Do not turn the page until 10:50.

Instructions

- This exam contains 5 pages, including this cover page. Show scratch work for partial credit, but put your final answers in the boxes and blanks provided.
- The last page is a reference sheet. Please detach it from the rest of the exam.
- The exam is closed book (no laptops, tablets, wearable devices, or calculators). You are allowed one page (US letter, double-sided) of handwritten notes.
- Please silence and put away all cell phones and other mobile or noise-making devices. Remove all hats, headphones, and watches.
- You have 60 minutes to complete this exam.

Advice

- Read questions carefully before starting. Skip questions that are taking a long time.
- Read all questions first and start where you feel the most confident.
- Relax. You are here to learn.

<table>
<thead>
<tr>
<th>Question</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Possible Points</td>
<td>22</td>
<td>20</td>
<td>14</td>
<td>24</td>
<td>20</td>
<td>100</td>
</tr>
</tbody>
</table>
Question 1: Number Representation  [22 pts]

Consider the signed char \( x = 0b\ 1010\ 1000 \).

(A) What is the value of \( x \)? You may answer as the sum of powers of 2. [2 pt]

\[
-2^7 + 2^5 + 2^3 = -88
\]

(B) In theory (math), what is the difference in the values of \((\text{unsigned char})x\) and \(x\)? Your answer should be a decimal number. [2 pt]

\((\text{unsigned char})x\) has value \(2^7 + 2^5 + 2^3\).

\[
2^n = 2^8 = 256
\]

(C) In C, what is the value of \( char\ y = (\text{unsigned char})x - x \)? [2 pt]

Subtraction is done at the bit level, so the result is all zeros.
Also, the “result” of 256 can’t be represented by an 8-bit integer.

\( 0 \)

(D) Which of the following expressions will result in a positive (non-negative, non-zero) result? (Circle one) [4 pt]

\[
\begin{align*}
x << 4 & \ \ \ \ \ \ \ 0b10000000 \\
x \wedge 0x81 & \ \ \ \ \ \ 0b00101001 \\
x | \sim x & \ \ \ \ \ \ 0b11111111 \\
!(x >> 1) & \ \ \ \ \ \ 0
\end{align*}
\]

For the rest of this problem we are working with a new floating point datatype \((\text{flo})\) that follows the same conventions as IEEE 754 except using 8 bits split into the following fields:

<table>
<thead>
<tr>
<th>Sign (1)</th>
<th>Exponent (3)</th>
<th>Mantissa (4)</th>
</tr>
</thead>
</table>

(E) What is the value of the numeral from above \( 0b\ 1010\ 1000 \) in this representation? [4 pt]

\( S = 1, E = 0b010, M = 0b1000. \) bias=\(2^{3-1-1}=3.\)

\((-1)^1 \times 1.10002 \times 2^{-3} = -1.12 \times 2^1 = -0.112 = -(0.5 + 0.25)\)

\(-0.75\)

(F) What is the encoding of the most negative real number that we can represent (\(\infty\) is not a real number) in this floating point scheme (binary)? [4 pt]

Largest normalized number, but negative: \( 0b\ 1\ 110\ 1111 \)

(G) What will occur if we cast \(\text{flo}\ f = (\text{flo})x\) (i.e. try to represent the value stored in \(x\) as a \(\text{flo}\))? (Circle one) [4 pt]

Rounding Underflow Overflow None of these

\(-88 = -(64+16+8) = -(2^6+2^4+2^3) = -10110002 = -1.0110 \times 2^6. \) Mantissa fits, but max exponent is \(0b110 - \text{bias} = 6 - 3 = 3.\)
Question 2: Pointers & Memory  [20 pts]

For this problem we are using a 64-bit x86-64 machine (little endian). The current state of memory (values in hex) is shown below:

