CSE410 Spring 2012 – Midterm Exam (4 May 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 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 and conventions. Feel free to separate that page from the exam if it is convenient. Other pages containing code for questions can also be detached if convenient – the bottom of the page will indicate if this is okay.

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: ________________________

Problem	**Max Score**	**Score**
1	24	
2	26	
3	24	
4	26	
TOTAL	**100**	

1. Bits (24 points) The following two questions are similar to the questions in Lab 1 and the same ground rules apply:

- Assume all values are 32-bit integers.
- You may only use the operators !, ~, &, ^, |, +, <<, and >> and integer constants from 0 through 255 (0xFF). No other operators, control constructs, data structures, etc., and you may not call other functions.
- You may only use function arguments and any additional `int` local variables you declare. No global variables or other external data.

(a)

```
/* Return 1 if x is even or 0 if x is odd. */
/* Note: x may be positive or negative.    */

int isEven(int x) {




    return _______________________;

}
```

(b)

```
/* Return x if x >= 0. If x < 0 return 0. */

int nonNeg(int x) {




    return _______________________;

}
```

2. x86 and C code. (26 points)

One of the new interns managed to erase the only remaining copy of a C function. We do have a 32-bit x86 assembly language version of it, but we need your help to reconstruct the original code. Here is the assembly language version:

```
mystery:
     pushl   %ebp
     movl    %esp, %ebp
     subl    $16, %esp
     movl    $0, -8(%ebp)
     movl    8(%ebp), %eax
     movl    %eax, -4(%ebp)
     jmp     .L2
.L4:
     cmpl    $0, -4(%ebp)
     jns     .L3            # jns means jump if ~SF, i.e., jump if non-negative
     addl    $1, -8(%ebp)
.L3:
     sall    -4(%ebp)
.L2:
     cmpl    $0, -4(%ebp)
     jne     .L4
     movl    -8(%ebp), %eax
     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 create a C function equivalent to the generated code above, and then describe what the function does.

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.

It would be worth taking a minute to figure out where variables are stored in memory or registers.

2. x86 Code and C (cont.)

a) (20 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 arg) {
  int ans = 0;
  int n = arg;

  while ( ______________________________ ) {

    if ( ________________________ ) {

      ans = ________________________ ;

    }

    n = _______________________________ ;

  }

  return ans;

}
```

b) (6 points) What value does this function compute and return?

3. x86 Code. (24 points)

The following C function computes an integer value derived from the integers in an array:

```
int hash(int a[], int n) {
  int k;
  int ans = 0;
  for (k = 0; k < n; k++) {
    ans = 9*ans + a[k];
  }
  return ans;
}
```

When this code was compiled by gcc on an 32-bit x86 machine it produced the following assembly code, except that the body of the loop has been replaced by an empty box.

```
hash:
     pushl   %ebp
     movl    %esp, %ebp
     pushl   %ebx
     movl    8(%ebp), %ebx      # copy a to %ebx
     movl    12(%ebp), %ecx     # copy n to #ecx
     movl    $0, %eax
     movl    $0, %edx
     testl   %ecx, %ecx
     jle     .L3                # return if n <= 0
.L6:



.L3:
     popl    %ebx
     popl    %ebp
     ret
```

On the next page there are several sequences of code that might correctly implement the body of the loop. Your job is to look at each sequence and circle yes if it can be inserted in the empty box above to create a correct translation of the C code. Circle no if the code does not work properly. You do not need to supply any reasons for your answers.

You may remove this page from the exam for reference while working on the problem if that is convenient.

Question 3. (cont.) For each of the following blocks of code, circle yes if it correctly implements the loop body for the code on the previous page. Circle no if it does not.

(a) **Yes** **No**

```
imull   $9, %eax
addl    (%ebx,%edx,4), %eax
addl    $1, %edx
cmpl    %ecx, %edx
jne     .L6
```

(b) **Yes** **No**

```
leal    (%eax,%eax,8), %eax
leal    (%ebx,%edx,4), %edx
addl    (%edx), $eax
addl    $1, %edx
cmpl    %ecx, %edx
jne     .L6
```

(c) **Yes** **No**

```
leal    8(%eax,%eax), %eax
addl    (%ebx,%edx,4), %eax
addl    $1, %edx
cmpl    %ecx, %edx
jne     .L6
```

4. Analyzing Bugs in Assembly Code (26 points)

The boss at Apps 'R Us thinks we have a killer app on our hands. It reads a number from 1-12 representing a month of the year, and suggests an interesting activity for that month.

What it's supposed to do is to suggest the following activities for these months:

- 1, 2 (i.e., Jan., Feb.): go skiing
- 3: watch TV
- 4, 5, 6: walk
- 7, 8, 9: hike
- 10, 11: run
- 12: go skiing.

