the first element of the array.

It would be worth taking a minute to figure out which variables are held in which registers.

3. x86 Code and C (cont.) (28 points)

a) (24 points) Complete the C function below so it is equivalent to the x86 version given on the previous page. You should only write code in the given blank areas. Do not add to or rearrange the statements. (This function, with the blanks filled in, was used to generate the x86 code, although the compiler did change the order of the x86 code somewhat compared to the original C code.)

```
int mystery(int x, int A[], int n) {
  int k, result;


  A[n] = x ;
  k = 0;
  while ( A[k] != x ) {
    k++ ;      // or k=k+1;  or k+=1;
  }
  if ( k < n ) {
    result = 1;
  } else {
    result = 0;
  }
  return result;
}
```

b) (4 points) What does this function do, in terms of its parameters x, A, and n?

Performs a sentinel search of A[0..n-1] and returns true if x appears in that part of the array, and false otherwise.

3. Analyzing and Fixing Assembly Code (40 points)

The Seattle Department of Transportation has just upgraded the software on one of their traffic lights and it's not working. Fortunately you have been able to plug your laptop into the traffic light and examine the code with `gdb`. We've isolated the trouble to a function named `tick` that is supposed to change the light from green to yellow to red and back on a regular basis. Here is the heading and description of that function:

```
void tick(int *state, int *timer, int interval)
```

Function `tick` is called periodically. Each time it is called it decreases the value of `timer` by one. When `timer` reaches zero, it is reset to the value given by `interval`, and the light is changed to the next color in the sequence red -> green -> yellow -> red…. Variable `state` controls the current setting of the light, using the values 1 = red, 2 = green, and 3 = yellow.

We've disassembled the code for `tick` in `gdb` and this is what we've got. (Feel free to remove this page from the exam for convenience)

```
0x00001f1a <tick+0>:     push   %ebp
0x00001f1b <tick+1>:     mov    %esp,%ebp
0x00001f1d <tick+3>:     mov    0xc(%ebp),%edx
0x00001f20 <tick+6>:     mov    (%edx),%eax
0x00001f22 <tick+8>:     dec    %eax
0x00001f23 <tick+9>:     mov    %eax,(%edx)
0x00001f25 <tick+11>:    test   %eax,%eax
0x00001f27 <tick+13>:    jg     0x1f5a <tick+64>
0x00001f29 <tick+15>:    mov    0x8(%ebp),%ecx
0x00001f2c <tick+18>:    mov    (%ecx),%eax
0x00001f2e <tick+20>:    cmp    $0x1,%eax
0x00001f31 <tick+23>:    jne    0x1f40 <tick+38>
0x00001f33 <tick+25>:    mov    0x10(%ebp),%eax
0x00001f36 <tick+28>:    mov    %eax,(%edx)
0x00001f38 <tick+30>:    movl   $0x2,(%ecx)
0x00001f3e <tick+36>:    jmp    0x1f5a <tick+64>
0x00001f40 <tick+38>:    cmp    $0x3,%eax
0x00001f43 <tick+41>:    jne    0x1f55 <tick+59>
0x00001f45 <tick+43>:    mov    0x10(%ebp),%ecx
0x00001f48 <tick+46>:    mov    %ecx,(%edx)
0x00001f4a <tick+48>:    mov    0x8(%ebp),%eax
0x00001f4d <tick+51>:    movl   $0x1,(%eax)
0x00001f53 <tick+57>:    jmp    0x1f5a <tick+64>
0x00001f55 <tick+59>:    mov    0x10(%ebp),%ecx
0x00001f58 <tick+62>:    mov    %ecx,(%edx)
0x00001f5a <tick+64>:    leave
0x00001f5b <tick+65>:    ret
```

3. Analyzing and Fixing Assembly Code (continued) (40 points)

a) (6 pts) For each parameter to `tick`, what is the corresponding memory address used in the assembly code (in terms of `%ebp`)?

state: **%ebp + 8**

timer: **%ebp + 12 (0xc)**

interval: **%ebp + 16 (0x10)**

b) (18 pts) Describe what each of these sections of code does, in 1-2 sentences each:

- <tick+3> through <tick+13>

  Subtracts 1 from `*timer`. If the resulting value is greater than 0, jumps to the end of the function to return immediately.

- <tick+15> through <tick+23>

  Tests the value of `*state`. If it is not 1 (red), control jumps ahead to <tick+38>, otherwise execution falls through and continues at <tick+25>.

- <tick+43> through <tick+51>

  Resets `*timer` to `interval`, and sets `*state` to 1 (red).

3. Analyzing and Fixing Assembly Code (continued) (40 points)

c) (8 pts) The problem with the traffic light is that it eventually gets stuck on one of the colors red, green, or yellow, and does not change after that. Which color does it get stuck on, and why? What is the bug in the code?

The light gets stuck on green. The bug is in the code starting at <tick+59>. Control reaches here if the timer has been decremented to 0 and the light is green (`*state` = 2). This code resets `*timer` to `interval`, to start counting down again, but does not change `*state`, leaving the light still green.

d) (8 points) How would you change the x86 code to fix the bug so the light will properly cycle between red, green, and yellow? Describe which instructions you would change or delete (if any), or which new instructions need to be inserted (if any) to fix the problem, and where these changes would be made.

The shortest fix is to observe that `%ecx` still contains the pointer to `state` when control reaches <tick+59>, so all we need to do to change the state to 3 is to insert the following instruction at <tick+59>, right before the `mov` instruction:

```
mov   $3,(%ecx)
```

Another approach would be to insert these two instructions either at <tick+59> or following the `mov` instruction at <tick+62>:

```
mov   8(%ebp),%ecx   ; target register could also be %eax or %edx
mov   $3,(%ecx)
```