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.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Max Score</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>TOTAL</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>
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 the others differ. There is no need to convert these to decimal values.
2. Assembly Code (20 points)

A function ‘flip’ has the following overall structure:

```c
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
  1     movl 8(%ebp), %ebx
  2     movl (%ebx), %esi
  3     movl $0, %eax
  4     movl $0, %ecx
  5     .L13:
  6     leal (%eax, %eax), %edx
  7     movl %esi, %eax
  8     andl $1, %eax
  9     orl %edx, %eax
 10     shrl %esi
 11     add $1, %ecx
 12     cmpl $32, %ecx
 13     jne .L13
 14     ret
```

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: 

Test: 

Update: 

Body: 

```c
int flip (*unsigned x) {
    int num=*x;
    int val=0;
    int i;
    for ( ___________ ; ___________ ; ___________ ) {
        ______________
        ______________
        ______________
    }
    return val;
}
```

B (5 pts). Describe what this function computes in one English sentence (or at most two).
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>
400555:    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?
B (5 pts). Why is %rbx pushed onto the stack initially and then popped at the end?

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

D (5 pts). What does the byte ‘ec’ most likely correspond to in the instruction at 0x40055a?
E (15 pts). Write out C code that would assemble into the routine above.

```
unsigned int mystery(unsigned int n) {
}
```

F (5 pts). What does this function do?
4. Stack Discipline (25 points)

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

<table>
<thead>
<tr>
<th>Line ref number</th>
<th>Address in memory</th>
<th>Value in memory</th>
<th>Check if ret addr</th>
<th>Check if arg or local var</th>
<th>Check if saved ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>0xfffffffffc</td>
<td>0x000000001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>0xfffffffff8</td>
<td>0x000000005</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>0xfffffffff4</td>
<td>0xfffffffffc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>0xfffffffff0</td>
<td>0x0040800a0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>0xfffffffffc</td>
<td>0xfffffffffc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>0xfffffffff8</td>
<td>0x000000005</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>0xfffffffff4</td>
<td>0x0040801e</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>0xffffffffe0</td>
<td>0xfffffffffc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>0xffffffffdc</td>
<td>0x000000004</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>0xffffffffd8</td>
<td>0x0040801e</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>0xffffffffd4</td>
<td>0xfffffffffe0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0xffffffffd0</td>
<td>0x000000003</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0xffffffffcc</td>
<td>0x0040801e</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>0xffffffffc8</td>
<td>0xfffffffffd4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0xffffffffc4</td>
<td>0x000000002</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0xffffffffc0</td>
<td>0x0040801e</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0xffffffffbc</td>
<td>0xffffffffc8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0xffffffffb8</td>
<td>0x00800000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0xffffffffb4</td>
<td>0x008000d0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0xffffffffb0</td>
<td>0x000000001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0xffffffffac</td>
<td>0x000000001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0xffffffffa8</td>
<td>0x00408053</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0xffffffffa4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0xffffffffa0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Furthermore, you know that your code is in memory in locations from 0x00400000 to 0x005ffffff and that your dynamic data heap is in locations 0x00800000 to 0x009ffffff.

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?
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.

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?
REFERENCES

Powers of 2:

<table>
<thead>
<tr>
<th>$2^n$</th>
<th>$2^{-n}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2⁰ = 1</td>
<td>$2^{-1} = .5$</td>
</tr>
<tr>
<td>2¹ = 2</td>
<td>$2^{-2} = .25$</td>
</tr>
<tr>
<td>2² = 4</td>
<td>$2^{-3} = .125$</td>
</tr>
<tr>
<td>2³ = 8</td>
<td>$2^{-4} = .0625$</td>
</tr>
<tr>
<td>2⁴ = 16</td>
<td>$2^{-5} = .03125$</td>
</tr>
<tr>
<td>2⁵ = 32</td>
<td>$2^{-6} = .015625$</td>
</tr>
<tr>
<td>2⁶ = 64</td>
<td>$2^{-7} = .0078125$</td>
</tr>
<tr>
<td>2⁷ = 128</td>
<td>$2^{-8} = .00390625$</td>
</tr>
<tr>
<td>2⁸ = 256</td>
<td>$2^{-9} = .001953125$</td>
</tr>
<tr>
<td>2⁹ = 512</td>
<td>$2^{-10} = .0009765625$</td>
</tr>
</tbody>
</table>

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 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
- **shr**: shift data by 1 bit to the right
- **jmp**: jump to address
- **cmp**: subtract 1ˢᵗ operand from 2ⁿᵈ and set flags
- **jne**: conditional jump to address if zero flag is not set