Please read through the entire examination first!

- You have 50 minutes for this exam. Don’t spend too much time on any one problem!
- The last page is a reference sheet. Feel free to detach it from the rest of the exam.
- The exam is CLOSED book and CLOSED notes (no summary sheets, no calculators, no mobile phones).

There are 5 problems for a total of 50 points. The point value of each problem is indicated in the table below. Write your answer neatly in the spaces provided.

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.

Good Luck!

Your Name:____Sample Solution_____

UWNet ID:_______woof2017_________

<table>
<thead>
<tr>
<th>Name of person to your left</th>
<th>Name of person to your right</th>
</tr>
</thead>
<tbody>
<tr>
<td>__________________________</td>
<td>__________________________</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem</th>
<th>Topic</th>
<th>Max Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Integers &amp; Floats</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>Hardware to Software</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>Structs &amp; Arrays</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>Pointers &amp; Memory</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>Stack Discipline</td>
<td>15</td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td>50</td>
</tr>
</tbody>
</table>
1. Integers and Floats (7 points)

a. In the card game Schnapsen, 5 cards are used (Ace, Ten, King, Queen, and Jack) from 4 suits, so 20 cards in total. What are the minimum number of bits needed to represent a single card in a Schnapsen deck?

\[
\text{5 bits}
\]

We need 2 bits to represent 4 suits, and 3 bits to represent 5 ranks. So 5 bits in total.

b. How many negative numbers can we represent if given 7 bits and using two’s complement?

\[
2^6
\]

Using 7 bits, the MSB has to be 1 for negative numbers. So there are \(2^6\) negative numbers in total.

Consider the following pseudocode (we’ve written out the bits instead of listing hex digits):

```plaintext
int a = 0b0100 0000 0000 0000 0000 0011 1100 0000
int b = (int)(float)a
int m = 0b0100 0000 0000 0000 0000 0011 0000 0000
int n = (int)(float)m
```

c. Circle one:  True or  False:

\[
a == b
\]

The right-most 1 will be truncated (cannot fit in Mantissa)

d. Circle one:  True  or  False:

\[
m == n
\]

No precision will be lost

e. How many IEEE single precision floating point numbers are in the range \([4, 6)\) (That is, how many floating point numbers are there where \(4 \leq x < 6\)?)

\[
2^{22}
\]

4 in binary is \(1.0 \cdot 2^2\).

6 in binary is \(1.1 \cdot 2^2\).

So in Mantissa the right-most 22 bits can be either 0 or 1. Therefore, there are \(2^{22}\) bits in range \([4, 6)\)
2. Hardware to Software (7 points)

a) If the word size of a machine is \( m \) bits, which of the following is typically true:

- TRUE / FALSE: \( m \) bits is the size of an integer

- TRUE / FALSE: \( m \) bits is the width of a register

b) If the size of a pointer on a machine is 32 bits, the size of the address space is how many bytes?

\( 2^{32} \)

c) ISA stands for: ___Instruction Set Architecture____

d) TRUE / FALSE: The number of registers available is part of the ISA.

e) Part of the object file that keeps track of symbols/labels needed by this source file is the:

Relocation Table

f) The tool used to combine one or more .o files into an executable is called the:

___Linker or ld_____.

(Hint: the answer is not “gcc”, we want the general or specific name of tool that does this particular step.)
3. Structs and Arrays (7 points)
You are given the following C program run on a 64-bit x86-64 (little endian) processor:

```c
struct diddle {
    int x;
    struct diddle *y;
    int z;
    char c[3];
};

int main(void) {
    struct diddle d;
    d.x = 0xdeadbeef;
    d.y = &d;
    d.z = d.x >> 16;
    d.c[0] = 0x12;
    d.c[1] = 0x34;
    d.c[2] = 0x56;
    return 0;
}
```

