Full Name:______________________________________

Student ID:______________________________________

UW CSE 351, Winter 2013

Midterm Exam

February 15, 2013

Instructions:

- Make sure that your exam is not missing any of the 9 pages, then write your full name and UW student ID on the front.
- Read over the entire exam before starting to work on the problems! The last page is a reference page that you may tear off for use during the exam; it does not have to be turned in.
- Write your answers in the space provided below each problem. If you make a mess, clearly indicate your final answer. Be sure to answer all parts of all questions.
- Don't spend too much time on a problem if there are other easy problems that you haven't solved yet. There are 50 total points and 50 minutes to take the exam, so try to answer questions at a rate of one point per minute.
- **No books, notes, or electronic devices may be used during the exam.** You may not communicate with other students during the exam, but please ask the instructor / TAs if you need clarification for some problem.

Problem 1	(8 points):	
Problem 2	(10 points):	
Problem 3	(18 points):	
Problem 4	(8 points):	
Problem 5	(6 points):	
TOTAL	(50 points):	

Problem 1. (8 points):

Consider an 8-bit machine that uses two's complement arithmetic for signed integers. What is the maximum signed integer value, *in decimal*, that can be represented with 8 bits?
With an n-bit two's complement representation, we can represent values from -2^{n-1} ***to*** $2^{n-1}-1$***, so the maximum signed value is*** $2^7 - 1 = 127$***. You could also calculate this if you know that the maximum binary value is*** 01111111***:*** $1+2+4+8+16+32+64=127$***.***

What is the minimum signed integer value, *in decimal*, that can be represented with 8 bits?
$\boldsymbol{-128}$***; again, if you know that the minimum binary value has the most-significant (negative-weight) bit set and no other (positive-weight) bits set, you could calculate this from*** 10000000 **=** -128***.***

What is the result, *in decimal*, if we add together the following two signed integers (represented in binary): $00010110 + 11111100$?
To solve this problem, you can either add the two binary numbers together using standard carry-addition (ignoring the final carry-out) and then convert to decimal, or you can convert both numbers to decimal first and then add. The result is $22 + -4 = 18$***.***

When we add together $50 + 100$ on this machine, we get the result -106. What phenomenon has occured here? (one word)
Overflow.

Problem 2. (10 points):

Consider the following assembly code for a C `for` loop:

```
loop:
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %ecx
        movl    12(%ebp), %edx
        movl    $0, %eax
        cmpl    %edx, %ecx
        jle     .L3
.L6:
        subl    $1, %ecx
        addl    $1, %edx
        addl    $1, %eax
        cmpl    %edx, %ecx
        jg      .L6
.L3:
        addl    $1, %eax
        popl    %ebp
        ret
```

Based on the assembly code above, fill in the blanks below in its corresponding C source code. (Note: you may only use the symbolic variables `x`, `y`, and `result` in your expressions below — *do not use register names.*)

```
int loop(int x, int y)
{
   int result;

   for (________________; ________________; result++) {

       ________________;

       ________________;
   }

   ________________;

   return result;
}
```

Solution: `int result; for (result = 0; x > y; result++) { x--; y++; } result++; return result;`

Problem 3. (18 points):

This page contains code for Problem 3. If you wish, you may carefully detach this page from the exam (make sure all other pages are still secure!) to avoid flipping back and forth; this page does not need to be turned in.

Consider the following C code:

```
int sum_plus_seven(int *xp, int *yp)
{
        int num = 7;
        int x = *xp;
        int y = *yp;
        return x + y + num;
}

int call_sum()
{
        int a = 3;
        int b = 5;
        int c = sum_plus_seven(&a, &b);
        return c;
}
```