Unfortunately, all we've got is a x86-64 binary app on our smartphone that isn't working. Using the attached gdb debugger we've been able to disassemble the code and see that it reads an integer into location `0xc(%rsp)`, then attempts to use that to call appropriate functions to suggest the various activities.

```
Dump of assembler code for function main:
   0x000000000040058f <+0>:   sub    $0x18,%rsp
   0x0000000000400593 <+4>:   lea    0xc(%rsp),%rsi # int address
   0x0000000000400598 <+9>:   mov    $0x400762,%edi
   0x000000000040059d <+14>:  mov    $0x0,%eax
   0x00000000004005a2 <+19>:  callq  0x400430 <scanf> # read int
   0x00000000004005a7 <+24>:  cmpl   $0xc,0xc(%rsp)
   0x00000000004005ac <+29>:  ja     0x4005dc <main+77>
   0x00000000004005ae <+31>:  mov    0xc(%rsp),%eax
   0x00000000004005b2 <+35>:  jmpq   *0x400768(,%rax,8)
   0x00000000004005b9 <+42>:  mov    $0x0,%eax
   0x00000000004005be <+47>:  callq  0x400530 <ski>
   0x00000000004005c3 <+52>:  jmp    0x4005f2 <main+99>
   0x00000000004005c5 <+54>:  mov    $0x0,%eax
   0x00000000004005ca <+59>:  callq  0x400543 <hike>
   0x00000000004005cf <+64>:  nop           # does nothing
   0x00000000004005d0 <+65>:  jmp    0x4005f2 <main+99>
   0x00000000004005d2 <+67>:  mov    $0x0,%eax
   0x00000000004005d7 <+72>:  callq  0x400556 <run>
   0x00000000004005dc <+77>:  mov    $0x0,%eax
   0x00000000004005e1 <+82>:  callq  0x40057c <watch_tv>
   0x00000000004005e6 <+87>:  jmp    0x4005f2 <main+99>
   0x00000000004005e8 <+89>:  mov    $0x0,%eax
   0x00000000004005ed <+94>:  callq  0x400569 <walk>
   0x00000000004005f2 <+99>:  mov    $0x0,%eax
   0x00000000004005f7 <+104>: add    $0x18,%rsp
   0x00000000004005fb <+108>: retq
```

(You can remove this page for reference while continuing to work on the problem on the next page.)

4. Analyzing Bugs in Assembly Code (continued)

The code appears to reference memory located at 0x400768, so we also examined that part of memory using the debugger:

```
x /18gx 0x400768
0x400768:  0x00000000004005dc    0x00000000004005b9
0x400778:  0x00000000004005b9    0x00000000004005dc
0x400788:  0x00000000004005e8    0x00000000004005e8
0x400798:  0x00000000004005dc    0x00000000004005c5
0x4007a8:  0x00000000004005c5    0x00000000004005c5
0x4007b8:  0x00000000004005d2    0x00000000004005d2
0x4007c8:  0x00000000004005b9    0x000000000040063a
0x4007d8:  0x000000000040060e    0x000000000040060e
0x4007e8:  0x000000000040063a    0x0000000000400630
```

The code contains bugs that keep it from working as described at the beginning of the problem. Explain what it does wrong by giving the input numbers that cause it to call incorrect or extra functions, and describe what happens when those numbers are entered.

For full credit you need to succinctly describe what's wrong. Don't give a long-winded explanation of what every line of code does.

REFERENCE:

Powers of 2:

$2^0 = 1$	$2^7 = 128$
$2^1 = 2$	$2^8 = 256$
$2^2 = 4$	$2^9 = 512$
$2^3 = 8$	$2^{10} = 1024$
$2^4 = 16$	$2^{11} = 2048$
$2^5 = 32$	$2^{12} = 4096$
$2^6 = 64$	

Assembly Code Instructions:

pushl	push a value onto the stack
leave	restore ebp from the stack
ret	pop return address from stack and jump there
movl	move 4 bytes between immediate values, registers and memory
movzbl	move 1 byte into the low-order byte of a long word, filling the other 3 bytes with 0s.
movsbl	move 1 byte into the low-order byte of a long word, filling the other 3 bytes by sign-extending the low-order byte that was moved
addl	add first operand to second with result stored in second
subl	subtract first operand from second with result stored in second
imull	multiply first operand and second with result stored in second
sall	left shift second operand by count given in first operand
sarl	right shift second operand by count given in first operand
andl	logical bitwise AND of first and second operands, result stored in second
xorl	logical bitwise XOR of first and second operands, result stored in second
jmp	jump to address
je	conditional jump to address if zero flag set
jne	conditional jump to address if zero flag is not set
cmpl	subtract first operand from second and set flags
testl	logical and of first and second operands to set flags
nop	"no operation" – does nothing (sometimes generated by compilers to pad or align generated code)

x86-64 Parameter Registers

First 6 arguments are passed in registers %rdi, %rsi, %rdx, %rcx, %r8, and %r9 in that order.