Please read through the entire examination first!
- You have 60 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 53 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:_________________________

<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>13</td>
</tr>
<tr>
<td>2</td>
<td>Hardware to Software</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>C &amp; Assembly</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>Stack Discipline</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>Pointers &amp; Memory</td>
<td>9</td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td>53</td>
</tr>
</tbody>
</table>
1. Integers and Floats (13 points total)
We define two new types as follows:

*Ten* _ints_ are 10-bit signed two’s complement integers.

*Ten* _floats_ are 10-bit floating point numbers with 4 bits for the exponent, 5 bits for the fraction, and 1 bit for the sign. *Ten* _floats_ are similar to IEEE floating point as far as layout of sign, exponent and fraction and represent special values (e.g. 0, pos and neg infinity, NAN) similar to how they are represented in 32 bit IEEE floating point.

a) (2 pts) What is the most negative number we can represent with *Ten* _ints_?

Give the bit pattern in binary: 1000000000

Give the value in decimal: \(-(2^9) = -512\)

b) (4 pts) Convert the following *Ten* _floats_ bit pattern into decimal.

Bit pattern in binary: 0101111000

Give the bias for *Ten* _float_ 7

\(2^{4+1} - 1 = 8 - 1 = 7\)

Give the value in decimal: 28

Exponent = 11 - 7 = 4
Value = \(0b1.11 \times 2^4 = 0b11100 = 16 + 8 + 4 = 28\)

c) (2 pts) What is the result in binary if you add 1 to the maximum positive *Ten* _ints_ number?

0 1111 1111 + 1 = (the most negative number)

10000 0000

In a sentence, describe what is happening in this case:

This is overflow: adding a positive number to a positive number, resulting in a negative number.
d) (5 pts) Assuming rules similar to those for conversions between IEEE floats and ints and addition in C, indicate whether the following statements are True or False.

- **TRUE / FALSE**: Given a `Ten_float` that is **negative**, it is possible to lose precision when converting from a `Ten_float` to a `Ten_int`.

- **TRUE / FALSE**: The largest positive number representable as a `Ten_int` **<** the largest non-infinite positive number representable as a `Ten_float`.

- **TRUE / FALSE**: Given a `Ten_int` that is **positive**, it is possible to lose precision when converting it to a `Ten_float`.

- **TRUE / FALSE**: Adding a negative `Ten_float` to a positive `Ten_float` will **never** result in a loss of precision.

- **TRUE / FALSE**: Adding a negative `Ten_int` to a positive `Ten_int` will **never** result in overflow.
2. Hardware to Software (8 points total)

a) (2 pts) Give one disadvantage of sign magnitude representation of ints compared to two’s complement representation of ints.

- Two representations of 0 (bad for checking equality)
- Arithmetic is cumbersome. Example: 4-3 != 4+(-3)

b) (2 pts) What does having a standardized calling convention enable that would not be possible otherwise?

Code generated by one compiler can be called by code generated by another compiler. This makes it possible to call library code.

c) (4 pts) Your friend proposes that the next generation of x86 machines only allow you to refer to the full 64-bit versions of registers (we could no longer refer to only a portion of a register and instructions would all have the suffix q). For example, instructions like `movb (%rax), $dil` would instead have to be: `movq (%rax), $rdi`

Give one drawback and one benefit of this approach.

**Drawbacks:**
- All datatypes are now the same size, overall memory use by your data is likely to be more
- X86 programs already written may no longer work anymore, no backwards compatibility

**Benefits:**
- No need to worry about alignment of different type sizes (since there is in effect only one size)
- Addresses could refer to 8 bytes instead of 1 byte, since there is no need or ability to access smaller chunks of memory, so your address space could become 8x larger without changing the bit width of an address
- Machine Encoding of instructions may be smaller – no need to specify suffix or as many possible registers
3. C and Assembly (11 points total)
You are given the following x86-64 assembly function:

```
mystery:
    movl    $0, %edx
    movl    $0, %eax
.L3:
    cmpl    %esi, %edx
    jge     .L1
    movslq  %edx, %rcx
    addl    (%rdi,%rcx,4), %eax
    addl    $1, %edx
    jmp     .L3
.L1:
    rep ret
```

a) (1 pt) What variable type would `%rdi` be in the corresponding C program?
   
   int*

b) (1 pt) What variable type would `%rsi` be in the corresponding C program?
   
   int

c) (7 pts) Fill in the missing C code that is equivalent to the x86-64 assembly above:

```c
_____ int_____ mystery( (answer to a) rdi, (answer to b) rsi) {
    ___ int _____ eax = ___ 0 ; ________

    for (int edx = 0; edx < rsi; edx++) {
        eax += rdi[edx];
    }

    return eax;
}
```

d) (2 pts) In 1 sentence, describe what this function is doing?