These procedures have the following disassembled form on an IA32 machine:

```
080483fc <sum_plus_seven>:
 80483fc:       55                      push   %ebp
 80483fd:       89 e5                   mov    %esp,%ebp
 80483ff:       8b 45 08                mov    0x8(%ebp),%eax
 8048402:       8b 00                   mov    (%eax),%eax
 8048404:       83 c0 07                add    $0x7,%eax
 8048407:       8b 55 0c                mov    0xc(%ebp),%edx
 804840a:       03 02                   add    (%edx),%eax
 804840c:       5d                      pop    %ebp
 804840d:       c3                      ret

0804840e <call_sum>:
 804840e:       55                      push   %ebp
 804840f:       89 e5                   mov    %esp,%ebp
 8048411:       83 ec 18                sub    $0x18,%esp
 8048414:       c7 45 fc 03 00 00 00    movl   $0x3,-0x4(%ebp)
 804841b:       c7 45 f8 05 00 00 00    movl   $0x5,-0x8(%ebp)
 8048422:       8d 45 f8                lea    -0x8(%ebp),%eax
 8048425:       89 44 24 04             mov    %eax,0x4(%esp)
 8048429:       8d 45 fc                lea    -0x4(%ebp),%eax
 804842c:       89 04 24                mov    %eax,(%esp)
 804842f:       e8 c8 ff ff ff          call   80483fc <sum_plus_seven>
 8048434:       c9                      leave
 8048435:       c3                      ret
```

Problem 3. (18 points):

A. Suppose our program executes `call_sum()`. Assume that after executing the `mov` instruction at address `0x804840f`, both `%esp` and `%ebp` contain the address `0xffffffec`. Simulate the execution of the program up to the point where *the* `mov` *instruction at address* `0x80483fd` *has just completed*. Fill in the diagram below with a name or description for each item that is placed on the stack, and the value of that item (you do not have to fill in values for the locations that are already filled with dashes). If a location on the stack is not used, write "unused" in the description for that address.

Address in memory	**Name / description of item on stack**	**Value**
0xffffffec	%ebp saved by call_sum	-----------------------
0xffffffe8	a	3
0xffffffe4	b	5
0xffffffe0	unused	-----------------------
0xffffffdc	unused	-----------------------
0xffffffd8	*b	0xffffffe4
0xffffffd4	*a	0xffffffe8
0xffffffd0	Return address	0x8048434
0xffffffcc	%ebp saved by sum_plus_seven	0xffffffec

(This problem continues on the next page!)

B. Continue simulating the execution of the program until *the* `pop` *instruction at address* `0x804840c` *has just completed*. What are the values in registers `%esp` and `%ebp` at this point? (Feel free to draw in the margins on the previous page, outside of the diagram, but write your answers here.)

After the `pop`***, the stack is in exactly the state that it was in after the*** `call` ***instruction at*** `0x804842f`***. The old*** `%ebp` ***that was saved at the beginning of*** `sum_plus_seven` ***was just popped off the stack and stored in*** `%ebp`***, so*** `%ebp = 0xffffffec`***. The return address is now on the top of the stack, and it will be popped off by the next instruction to be executed, the*** `ret` ***at*** `0x804840d`***. The stack pointer points at the return address on the stack:*** `%esp = 0xffffffd0`***.***

C. Suppose that we compiled this C code on an x86-64 machine rather than on an IA32 machine. Describe ***one*** way that you would expect the generated assembly code to change.

There are many possible answers, including: we might see the use of the additional registers `%r8 - %r15`***; registers would be used to pass arguments rather than the stack; a stack frame probably would not be allocated at all for*** `sum_plus_seven`***; if a stack frame was used, offsets would be taken from*** `%rsp` ***and*** `%rbp` ***would be available for general-purpose use; etc.***

Problem 4. (8 points):

Consider the following C struct declaration on an IA32 Linux system:

```
struct node {
    short p[3];
    int r;
    struct node *next;
}
```

Recall that in C the size of a `short` is two bytes.