a. Below is a view of the stack. Suppose we have just reached the return statement and assume `d` is placed at address `0x7fffffffac0`. Please fill in the bytes on the stack in hex (you may omit the 0x prefix).

```
<table>
<thead>
<tr>
<th>Address</th>
<th>+0</th>
<th>+1</th>
<th>+2</th>
<th>+3</th>
<th>+4</th>
<th>+5</th>
<th>+6</th>
<th>+7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7fffffffac0</td>
<td>ef</td>
<td>be</td>
<td>ad</td>
<td>de</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7fffffffac8</td>
<td>c0</td>
<td>fa</td>
<td>ff</td>
<td>ff</td>
<td>ff</td>
<td>07</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>0x7ffffffffad0</td>
<td>ad</td>
<td>de</td>
<td>ff</td>
<td>ff</td>
<td>12</td>
<td>34</td>
<td>56</td>
<td></td>
</tr>
<tr>
<td>0x7ffffffffad8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

c. What is the total size of this struct in bytes?

**24 Bytes**

d. Is there a reordering of the fields in diddle that would reduce its total size? If so, what is it?

**No. The struct has to end at an address of multiple of 8.**
4. **Pointers, Memory & Registers (14 points)**

Assuming a 64-bit x86-64 machine (little endian), you are given the following variables and initial state of memory (values in hex) shown below:

<table>
<thead>
<tr>
<th>Address</th>
<th>+0</th>
<th>+1</th>
<th>+2</th>
<th>+3</th>
<th>+4</th>
<th>+5</th>
<th>+6</th>
<th>+7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>AB</td>
<td>EE</td>
<td>1E</td>
<td>AC</td>
<td>D5</td>
<td>8E</td>
<td>10</td>
<td>E7</td>
</tr>
<tr>
<td>0x08</td>
<td>F7</td>
<td>84</td>
<td>32</td>
<td>2D</td>
<td>A5</td>
<td>F2</td>
<td>3A</td>
<td>CA</td>
</tr>
<tr>
<td>0x10</td>
<td>83</td>
<td>14</td>
<td>53</td>
<td>B9</td>
<td>70</td>
<td>03</td>
<td>F4</td>
<td>31</td>
</tr>
<tr>
<td>0x18</td>
<td>01</td>
<td>20</td>
<td>FE</td>
<td>34</td>
<td>46</td>
<td>E4</td>
<td>FC</td>
<td>52</td>
</tr>
<tr>
<td>0x20</td>
<td>4C</td>
<td>A8</td>
<td>B5</td>
<td>C3</td>
<td>D0</td>
<td>ED</td>
<td>53</td>
<td>17</td>
</tr>
</tbody>
</table>

\[
\text{int* } \text{ip} = 0x00; \\
\text{short* } \text{sp} = 0x20; \\
\text{long* } \text{yp} = 0x10; \\
\]

a) Fill in the type and value for each of the following C expressions. If a value cannot be determined from the given information answer UNKNOWN.

<table>
<thead>
<tr>
<th>Expression (in C)</th>
<th>Type</th>
<th>Value (in hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{yp + 2})</td>
<td>long*</td>
<td>0x20</td>
</tr>
<tr>
<td>(* (\text{sp} - 1))</td>
<td>short</td>
<td>0x52FC</td>
</tr>
<tr>
<td>(\text{ip[5]})</td>
<td>int</td>
<td>0x31F40370</td>
</tr>
<tr>
<td>&amp;ip</td>
<td>int**</td>
<td>UNKNOWN</td>
</tr>
</tbody>
</table>

b) Assuming that all registers start with the value 0, except \$rax which is set to 0x4, fill in the values (in hex) stored in each register after the following x86 instructions are executed. Remember to give enough hex digits to fill up the width of the register name listed.