   Summing the first rsi elements of the int array starting at rdi
Examine the following recursive function:

```c
long husky(long *x, long y) {
    long woof = 0;
    if (y < 3) {
        return *x * 3;
    } else {
        woof = y + 1;
        return woof + husky(&woof, y - 1);
    }
}
```

Here is the x86_64 assembly for the same function:

```
000000000040057a <husky>:
   40057a:  cmp   $0x2,%rsi
   40057e:  jg    400588 <husky+0xe>
   400580:  mov   (%rdi),%rax
   400583:  lea   (%rax,%rax,2),%rax
   400587:  retq
   400588:  push  %rbx
   400589:  sub   $0x10,%rsp
   40058d:  lea   0x1(%rsi),%rbx
   400591:  mov   %rbx,0x8(%rsp)
   400596:  sub   $0x1,%rsi
   400599:  lea   0x8(%rsp),%rdi
   40059f:  callq 40057a <husky>
   4005af:  add   %rbx,%rax
   4005a7:  add   $0x10,%rsp
   4005ab:  pop   %rbx
   4005ac:  retq
```

We call `husky` from `main()`, with registers `%rdi = 0x7ff...ffbd8` and `%rsi = 4`. The value stored at address `0x7ff...ffbd8` is the long value 16 (0x10). We set a breakpoint at “return *x * 3” (i.e. we are just about to return from `husky()` without making another recursive call). We have executed the `lea` instruction at 400583 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 for %rdi. Add more rows to the table as needed. Also, fill in the box on the next page to include the value this call to `husky` will finally return to `main`. 
<table>
<thead>
<tr>
<th>Register</th>
<th>Original Value</th>
<th>Value at Breakpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>rsp</td>
<td>0x7ff...ffbd0</td>
<td>0x7ffffffffffffffffb90</td>
</tr>
<tr>
<td>rdi</td>
<td>0x7ff...ffbd8</td>
<td>0x7ffffffffffffffffba0</td>
</tr>
<tr>
<td>rsi</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>rbx</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>rax</td>
<td>7</td>
<td>12</td>
</tr>
</tbody>
</table>

What value is **finally** returned to `main` by this call? **21**

<table>
<thead>
<tr>
<th>Memory address on stack</th>
<th>Name/description of item</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7ffffffffffffffffb8d8</td>
<td>Local var in main</td>
<td>0x10</td>
</tr>
<tr>
<td>0x7ffffffffffffffffb0</td>
<td>Return address back to <code>main</code></td>
<td>0x400975</td>
</tr>
<tr>
<td>0x7ffffffffffffffffbc8</td>
<td>Old rbx</td>
<td>9</td>
</tr>
<tr>
<td>0x7ffffffffffffffffbc0</td>
<td><code>woof</code></td>
<td>5</td>
</tr>
<tr>
<td>0x7ffffffffffffffffbb8</td>
<td><code>Unused</code></td>
<td><code>Unknown</code></td>
</tr>
<tr>
<td>0x7ffffffffffffffffbb0</td>
<td>Return address back to <code>husky</code></td>
<td>0x4005a4</td>
</tr>
<tr>
<td>0x7ffffffffffffffffba8</td>
<td><code>Old rbx</code></td>
<td>5</td>
</tr>
<tr>
<td>0x7ffffffffffffffffba0</td>
<td><code>woof</code></td>
<td>4</td>
</tr>
<tr>
<td>0x7ffffffffffffffffb98</td>
<td><code>Unused</code></td>
<td><code>Unknown</code></td>
</tr>
<tr>
<td>0x7ffffffffffffffffb90</td>
<td>Return address back to <code>husky</code></td>
<td>0x4005a4</td>
</tr>
<tr>
<td>0x7ffffffffffffffffb88</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7ffffffffffffffffb80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7ffffffffffffffffb78</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7ffffffffffffffffb70</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7ffffffffffffffffb68</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7ffffffffffffffffb60</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

DON’T FORGET
5. Pointers, Memory & Registers (9 points total)

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>42</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>

long* yp = 0x18;
int* ip = 0x10;
short* sp = 0x08;

a) (5 pts) 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>*(ip + 2)</td>
<td>int</td>
<td>0x34FE2001</td>
</tr>
<tr>
<td>sp + 3</td>
<td>short*</td>
<td>0x0E</td>
</tr>
<tr>
<td>yp[-1]</td>
<td>long</td>
<td>0x31F40370B9531483</td>
</tr>
<tr>
<td>(*sp) + 1</td>
<td>short</td>
<td>0x8443</td>
</tr>
<tr>
<td>&amp;(ip[1])</td>
<td>int*</td>
<td>0x14</td>
</tr>
</tbody>
</table>

b) (4 pts) What are the values (in hex) stored in each register shown after the following x86 instructions are executed? *Remember to use the appropriate bit widths.*

movw (,%rax,4), %bx
movswq 4(,%rax,2), %rcx
leaq (%rsi,%rax,8), %rdi
addl 4(%rax,%rax), %esi

<table>
<thead>
<tr>
<th>Register</th>
<th>Value (in hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0x0000 0000 0000 0002</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x0000 0000 0000 0010</td>
</tr>
<tr>
<td>%bx</td>
<td>0x8442</td>
</tr>
<tr>
<td>%rcx</td>
<td>0xffffffffffffffff8442</td>
</tr>
<tr>
<td>%rdi</td>
<td>0x0000000000000020</td>
</tr>
<tr>
<td>%esi</td>
<td>0x2D328452</td>
</tr>
</tbody>
</table>