Floating Point Representation

COMP370
Introduction to Computer Architecture

Binary Fractions

- Each position is twice the value of the position to the right.

<table>
<thead>
<tr>
<th>2^3</th>
<th>2^2</th>
<th>2^1</th>
<th>2^0</th>
<th>.</th>
<th>2^-1</th>
<th>2^-2</th>
<th>2^-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>.</td>
<td>1/2</td>
<td>1/4</td>
<td>1/8</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>.</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Adding the powers of 2 gives
8+4+2+0.25 = 14.25

What is 111.11 in decimal?

1. 7.75
2. 31
3. 7.375
4. 15.25

What is 8.5 in binary?

1. 11111111.11111
2. 1000.01
3. 0.100011
4. 1000.10
Range of Values

• Unsigned integers: 0 to $2^n-1$
  – For byte, from 0 to 255
  – For int, from 0 to $4.2\times10^9$
• 2’s complement: $-(2^n-1)$ to $+(2^n-1)$
  – For byte, from -128 to 127
  – For short, from -32768 to 32767
  – For int, from -2147483648 to 2147483647
  – For long, from -$9.2\times10^{18}$ to $9.2\times10^{18}$

Scientific Notation

241,506,800 = $0.2415068 \times 10^9$

Binary Scientific Notation

• A binary number, such as 10110011, can be expressed as:
  
  $1.0110011 \times 2^7$

• Note the exponent is a power of two not ten.
Shifting Binary Exponents

• A binary number can be expressed in “scientific” notation is several ways like decimal numbers.

<table>
<thead>
<tr>
<th>Binary</th>
<th>Scientific</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.110010 x 2^5</td>
<td>0.78125 x 32 = 25</td>
</tr>
<tr>
<td>1.10010 x 2^4</td>
<td>1.5625 x 16 = 25</td>
</tr>
<tr>
<td>11.0010 x 2^3</td>
<td>3.125 x 8 = 25</td>
</tr>
<tr>
<td>110.010 x 2^2</td>
<td>6.25 x 4 = 25</td>
</tr>
<tr>
<td>1100.10 x 2^1</td>
<td>12.5 x 2 = 25</td>
</tr>
<tr>
<td>11001.0 x 2^0</td>
<td>25 x 1 = 25</td>
</tr>
</tbody>
</table>

110.010 is equivalent to

1. 11001.0 x 2^{-2}
2. 0.110010 x 2^3
3. 6.25
4. All of the above
5. None of the above

Standard Format

• Most computers (including Intel Pentiums) follow the *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Standard 754-1985
• Before the standard different computers used different formats for floating point numbers.
• The standard defines the format, accuracy and action taken when errors are detected.

Floating-point Sizes

• ANSI/IEEE Standard 754-1985
  – Single-precision (32 bits)
  – Double-precision (64 bits)
  – Extended-precision (80 bits)
Single-Precision Floating-point Numbers

- **float** variables in C++ or Java

Signed Magnitude
- For positive numbers, the sign bit is zero
- For negative numbers, the sign bit is one and everything else is the same

Single Precision Float Range
- A little more than 7 decimal digits accuracy
- From $-3.4 \times 10^{38}$ to $3.4 \times 10^{38}$
- Positive numbers can be as small as $1.18 \times 10^{-38}$ before going to zero.

Double Precision Floating Point

- **double** variables in C++ or Java
- Approximately 16 decimal digits of accuracy
- From $-1.798 \times 10^{308}$ to $1.798 \times 10^{308}$

Extended Precision Floating Point

- Almost 20 decimal digits of accuracy
- $3.37 \times 10^{-4932}$ to $1.18 \times 10^{4932}$
- Not directly supported in C++ or Java
- Often used internally for calculations which are then rounded to desired precision
Exponent Bias

- The exponent represents the power of 2.
- The single precision exponent is biased by adding 127 to the actual exponent.
- This avoids an extra sign bit for the exponent.
- The exponent range is -126 to 128.

