CSE351 Autumn 2012 – Midterm Exam (5 Nov 2012)

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 4 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 independent of each other.

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 Solution***______

ID#: _________________________

Problem	**Max Score**	**Score**
1	15	***15***
2	20	***20***
3	40	***40***
4	25	***25***
TOTAL	**100**	***100***

1. Number Representation (15 points)

The decimal value 11,184,810 is represented as a 32-bit signed binary with the bit pattern below (0x00aaaaaa):

```
0000   0000   1010   1010   1010   1010   1010   1010
```

When it is cast as a float, it is represented by the 32-bit floating point format (8-bits exp, 23-bit fraction) as (0x4b2aaaaa):

```
0100   1011   0010   1010   1010   1010   1010   1010
```

Explain why so many of the low-order bits are the same and why do the others differ. There is no need to convert these to decimal values.

> *The exponent part of the float value is 1001 0110 which translates to E = 150 – bias = 150 – 127 = 23. Given that the value of the float is 1.frac * 2^{23}, and the fractional part is 23 bits, the binary point moves 23 bits to the right. Therefore, we expect to see the same 23 low-order bits in the integer value with the next high-order bit being 1 (the implied 1. of the float representation). The remaining 8 high-order bits of the integer value are 0.*

2. Assembly Code (20 points)

A function ‘flip’ has the following overall structure:

```
int flip (*unsigned x) {
     int num=*x;
     int val=0;
     int i;
     for (__initialize__; __test__; __update__) {
          __body__
     }
     return val;
}
```

The GCC C compiler generates the following assembly code:

x at %ebp+8

```
        movl   8(%ebp), %ebx          get x, a pointer to an unsigned int
        movl   (%ebx), %esi           dereference x to get the unsigned int into num
        movl   $0, %eax               initialize val to 0, stored in %eax
        movl   $0, %ecx               initalize i to 0, stored in %ecx
  .L13:
        leal   (%eax, %eax), %edx     double val, like shifting 1 left, put in %edx
        movl   %esi, %eax             copy num to %eax
        andl   $1, %eax               mask to get low-order bit of num
        orl    %edx, %eax             OR the low-order bit with doubled val
        shrl   %esi                   shift num to right by 1
        add    $1, %ecx               increment i
        cmpl   $32, %ecx              test if reached 32
        jne    .L13                   jump to L13 (top of loop) if i < 32
        ret                           return with val stored in %eax
```

Reverse engineer the operation of this code and then do the following:

A (15 pts). Use the assembly-code version to fill in the missing parts of the C code below. Also specify which lines above represent each of initialize, test, update, and body.

Initialize:	____*4*__________
Test:	____*12, 13*______
Update:	____*11*_________
Body:	____*6, 7, 8, 9, 10*_

```
int flip (*unsigned x) {
     int num=*x;
     int val=0;
     int i;
     for ( ___i=0_______ ; ___i<32_____ ; ___i++_______ ) {

          ___val = (val << 1) | (num & 0x1);___

          ___num = num >> 1;___________________

     }
     return val;
}
```

B (5 pts). Describe what this function computes in one English sentence (or at most two).

The function returns an int that has the same bits as stored at x but in reverse order.

3. Procedures (40 points)

The following assembly routine takes a positive integer as input and returns a positive integer:

```
0000000000400525 <mystery>:
  400525:   55                      push   %rbp
  400526:   48 89 e5                mov    %rsp,%rbp
  400529:   53                      push   %rbx
  40052a:   48 83 ec 18             sub    $0x18,%rsp

  40052e:   89 7d ec                mov    %edi,-0x14(%rbp)
  400531:   83 7d ec 00             cmpl   $0x0,-0x14(%rbp)
  400535:   75 07                   jne    40053e <mystery+0x19>
  400537:   b8 00 00 00 00          mov    $0x0,%eax
  40053c:   eb 2b                   jmp    400569 <mystery+0x44>
  40053e:   83 7d ec 01             cmpl   $0x1,-0x14(%rbp)
  400542:   75 07                   jne    40054b <mystery+0x26>
  400544:   b8 01 00 00 00          mov    $0x1,%eax
  400549:   eb 1e                   jmp    400569 <mystery+0x44>

  40054b:   8b 45 ec                mov    -0x14(%rbp),%eax
  40054e:   83 e8 01                sub    $0x1,%eax
  400551:   89 c7                   mov    %eax,%edi
  400553:   e8 cd ff ff ff          callq  400525 <mystery>
  400558:   89 c3                   mov    %eax,%ebx
  40055a:   8b 45 ec                mov    -0x14(%rbp),%eax
  40055d:   83 e8 02                sub    $0x2,%eax
  400560:   89 c7                   mov    %eax,%edi
  400562:   e8 be ff ff ff          callq  400525 <mystery>
  400567:   01 d8                   add    %ebx,%eax

  400569:   48 83 c4 18             add    $0x18,%rsp
  40056d:   5b                      pop    %rbx
  40056e:   5d                      pop    %rbp
  40056f:   c3                      retq
```

A (5 pts). Does this assembly code appear to follow the 32-bit or 64-bit parameter-passing guidelines? How can you tell?