A. Using the template below (allowing a maximum of 24 bytes), diagram how the compiler will lay out the members of a `struct node` in memory, using the IA32 Linux alignment rules. Mark off and label the bytes for each individual element (arrays may be labeled as a single element). **Shade or cross-hatch bytes that are allocated, but are not used (to satisfy alignment). Clearly indicate the right-hand boundary of the data structure.**

```
struct node:

  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15|16 17 18 19 20 21 22 23
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--|--+--+--+--+--+--+--+--+
|   short p[3]    |#####|   int r   |   *next   |                       |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--|--+--+--+--+--+--+--+--+
```

B. When a struct is placed in memory, its initial address (the address of its first byte) will be a multiple of K. What is the value of K for a `struct node` on an IA32 Linux system?

K is set to the maximum alignment requirement of any of the members in the struct. The `short` ***elements are aligned to multiples of 2 bytes, the*** `int` ***is aligned to a multiple of 4, and the pointer is aligned to a multiple of 4, so K = 4.***

C. Can we reduce the number of bytes required for a `struct node` by defining its members in a different order? Why or why not?

No, because the total size of the struct must also be a multiple of K; even if we rearrange the members of the struct, it will still occupy 16 bytes total, with two bytes of padding somewhere.

D. When we allocate a nested (e.g. two-dimensional) array in C, is it laid out in memory with the rows in contiguous bytes, or with the columns in contiguous bytes? (Note: this question is unrelated to `struct node`.)

Rows.

Problem 5. (6 points):

Match each of the assembly procedures on the left with the equivalent C function on the right. ***You must show some work*** (e.g. write a note or two on the assembly functions) ***in order to receive credit!***
Note that the `shr` instruction performs a ***logical*** right-shift. `int`s are four bytes in size, as usual.

```
foo1:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %eax
    movl    (%eax), %eax
    addl    %eax, %eax
    popl    %ebp
    ret

foo2:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    %edx, %eax
    sall    $4, %eax
    subl    %edx, %eax
    popl    %ebp
    ret

foo3:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %eax
    shrl    $31, %eax
    popl    %ebp
    ret
```

```
int choice1(int x)
{
        return (x < 0);
}

int choice2(int x)
{
        return (x << 31) & 1;
}

int choice3(int x)
{
        return 15 * x;
}

int choice4(int x)
{
        return (x ^ 31) & 1;
}

int choice5(int *x)
{
        return *x + *x;
}

int choice6(int *x)
{
        return *x * *x;
}

int choice7(int *x)
{
        return (*x >> 31);
}
```

Fill in your answers here:
foo1 corresponds to choice **5**.
foo2 corresponds to choice **3**.
foo3 corresponds to choice **1**.

`choice3()`: to multiply by 15, first shift x left 4 (equivalent to multiplying by $2^4 = 16$), then subtract x from that result.
`choice1()`: to check if x < 0, this code returns 1 if the most-significant bit in x is 1, otherwise it returns 0.

References

Powers of 2:

$2^0 = 1$	
$2^1 = 2$	$2^{-1} = 0.5$
$2^2 = 4$	$2^{-2} = 0.25$
$2^3 = 8$	$2^{-3} = 0.125$
$2^4 = 16$	$2^{-4} = 0.0625$
$2^5 = 32$	$2^{-5} = 0.03125$
$2^6 = 64$	$2^{-6} = 0.015625$
$2^7 = 128$	$2^{-7} = 0.0078125$
$2^8 = 256$	$2^{-8} = 0.00390625$
$2^9 = 512$	$2^{-9} = 0.001953125$
$2^{10} = 1024$	$2^{-10} = 0.0009765625$

x86 assembly 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
leave	`mov %ebp, %esp`, then `pop %ebp`
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 1st operand to 2nd with result stored in 2nd
sub	subtract 1st operand from 2nd with result stored in 2nd
and	bit-wise AND of two operands with result stored in 2nd
or	bit-wise OR of two operands with result stored in 2nd
sal, shl	left shift
sar	arithmetic right shift
shr	logical right shift
cmp	subtract 1st operand from 2nd and set condition flags
jmp	jump to address
jg	conditional jump to address if signed comparison is greater-than
jge	conditional jump to address if signed comparison is greater-than-or-equal
jl	conditional jump to address if signed comparison is less-than
jle	conditional jump to address if signed comparison is less-than-or-equal