1. Number Representation

Consider the binary value $110101_2$:

(a) Interpreting this value as an unsigned 6-bit integer, what is its value in decimal?

(b) If we instead interpret it as a signed (two’s complement) 6-bit integer, what would its value be in decimal?

(c) Assuming these are all signed two’s complement 6-bit integers, compute the result (leaving it in binary is fine) of each of the following additions. For each, indicate if it resulted in overflow.

\[
\begin{align*}
001001 & \quad 110001 & \quad 011001 & \quad 101111 \\
+ 110110 & \quad + 111011 & \quad + 001100 & \quad + 011111
\end{align*}
\]

Result:

\[
\begin{array}{cccc}
\square & \square & \square & \square \\
\end{array}
\]

Overflow?

\[
\begin{array}{cccc}
\square & \square & \square & \square \\
\end{array}
\]
Now assume that our fictional machine with 6-bit integers also has a 6-bit IEEE-like floating point type, with 1 bit for the sign, 3 bits for the exponent \(\text{exp}\) with a bias of 3, and 2 bits to represent the mantissa \(\text{frac}\), not counting implicit bits.

(d) If we reinterpret the bits of our binary value from above as our 6-bit floating point type, what value, in decimal, do we get?

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign</td>
<td>exp</td>
<td>frac</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(e) If we treat 110101\textsubscript{2} as a signed integer, as we did in (b), and then cast it to a 6-bit floating point value, do we get the correct value in decimal? (That is, can we represent that value in our 6-bit float?) If yes, what is the binary representation? If not, why not? (and in that case you do not need to determine the rounded bit representation)

(f) Assuming the same rules as standard IEEE floating point, what value (in decimal) does the following represent?

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign</td>
<td>exp</td>
<td>frac</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Imagine we’re designing a new, super low-power computing device that will be powered by ambient radio waves (that part is actually a real research project). Our imaginary device’s CPU supports the x86-64 ISA, but its general-purpose integer multiply instruction (imul) is very bad and consumes lots of power. Luckily, we have learned several other ways to do multiplication in x86-64 in certain situations. To take advantage of these, we are designing a custom multiply function, spmult, that checks for specific arguments where we can use other instructions to do the multiplication. But we need your help to finish the implementation.

*Fill in the blanks with the correct instructions or operands.* It is okay to leave off size suffixes.

*Hint:* there are reference sheets with x86-64 registers and instructions at the end of the exam.

```c
long spmult(long x, long y) {
    if (y == 0) return 0;
    else if (y == 1) return x;
    else if (y == 4) return x * 4;
    else if (y == 5) return x * 5;
    else if (y == 16) return x * 16;
    else return x * y;
}
```

```assembly
spmult(long, long):
    testq $0
    js .L3
    .L1:
    movq 0(%rdi,4), %rax
    ret
    .L3:
    movq 0(%rax), %rdx
    ret
    .L4:
    movq 0,%rax
    ret
    .case4:
    # fall back to multiply
    movq %rsi, %rax
    imulq %rdi, %rax
    ret
```

3. Pointers and Memory

For this section, refer to this 8-byte aligned diagram of memory, with addresses increasing top-to-bottom and left-to-right (address 0x00 at the top left). When answering the questions below, don’t forget that x86-64 machines are little-endian. If you don’t remember exactly how endianness works, you should still be able to get significant partial credit.

```
int* x = 0x10;
long* y = 0x20;
char* s = 0x00;
```

<table>
<thead>
<tr>
<th>Memory 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>aa</td>
<td>bb</td>
<td>cc</td>
<td>dd</td>
<td>ee</td>
<td>ff</td>
<td>00</td>
<td>11</td>
</tr>
<tr>
<td>0x08</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>0x10</td>
<td>ab</td>
<td>01</td>
<td>51</td>
<td>f0</td>
<td>07</td>
<td>06</td>
<td>05</td>
<td>04</td>
</tr>
<tr>
<td>0x18</td>
<td>de</td>
<td>ad</td>
<td>be</td>
<td>ef</td>
<td>10</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>0x20</td>
<td>ba</td>
<td>ca</td>
<td>ff</td>
<td>ff</td>
<td>1a</td>
<td>2b</td>
<td>3c</td>
<td>4d</td>
</tr>
<tr>
<td>0x28</td>
<td>a0</td>
<td>b0</td>
<td>c0</td>
<td>d0</td>
<td>a1</td>
<td>b1</td>
<td>c1</td>
<td>d1</td>
</tr>
</tbody>
</table>

(a) Fill in the type and value for each of the following C expressions:

<table>
<thead>
<tr>
<th>Expression (in C)</th>
<th>Type</th>
<th>Value (in hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>*x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*(y-1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s[4]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) Assume that all registers start with the value 0, except %rax which is set to 8. Determine what the final values of each of these registers will be after executing the following instructions:

```
movb %al, %bl
leal 2(%rax),%ecx
movsbw (,%rax,4),%dx
```
4. The Stack

The recursive factorial function `fact()` and its x86-64 disassembly is shown below:

```c
int fact(int n) {
    if(n==0 || n==1)
        return 1;
    return n*fact(n-1);
}
```

```assembly
000000000040052d <fact>:  
  40052d:  83 ff 00   cmpl  $0, %edi
  400530:  74 05     je    400537 <fact+0xa>
  400532:  83 ff 01   cmpl  $1, %edi
  400535:  75 07     jne   40053e <fact+0x11>
  400537:  b8 01 00 00 00 movl  $1, %eax
  40053c:  eb 0d     jmp   40054b <fact+0x1e>
  40053e:  57        pushq %rdi
  40053f:  83 ef 01   subl  $1, %edi
  400542:  e8 e6 ff ff ff call  40052d <fact>
  400547:  5f        popq  %rdi
  400548:  0f af c7   imull %edi, %eax
  40054b:  f3 c3     rep ret
```

(A) Circle one: `fact()` is saving `%rdi` to the Stack as a [ ] Caller // [ ] Callee

(B) How much space (in bytes) does this function take up in our final executable?

(C) **Stack overflow** is when the stack exceeds its limits (i.e. runs into the Heap). Provide an argument to `fact(n)` here that will cause stack overflow.

«Problem continued on next page»
(D) If we use the `main` function shown below, answer the following for the execution of the entire program:

```c
void main() {
    printf("result = %d\n", fact(3));
}
```

<table>
<thead>
<tr>
<th>Total frames created:</th>
<th>Maximum stack frame depth:</th>
</tr>
</thead>
</table>

(E) In the situation described above where `main()` calls `fact(3)`, we find that the word `0x2` is stored on the Stack at address `0x7fffdc7ba888`. At what address on the Stack can we find the return address to `main()`?

```c

```