<table>
<thead>
<tr>
<th>Word Addr</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>AC</td>
<td>AB</td>
<td>03</td>
<td>01</td>
<td>BA</td>
<td>5E</td>
<td>BA</td>
<td>11</td>
</tr>
<tr>
<td>0x08</td>
<td>5E</td>
<td>00</td>
<td>68</td>
<td>0C</td>
<td>BE</td>
<td>A7</td>
<td>CE</td>
<td>FA</td>
</tr>
<tr>
<td>0x10</td>
<td>1D</td>
<td>B0</td>
<td>99</td>
<td>DE</td>
<td>AD</td>
<td>60</td>
<td>BB</td>
<td>40</td>
</tr>
<tr>
<td>0x18</td>
<td>14</td>
<td>1D</td>
<td>EC</td>
<td>AF</td>
<td>EE</td>
<td>FF</td>
<td>CO</td>
<td>70</td>
</tr>
<tr>
<td>0x20</td>
<td>BA</td>
<td>B0</td>
<td>41</td>
<td>20</td>
<td>80</td>
<td>AA</td>
<td>BE</td>
<td>EF</td>
</tr>
</tbody>
</table>

char* charP = 0x12
int* intP = 0x8
long* longP = 0x30

(A) Using the values shown above, complete the C code below to fulfill the behaviors described in the comments using pointer arithmetic.  [8 pt]

```
char v1 = *(charP + ___9___);  // set v1 = 0xAF
int* v2 = &intP[___3___];     // set v2 = 0x14
```

v1: Byte 0xAF is at address 0x1B. 0x1B - charP = 9.

v2: No dereferencing; just pointer arithmetic (scaled by sizeof(int) = 4). intP = 0x8. To get to 0x14, need to add 12 (3 by pointer arithmetic).

(B) What are the values (in hex) stored in each register shown after the following x86-64 instructions are executed? We are still using the state of memory shown above. *Remember to use the appropriate bit widths.*  [12 pt]

```
leab 7(%rsi),    %r9b
movl (%rdi,%rsi), %eax
movzbq -2(,%rsi,8), %r8
```

leab calculates address 7 + 0x3.

movl pulls four bytes starting at memory address 0x1C (remember little endian!).

movzbq instruction pulls the byte at memory address 0x3*8-2 = 22 = 0x16, which is 0xBB. Then zero-extend out to 8 bytes.
Question 3: Design Questions  [14 pts]

Answer the following questions in the boxes provided with a single sentence fragment. Please try to write as legibly as possible.

(A) Name the two issues with Sign and Magnitude that Two’s Complement fixed.  [4 pt]

- Two zeros.
- Bad arithmetic.
- Negatives incrementing in the wrong direction.

(B) Briefly describe an advantage of making addresses and registers both the width of a word.  [4 pt]

- Addresses are guaranteed to be able to fit/be stored in a register.
- Memory operands can be specified easily using registers.

(C) Briefly explain your answers to the following questions if we moved 1 bit from the mantissa field (now 22 bits) to the exponent field (now 9 bits) in floating point:  [6 pt]

Will the total number of representable floats (normalized + denormalized + special cases) change? Circle one: Yes  No

Explanation: We still have the same fixed width (32 bits), so we can still only represent $2^{22}$ floats. We’ve changed the numbers we represent, but not how many we represent.

Credit also given if Yes was argued by saying that there are more NaN’s than before, so fewer floats representable. This relies on counting all NaN’s as a single “special case,” which wasn’t the intention of the question, but is a reasonable interpretation.

The frequency of rounding will (circle one): Increase, Decrease, or Stay the same

Explanation: By shortening the M field, we lose precision. There are now more numbers than before that will have some value rounded off when encoded (any number whose mantissa has 1’s at least 23 digits apart instead of at least 24 digits apart).

The gaps between representable numbers doubled for the same exponent, so it’s easier to fall in-between representable numbers and get rounded.
Question 4: C & Assembly  [24 pts]

Answer the questions below about the following x86-64 assembly function:

```
.mystery:
    movl $0, %eax        # Line 1
.L2:  cmp %esi, %eax    # Line 2
     jge .L1            # Line 3
    movslq %eax, %rdx   # Line 4
    leaq (%rdi,%rdx,2), %rcx # Line 5
    movzwl (%rcx), %edx # Line 6
    addl $1, %edx       # Line 7
    movw %dx, (%rcx)    # Line 8
    addl $1, %eax       # Line 9
    jmp .L2             # Line 10
.L1:  retq             # Line 11
```

(A) What variable type would %rdi be in the corresponding C program?  [4 pt]

%rcx is calculated from %rdi with scale 2 (Line 5) and then __short*__ rdi dereferenced with a movzwl instruction (Line 6).

