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.

<table>
<thead>
<tr>
<th>Problem 1</th>
<th>(8 points):</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem 2</td>
<td>(10 points):</td>
</tr>
<tr>
<td>Problem 3</td>
<td>(18 points):</td>
</tr>
<tr>
<td>Problem 4</td>
<td>(8 points):</td>
</tr>
<tr>
<td>Problem 5</td>
<td>(6 points):</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td>(50 points):</td>
</tr>
</tbody>
</table>
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?

What is the minimum signed integer value, *in decimal*, that can be represented with 8 bits?

What is the result, *in decimal*, if we add together the following two signed integers (represented in binary):
00010110 + 11111100 ?

When we add together 50 + 100 on this machine, we get the result −106. What phenomenon has occurred here? (one word)
Problem 2. (10 points):
Consider the following assembly code for a C for loop:

```assembly
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;
}
```
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:

```c
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
  8048410: 89 e5 mov %esp,%ebp
  8048412: 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 0xffffffffec. 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.

<table>
<thead>
<tr>
<th>Address in memory</th>
<th>Name / description of item on stack</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xffffffffec</td>
<td>%ebp saved by call_sum</td>
<td>--------------------</td>
</tr>
<tr>
<td>0xfffffffff8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xfffffffff4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xfffffffff0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xffffffffdc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xffffffffd8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xffffffffd4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xffffffffd0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xffffffffcc</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(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.)

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.
Problem 4. (8 points):
Consider the following C struct declaration on an IA32 Linux system:

```c
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
  +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+-- +--+--+--+--+--+--+--+
  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
  +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+-- +--+--+--+--+--+--+--+
```

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?

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?

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.)
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. ints 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 (%eax), %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

Fill in your answers here:
foo1 corresponds to choice ________
foo2 corresponds to choice ________
foo3 corresponds to choice ________

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);
}
References

Powers of 2:

<table>
<thead>
<tr>
<th>$2^0$</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^1$</td>
<td>2</td>
</tr>
<tr>
<td>$2^2$</td>
<td>4</td>
</tr>
<tr>
<td>$2^3$</td>
<td>8</td>
</tr>
<tr>
<td>$2^4$</td>
<td>16</td>
</tr>
<tr>
<td>$2^5$</td>
<td>32</td>
</tr>
<tr>
<td>$2^6$</td>
<td>64</td>
</tr>
<tr>
<td>$2^7$</td>
<td>128</td>
</tr>
<tr>
<td>$2^8$</td>
<td>256</td>
</tr>
<tr>
<td>$2^9$</td>
<td>512</td>
</tr>
<tr>
<td>$2^{10}$</td>
<td>1024</td>
</tr>
</tbody>
</table>

$x86$ 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