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?

$$2^5 + 2^4 + 2^2 + 2^0 = 32 + 16 + 4 + 1 = 53$$

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

$$-2^5 + 2^4 + 2^2 + 2^0 = -32 + 16 + 4 + 1 = -11$$

*(most significant bit becomes "negatively weighted")*

(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.

Note: TMIN = -32

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>001001</td>
<td>-15</td>
<td>110001</td>
<td>011001</td>
<td>101111</td>
</tr>
<tr>
<td>-10</td>
<td>+110110</td>
<td>-5</td>
<td>+111011</td>
<td>+001100</td>
<td>+011111</td>
</tr>
</tbody>
</table>

Result:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>111111</td>
<td>↓–101100</td>
<td>100101</td>
<td>↓–001110</td>
<td></td>
</tr>
</tbody>
</table>

Overflow?

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

*Overflow only occurs for signed addition if the result comes out wrong. The easiest way to determine this is by looking at the signs: if 2 positive values result in a negative result, or 2 negatives result in a positive, then overflow must have occurred.*
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 (exp) with a bias of 3, and 2 bits to represent the mantissa (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?

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

\[-1.012 \times 2^{(4+1-3)} = -1.012 \times 2^2 = -1012 = -5\]

(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)

No, we cannot represent it exactly because there are not enough bits for the mantissa.

To determine this, we have to find out what the mantissa would be once we are in "sign-and-magnitude" style: 110101 (-11) → 001011 (+11). In normalized form, this would be: \((-1)^1 \times 1.011 \times 2^3\), which means frac would need to be 011, which doesn’t fit in 2 bits.

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

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

0.0 (it is a denormalized case)
2. C to Assembly

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 %rsi, %rsi
    je .L3
    cmpq $1, %rsi
    je .L4
    cmpq $4, %rsi
    jne .L1
    .case4:
    leaq 0(%rdi,4), %rax
    ret
    .L1:
    cmpq $5, %rsi
    jne .L2
    leaq (%rdi,%rdi,4), %rax
    ret
    .L2:
    cmpq $16, %rsi
    jne .else
    movq %rdi, %rax
    salq $4, %rax
    ret
    .L3:
    movq $0, %rax
    ret
    .L4:
    movq %rdi, %rax
    ret
    .else: # 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.

\[
\begin{array}{|c|c|c|c|c|c|c|c|}
\hline
\text{Memory Address} & +0 & +1 & +2 & +3 & +4 & +5 & +6 & +7 \\
\hline
0x00 & aa & bb & cc & dd & ee & ff & 00 & 11 \\
0x08 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\
0x10 & ab & 01 & 51 & f0 & 07 & 06 & 05 & 04 \\
0x18 & de & ad & be & ef & 10 & 00 & 00 & 00 \\
0x20 & ba & ca & ff & ff & 1a & 2b & 3c & 4d \\
0x28 & a0 & b0 & c0 & d0 & a1 & b1 & c1 & d1 \\
\hline
\end{array}
\]

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

(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>int</td>
<td>0x0f05101ab</td>
</tr>
<tr>
<td>x+1</td>
<td>int*</td>
<td>0x14</td>
</tr>
<tr>
<td>*(y-1)</td>
<td>long</td>
<td>0x00000010efbeadde</td>
</tr>
<tr>
<td>s[4]</td>
<td>char</td>
<td>0xEE</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:

\[
\begin{align*}
&\text{movb } %al, %bl \\
&\text{leal } 2(%rax), %ecx \\
&\text{movsbw (},%rax,4), %dx
\end{align*}
\]

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rax</code></td>
<td>8</td>
</tr>
<tr>
<td><code>%bl</code></td>
<td>8 or 0x8</td>
</tr>
<tr>
<td><code>%ecx</code></td>
<td>10 or 0xa</td>
</tr>
<tr>
<td><code>%dx</code></td>
<td>65466 or 0xffba</td>
</tr>
</tbody>
</table>
4. The Stack

The recursive factorial function \texttt{fact()} and its x86-64 disassembly is shown below:

\begin{verbatim}
int fact(int n) {
    if(n==0 || n==1)
        return 1;
    return n*fact(n-1);
}
\end{verbatim}

\begin{verbatim}
000000000040052d <fact>:
  40052d:  83 ff 00  cmp $0, %edi
  400530:  74 05  je  400537 <fact+0xa>
  400532:  83 ff 01  cmp $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
\end{verbatim}

(A) Circle one: \texttt{fact()} is saving \%rdi to the Stack as a \textbf{Caller} // \textbf{Callee}

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

\begin{itemize}
    \item Count all bytes (middle columns) or subtract address of next instruction (0x40054d) from 0x40052d.
    \item \textbf{32 B}
\end{itemize}

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

\begin{itemize}
    \item \textbf{Any negative int}
\end{itemize}

We did mention in the lecture slides that the Stack has 8 MiB limit in x86-64, so since 16B per stack frame, credit for anything between 512 and TMax ($2^{31}$-1).
(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>
<tbody>
<tr>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

main → fact(3) → fact(2) → fact(1) → printf

(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()`?

```
0x7fffdc7ba8a0
```

Only `%rdi (current n)` and return address get pushed onto Stack during `fact()`.

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7fffdc7ba8a0</td>
<td>Return addr to main()</td>
</tr>
<tr>
<td>0x7fffdc7ba898</td>
<td>Old <code>%rdi (n=3)</code></td>
</tr>
<tr>
<td>0x7fffdc7ba890</td>
<td>Return addr to fact()</td>
</tr>
<tr>
<td>0x7fffdc7ba888</td>
<td>Old <code>%rdi (n=2)</code></td>
</tr>
<tr>
<td>0x7fffdc7ba880</td>
<td>Return addr to fact()</td>
</tr>
</tbody>
</table>