(B) Briefly describe why Line 4 is needed before Line 5. [4 pt]

Memory operands (Line 5) must take 64-bit register names, since addresses are 8 bytes wide. So the 4-byte value in %eax, must be extended to 8 bytes beforehand.

(C) This function uses a for loop. Fill in the corresponding parts below, using register names as variable names. None should be blank.  [8 pt]

```
for ( __eax = 0__ ; __eax < esi__ ; __eax++__ )
```

Init is from Line 1, Test is from Lines 2-3, Update is from Line 9.

(D) If we call this function with the value 3 as the second argument, what value is returned?  [4 pt]

Return value is %rax and we exit the loop when %eax = %esi.  3

(E) Describe at a high level what you think this function accomplishes (not line-by-line).  [4 pt]

 Overrides an array of shorts with the parity of the entries (1 for odd, 0 for even – given by the least significant bit).
Question 5: Procedures & The Stack  [20 pts]

The recursive power function \texttt{power()} calculates \texttt{base}^\texttt{pow} and its x86-64 disassembly is shown below:

```c
int power(int base, unsigned int pow) {
    if (pow) {
        return base * power(base,pow-1);
    }
    return 1;
}
```

(A) How much space (in bytes) does this function take up in our final executable? [2 pt]

Count all bytes (middle columns) or subtract address of next instruction (0x4005bc) from 0x4005a0.

28 B

(B) Circle one: The label power will show up in which table(s) in the object file? [4 pt]

\textbf{Symbol Table} \hspace{1cm} \textbf{Relocation Table} \hspace{1cm} \textbf{Both Tables} \hspace{1cm} \textbf{Neither Table}

\texttt{power} is called in this file (recursively) and can be called by external files, so in both.

(C) Which register is being saved on the stack? [2 pt]

See \texttt{pushq} instruction (0x4005a4).

%rbx
(D) What is the return address to \texttt{power} that gets stored on the stack? Answer in hex. [2 pt]

The address of the instruction \textit{after call}. \hfill \textbf{0x4005af}

(E) Assume \texttt{main} calls \texttt{power}(8,3). Fill in the snapshot of memory below the top of the stack \textbf{in hex} as this call to \texttt{power} returns to \texttt{main}. For unknown words, write “unknown”. [6 pt]

\begin{center}
\begin{tabular}{|c|c|}
\hline
0x7ffffeca3f748 & \texttt{<ret addr to main>} \hfill \texttt{power}(8,3) \\
0x7ffffeca3f740 & \texttt{<original rbx>} \hfill \texttt{power}(8,2) \\
0x7ffffeca3f738 & \texttt{0x4005af <ret addr>} \hfill \texttt{power}(8,1) \\
0x7ffffeca3f730 & \texttt{0x8 <base>} \hfill \texttt{power}(8,0) \\
0x7ffffeca3f728 & \texttt{0x4005af <ret addr>} \hfill unknown \\
0x7ffffeca3f720 & \texttt{0x8 <base>} \hfill unknown \\
0x7ffffeca3f718 & \texttt{0x4005af <ret addr>} \hfill unknown \\
0x7ffffeca3f710 & \texttt{unknown} \hfill unknown \\
\hline
\end{tabular}
\end{center}

The base case doesn’t push \%rbx onto the stack, so 0x7ffffeca3f710 remains unknown.

(F) Harry the Husky claims that we could have gotten away with not pushing a register onto the stack in \texttt{power}. Is our intrepid school’s mascot correct or not? Briefly explain. [4 pt]

\textbf{Harry is correct! base doesn’t change between recursive calls and power doesn’t call other procedures, so there is no need to save \%rdi in \%rbx.}

In fact, if you compile the C function with an optimization flag of \texttt{-O2}, it doesn’t push \%rbx onto the stack!