```
\text{movl 2(\%rax), \%ebx}
\text{leal (\%rax,\%rax,2), \%ecx}
\text{movslb 4(\%rax), \%edi}
\text{subw (,\%rax,2), \%si}
```
5. Stack Discipline (15 points)

Examine the following recursive function:

```c
long sunny(long a, long *b) {
    long temp;
    if (a < 1) {
        return *b - 8;
    } else {
        temp = a - 1;
        return temp + sunny(temp - 2, &temp);
    }
}
```

Here is the x86_64 assembly for the same function:

```
0000000000400536 <sunny>:
400536:       test   %rdi,%rdi
400539:       jg     400543 <sunny+0xd>
40053b:       mov    (%rsi),%rax
40053e:       sub    $0x8,%rax
400542:       retq
400543:       push   %rbx
400544:       sub    $0x10,%rsp
400548:       lea    -0x1(%rdi),%rbx
40054c:       mov    %rbx,0x8(%rsp)
400551:       sub    $0x3,%rdi
400555:       lea    0x8(%rsp),%rsi
40055a:       callq  400536 <sunny>
40055f:       add    %rbx,%rax
400562:       add    $0x10,%rsp
400566:       pop    %rbx
400567:       retq
```

We call `sunny` from `main()`, with registers `%rsi = 0x7ff...ffad8` and `%rdi = 6`. The value stored at address `0x7ff...ffad8` is the long value 32 (0x20). We set a breakpoint at “return *b - 8” (i.e. we are just about to return from `sunny()` without making another recursive call). We have executed the `sub` instruction at `40053e` but have not yet executed the `retq`.

Fill in the register values on the next page and draw what the stack will look like when the program hits that breakpoint. Give both a description of the item stored at that location and the value stored at that location. If a location on the stack is not used, write “unused” in the Description for that address and put “-----” for its Value. You may list the Values in hex or decimal. Unless preceded by `0x` we will assume decimal. It is fine to use `f...f` for sequences of `f`’s as shown above for `%rsi`. Add more rows to the table as needed. Also, fill in the box on the next page to include the value this call to `sunny` will finally return to `main`. 
### Register Table

<table>
<thead>
<tr>
<th>Register</th>
<th>Original Value</th>
<th>Value at Breakpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>rsp</td>
<td>0x7ffe...ffad0</td>
<td>0x7ffe...ffa90</td>
</tr>
<tr>
<td>rdi</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>rsi</td>
<td>0x7ffe...ffad8</td>
<td>0x7ffe...ffaa0</td>
</tr>
<tr>
<td>rbx</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>rax</td>
<td>5</td>
<td>-6</td>
</tr>
</tbody>
</table>

### Memory Address Table

<table>
<thead>
<tr>
<th>Memory address on stack</th>
<th>Name/description of item</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7fffffffffffffad8</td>
<td>Local var in main</td>
<td>0x20</td>
</tr>
<tr>
<td>0x7fffffffffffffad0</td>
<td>Return address back to main</td>
<td>0x400827</td>
</tr>
<tr>
<td>0x7fffffffffffffffac8</td>
<td>Saved %rbx</td>
<td>4</td>
</tr>
<tr>
<td>0x7fffffffffffffffac0</td>
<td>temp</td>
<td>5</td>
</tr>
<tr>
<td>0x7fffffffffffffffac0</td>
<td>Unused</td>
<td>-----------</td>
</tr>
<tr>
<td>0x7fffffffffffffffab8</td>
<td>Return address to sunny</td>
<td>0x40055f</td>
</tr>
<tr>
<td>0x7fffffffffffffffaa8</td>
<td>Saved %rbx</td>
<td>5</td>
</tr>
<tr>
<td>0x7fffffffffffffffaa0</td>
<td>temp</td>
<td>2</td>
</tr>
<tr>
<td>0x7fffffffffffffff98</td>
<td>Unused</td>
<td>-----------</td>
</tr>
<tr>
<td>0x7fffffffffffffff90</td>
<td>Return address to sunny</td>
<td>0x40055f</td>
</tr>
<tr>
<td>0x7fffffffffffffff88</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7fffffffffffffff80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7fffffffffffffff78</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7fffffffffffffff70</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7fffffffffffffff68</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7fffffffffffffff60</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

DON’T FORGET: What value is **finally** returned to **main** by this call?

1