> *The function uses mostly 64-bit registers. Also, input arguments are passed using %edi which indicates parameter passing through registers another sign of a 64-bit architecure. Don't be confused by the presence of the base pointer register, %rbp. A frame base pointer can still be used in the 64-bit architecture.*

B (5 pts). Why is %rbx pushed onto the stack initially and then popped at the end?

%rbx is a callee-saved register, and the code uses %rbx (or $ebx) so it must save it before it overwrites it and then restore it before it returns.

C (5 pts). There are two if() statements in the code that produced this assembly. At which instruction addresses do they begin?

0x400531 and 0x40053e, where both of the comparisons occur.

D (5 pts). What does the byte 'ec' most likely correspond to in the instruction at `0x40055a`?

'ec' is the twos-complement version of -0x14, the offset being used in the move instruction at that address.

E (15 pts). Write out C code that would assemble into the routine above.

```
unsigned int mystery(unsigned int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;
  return ( mystery(n-1) + mystery(n-2) );
}
```

F (5 pts). What does this function do?

It returns the nth Fibonacci number (0,1,1,2,3,5,8,...).

4. Stack Discipline (25 points)

Consider a stack from an IA32 machine with the following contents:

Line ref number	Address in memory	Value in memory	Check if ret addr	Check if arg or local var	Check if saved ebp
22	`0xfffffffc`	`0x00000001`		✓	
21	`0xfffffff8`	`0x00000005`		✓	
20	`0xfffffff4`	`0xfffffffc`		✓	
19	`0xfffffff0`	`0x004080a0`	✓		
18	`0xffffffec`	`0xfffffffc`			✓
17	`0xffffffe8`	`0x00000005`		✓	
16	`0xffffffe4`	`0x0040801e`	✓		
15	`0xffffffe0`	`0xffffffec`			✓
14	`0xffffffdc`	`0x00000004`		✓	
13	`0xffffffd8`	`0x0040801e`	✓		
12	`0xffffffd4`	`0xffffffe0`			✓
11	`0xffffffd0`	`0x00000003`		✓	
10	`0xffffffcc`	`0x0040801e`	✓		
9	`0xffffffc8`	`0xffffffd4`			✓
8	`0xffffffc4`	`0x00000002`		✓	
7	`0xffffffc0`	`0x0040801e`	✓		
6	`0xffffffbc`	`0xffffffc8`			✓
5	`0xffffffb8`	`0x00800000`		✓	
4	`0xffffffb4`	`0x008000d0`		✓	
3	`0xffffffb0`	`0x00000001`		✓	
2	`0xffffffac`	`0x00000001`		✓	
1	`0xffffffa8`	`0x00408053`	✓		
	`0xffffffa4`				
	`0xffffffa0`				

Furthemore, you know that your code is in memory in locations from 0x00400000 to 0x005fffff and that your dynamic data heap is in locations 0x00800000 to 0x009fffff.

A (5 pts). Assume that machine execution has just been stopped just before the first instruction of a procedure. What address will we return to after that procedure completes?

The return address last placed on the stack in line 1, or 0x00408053.

B (5 pts). How much space did the calling procedure making this last call allocate on the stack for local variables and arguments? List the reference numbers of stack elements.

2, 3, 4, 5

C (10 pts). Annotate the stack on the previous page with the type of data stored at that location on the stack by placing a check mark in the appropriate column.

D (5 pts). Is there a recursive procedure on the stack? If so, how many calls deep is the recursion at the point represented by the stack above?

Yes, the four return addresses at lines 7, 10, 13, and 16 are the same indicating that we are four deep into a recursion.

REFERENCES

Powers of 2:

$2^0 = 1$	
$2^1 = 2$	$2^{-1} = .5$
$2^2 = 4$	$2^{-2} = .25$
$2^3 = 8$	$2^{-3} = .125$
$2^4 = 16$	$2^{-4} = .0625$
$2^5 = 32$	$2^{-5} = .03125$
$2^6 = 64$	$2^{-6} = .015625$
$2^7 = 128$	$2^{-7} = .0078125$
$2^8 = 256$	$2^{-8} = .00390625$
$2^9 = 512$	$2^{-9} = .001953125$
$2^{10} = 1024$	$2^{-10} = .0009765625$

Assembly Code Instructions:

push	push a value onto the stack and decrement the stack pointer
pop	pop a value from the stack and increment the stack pointer
call	jump to a procedure after first pushing a return address onto the stack
ret	pop return address from stack and jump there
mov	move a value between registers and memory
lea	compute effective address and store in a register
add	add 1^{st} operand to 2^{nd} with result stored in 2^{nd}
sub	subtract 1^{st} operand from 2^{nd} with result stored in 2^{nd}
and	bit-wise AND of two operands with result stored in 2^{nd}
or	bit-wise OR of two operands with result stored in 2^{nd}
shr	shift data by 1 bit to the right
jmp	jump to address
cmp	subtract 1^{st} operand from 2^{nd} and set flags
jne	conditional jump to address if zero flag is not set