Binary Arithmetic
Binary Arithmetic

- Topics
  - Basic addition
  - Overflow
  - Multiplication & division
  - Floating-point preview
Basic addition

- Binary and hex addition are fundamentally the same as decimal addition
  - Add digit-by-digit, using a carry as necessary
  - Result could require one more bit than the operands

\[
\begin{array}{c}
\text{Dec} & \text{Bin} & \text{Hex} \\
12540 & 10011100 & \text{b0994f} \\
+ 4683 & + 1010110 & + 7120 \\
\end{array}
\]
Basic addition

- Binary and hex addition are fundamentally the same as decimal addition
  - Add digit-by-digit, using a carry as necessary
  - Result could require one more bit than the operands

\[
\begin{align*}
11 & \quad \text{Dec} & \quad 111 & \quad \text{Bin} \\
12540 & & 10011100 & \\
+4683 & & +1010110 & \\
17223 & & \text{11110010} & \\
\end{align*}
\]

\[
\begin{align*}
1 & \quad \text{Hex} \\
\text{b0994f} & & \text{b10a6f} & \\
+7120 & & & \\
\text{b10a6f} & & & \\
\end{align*}
\]
Overflow

- Unsigned addition is subject to overflow
  - Caused by truncation to integer size

\[
1 \quad 994f \\
+ \quad 7120 \\
\underline{10a6f} = 0a6f
\]

(assume a 16-bit integer)

Figure 2.23  Unsigned addition. With a 4-bit word size, addition is performed
modulo 16.
Overflow

- Two’s complement addition is identical to unsigned mechanically
  - Subject to both positive and negative overflow
  - Overflows if carry-in and carry-out differ for sign bit
  - Same for subtraction (overflows if borrow-in and borrow-out of sign bit differ)

*NOTE: this figure is printed incorrectly in your textbook!*
Overflow

- Examples (in 4-bit two’s complement):

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1</td>
<td>3</td>
</tr>
</tbody>
</table>
  + 0 0 1 0 + 2
  ---------
  0 1 0 1

  No carry in, no carry out (OK)

<table>
<thead>
<tr>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1</td>
</tr>
</tbody>
</table>
  + 0 1 0 0 + 4
  ---------
  0 0 0 1

  Carry in, carry out (OK)

<table>
<thead>
<tr>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1</td>
</tr>
</tbody>
</table>
  + 0 1 0 0 + 4
  ---------
  1 0 0 1

  Carry in, no carry out (OVERFLOW!)

<table>
<thead>
<tr>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1</td>
</tr>
</tbody>
</table>
  - 1 1 1 0 - 2
  ---------
  1 1 1 1

  No carry in, no carry out (OK)

<table>
<thead>
<tr>
<th>1 1 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1</td>
</tr>
</tbody>
</table>
  - 0 0 1 0 - 2
  ---------
  1 1 1 1

  Borrow in, borrow out (OK)

Observation: In two’s complement, adding the inverse is equivalent to subtracting!
Case study: MTG Arena

- “Evra, Halcyon Witness”
  - Card from Magic: The Gathering Arena (PC video game)
  - **Ability**: gain player life equal to Evra’s power (“lifelink”)
  - **Ability**: exchange player life total w/ Evra’s power
  - Alternate abilities to double life every few turns
  - Overflows at ~2 billion b/c player life is stored as a signed 32-bit integer

https://www.youtube.com/watch?v=8cqID9lpC3I
Multiplication & division

• Like addition, fundamentally the same as base 10
  - Actually, it’s even simpler!
  - Same regardless of encoding

• Special case: multiply by powers of 2 (shift left)
  
  \[
  \begin{align*}
  2 \ll 1 &= 4 & (2 \times 2) \\
  1 \ll 2 &= 4 & (1 \times 2 \times 2) \\
  1 \ll 4 &= 16 & (1 \times 2 \times 2 \times 2 \times 2) \\
  4 \ll 1 &= 8 & (4 \times 2) \\
  4 \ll 2 &= 16 & (4 \times 2 \times 2)
  \end{align*}
  \]

• Division is expensive!
  
  - Special case: divide by powers of two (shift right)
    • Logical shift for unsigned numbers, arithmetic shift for signed numbers
## Review

- **One-byte integers:**

<table>
<thead>
<tr>
<th>Binary</th>
<th>Unsigned</th>
<th>Two's C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111 1111</td>
<td>255</td>
<td>-1</td>
</tr>
<tr>
<td>1111 1110</td>
<td>254</td>
<td>-2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1000 0001</td>
<td>129</td>
<td>-127</td>
</tr>
<tr>
<td>1000 0000</td>
<td>128</td>
<td>-128</td>
</tr>
<tr>
<td>0111 1111</td>
<td>127</td>
<td>127</td>
</tr>
<tr>
<td>0111 1110</td>
<td>126</td>
<td>126</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0000 0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0000 0000</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- **Overflow**
  - when $x + y > 255$
  - Positive overflow when $x + y > 127$
  - Negative overflow when $x + y < -128$

---

*Figure 2.24*

Relation between integer and two's-complement addition. When $x + y$ is less than $-2^{w-1}$, there is a negative overflow. When it is greater than or equal to $2^{w-1}$, there is a positive overflow.
Binary fractions

• Now we can store integers
  – But what about general real numbers?
• Extend positional binary integers to store fractions
  – Designate a certain number of bits for the fractional part
  – These bits represent negative powers of two
  – (Just like fractional digits in decimal fractions!)

\[
\begin{align*}
101.101 & \quad 4 \quad 2 \quad 1 \quad 1/2 \quad 1/4 \quad 1/8 \\
4 + 1 + 0.5 + 0.125 &= 5.625 \\
\text{(alternatively: } 5 + 5/8)\end{align*}
\]
Another problem

• For scientific applications, we want to be able to store a wide range of values
  - From the scale of galaxies down to the scale of atoms
• Doing this with fixed-precision numbers is difficult
  - Even signed 64-bit integers
    • Perhaps allocate half for whole number, half for fraction
    • Range: $\sim 2 \times 10^{-9}$ through $\sim 2 \times 10^9$
Floating-point numbers

- Scientific notation to the rescue!
  - Traditionally, we write large (or small) numbers as \( x \cdot 10^e \)
  - This is how floating-point representations work
    - Store exponent and fractional parts (the significand) separately
    - The decimal point “floats” on the number line
    - Position of point is based on the exponent

\[
\begin{align*}
0.0123 \times 10^2 \\
0.123 \times 10^1 \\
1.23 \quad = \\
1.23 \times 10^0 \\
12.3 \times 10^{-1} \\
123.0 \times 10^{-2}
\end{align*}
\]
Floating-point numbers

• However, computers use binary
  - So floating-point numbers use base 2 scientific notation \((x \cdot 2^e)\)

• Fixed width field
  - Reserve one bit for the sign bit (0 is positive, 1 is negative)
  - Reserve \(n\) bits for biased exponent (bias is \(2^{n-1} - 1\))
    • Avoids having to use two’s complement
  - Use remaining bits for normalized fraction (implicit leading 1)
    • Exception: if the exponent is zero, don’t normalize

\[
2.5 \rightarrow \begin{array}{c}
0 \\
1000 \\
010
\end{array}
\]

Value = \((-1)^s \times 1.f \times 2^e\)