<table>
<thead>
<tr>
<th>Exponent value</th>
<th>Decimal exponent</th>
<th>Binary exponent</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^0$</td>
<td>127</td>
<td>01111111</td>
</tr>
<tr>
<td>$2^5$</td>
<td>132</td>
<td>10000100</td>
</tr>
<tr>
<td>$2^{-5}$</td>
<td>122</td>
<td>01111010</td>
</tr>
</tbody>
</table>

Normalization

- Floating point numbers are adjusted so the mantissa or fractional part has a single “1” bit before the radix point.

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Normalized</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.75</td>
<td>101.11 $2^0$</td>
<td>$1.0111 \times 2^2$</td>
</tr>
<tr>
<td>0.125</td>
<td>0.001 $2^0$</td>
<td>$1.0 \times 2^{-3}$</td>
</tr>
<tr>
<td>32.0</td>
<td>100000.0 $2^0$</td>
<td>$1.0 \times 2^5$</td>
</tr>
</tbody>
</table>

Saving a Bit

- The fractional part or mantissa is always adjusted so the leftmost bit is a one.
- Since this bit is *always* a one, it is not actually stored in the floating point number.
- The mantissa is stored without the leading one bit although the one bit is assumed in calculating the value of the number.

Creating a Floating Point Number

1. Write the number in binary with a fractional part as necessary.
2. Adjust the exponent so the radix point is to the right of the first one bit.
3. The mantissa is the binary number without the leading one bit.
4. The exponent field is created by adding 127 to the binary exponent.
5. The sign is the same as the number’s sign.
### Decimal to Floating Point Example

- Convert 4.5 to single precision floating point
- Decimal 4.5 is 100.1 in binary
- Adjust radix to get $1.001 \times 2^2$
- The exponent field is $127+2 = 129 = 10000001$
- The floating point number in binary is

<table>
<thead>
<tr>
<th>S</th>
<th>Exponent</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000001</td>
<td>001000000000000000000000</td>
</tr>
</tbody>
</table>

### Convert 15.375 to Floating Point

- Decimal 15.375 is 1111.011 in binary
- Adjust the exponent to $1.111011 \times 2^3$
- Exponent field is $3 + 127 = 130_{10} = 10000010_2$

<table>
<thead>
<tr>
<th>S</th>
<th>Exponent</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1000001</td>
<td>010010000000000000000000</td>
</tr>
</tbody>
</table>

### Floating Point to Decimal Example

- What is the decimal value of this number?
  - Exponent 10000011 = 131 – 127 = 4
  - Mantissa is $1.010010000000000000000000$
  - $-1.01001 \times 2^4 = -10100.1$
  - -10100.1 is -20.5 in decimal
What is the decimal value of

<table>
<thead>
<tr>
<th>S</th>
<th>Exponent</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10000001</td>
<td>101000000000000000000000</td>
</tr>
</tbody>
</table>

1. 4.5
2. 3.25
3. 6.5
4. 13.0

Special Floating Point Values

- **Zero** is represented as all zero bits.
- Not a Number (**NaN**) is a special value that indicates a floating point error, such as taking the square root of a negative number.
- **Infinity** (**INF**) both positive and negative.

Special Value Representation

<table>
<thead>
<tr>
<th>Value</th>
<th>Sign</th>
<th>Exponent</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>+INF</td>
<td>0</td>
<td>11111111</td>
<td>0</td>
</tr>
<tr>
<td>-INF</td>
<td>1</td>
<td>11111111</td>
<td>0</td>
</tr>
<tr>
<td>NaN</td>
<td>0 or 1</td>
<td>0</td>
<td>Not zero</td>
</tr>
</tbody>
</table>

Overflow and Underflow

- When you calculate a number that is too big to fit into the floating point format, the result is infinity.
- Calculating a number that is too small (a positive number smaller than 1.18x10^{-38} for single precision) produces zero.
- Dividing by zero produces infinity with the proper sign.
Calculating with Infinity

• (+INF) + (+7) = (+INF)
• (+INF) × (−2) = (−INF)
• (+INF) × 0 = NaN—meaningless result