Worlframalpha.com
Facebook Report

Map of the world showing friends' locations.

Pie chart showing gender distribution:
- Male: 59.3% (134 friends)
- Female: 40.7% (92 friends)
(based on 226 of 240 friends)

Histogram showing age distribution:
(based on 52 of 240 friends)

Pie chart showing relationship status:
- Single: 9.9% (9 friends)
- In a relationship: 11.0% (10 friends)
- Engaged: 0% (0 friends)
- Married: 74.7% (68 friends)
- In a domestic partnership: 12.5% (11 friends)
(based on 91 of 240 friends)
• For next Tuesday: have read up through p.99 in textbook
• true defined?
  – stdbool.h (C99)

  #ifdef false
  #define false(0)
  #define true (!(false))
  #endif

• Disease spread: twitter
### Mapping Signed ↔ Unsigned

<table>
<thead>
<tr>
<th>Bits</th>
<th>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>-8</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>-7</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>-6</td>
<td>10</td>
</tr>
<tr>
<td>1011</td>
<td>-5</td>
<td>11</td>
</tr>
<tr>
<td>1100</td>
<td>-4</td>
<td>12</td>
</tr>
<tr>
<td>1101</td>
<td>-3</td>
<td>13</td>
</tr>
<tr>
<td>1110</td>
<td>-2</td>
<td>14</td>
</tr>
<tr>
<td>1111</td>
<td>-1</td>
<td>15</td>
</tr>
</tbody>
</table>

+16
Summary

Casting Signed ↔ Unsigned: Basic Rules

- Bit pattern is maintained
- But reinterpreted
- Can have unexpected effects: adding or subtracting $2^w$

- Expression containing signed and unsigned int
  - int is cast to unsigned!!
Summary:
Expanding, Truncating: Basic Rules

- **Expanding** (e.g., short int to int)
  - Unsigned: zeros added
  - Signed: sign extension
  - Both yield expected result

- **Truncating** (e.g., unsigned to unsigned short)
  - Unsigned/signed: bits are truncated
  - Result reinterpreted
  - Unsigned: mod operation
  - Signed: similar to mod
  - For small numbers yields expected behaviour
**Unsigned Addition**

Operands: \( w \) bits

\[
\begin{array}{c}
\text{u} \\
\text{+ v} \\
\text{u + v}
\end{array}
\]

True Sum: \( w+1 \) bits

Discard Carry: \( w \) bits

\( \text{UAdd}_w(u, v) \)

- **Standard Addition Function**
  - Ignores carry output

- **Implements Modular Arithmetic**

\[
\begin{align*}
s &= \text{UAdd}_w(u, v) \\
&= u + v \mod 2^w
\end{align*}
\]

\( \text{UAdd}_w(u, v) = \begin{cases} 
  u + v & u + v < 2^w \\
  u + v - 2^w & u + v \geq 2^w
\end{cases} \)
Visualizing Unsigned Addition

- Wraps Around
  - If true sum $\geq 2^w$
  - At most once

True Sum

- $2^{w+1}$
- $2^w$
- 0

Modular Sum

$UAdd_4(u, v)$

Overflow
Two’s Complement Addition

Operands: $w$ bits

True Sum: $w+1$ bits

Discard Carry: $w$ bits

$TAdd_w(u, v)$

- $TAdd$ and $UAdd$ have Identical Bit-Level Behavior
  - Signed vs. unsigned addition in C:
    ```
    int s, t, u, v;
    s = (int) ((unsigned) u + (unsigned) v);
    t = u + v
    Will give $s == t$
    ```
**TAdd Overflow**

**Functionality**
- True sum requires $w+1$ bits
- Drop off MSB
- Treat remaining bits as 2’s comp. integer

**True Sum**

<table>
<thead>
<tr>
<th>True Sum</th>
<th>TAdd Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$011...1$</td>
<td>$011...1$</td>
</tr>
<tr>
<td>$010...0$</td>
<td>$000...0$</td>
</tr>
<tr>
<td>$000...0$</td>
<td>$100...0$</td>
</tr>
<tr>
<td>$101...1$</td>
<td>$-2^w$</td>
</tr>
<tr>
<td>$1000...0$</td>
<td>$-2^w$</td>
</tr>
</tbody>
</table>

**PosOver**

**NegOver**
Visualizing 2’s Complement Addition

- **Values**
  - 4-bit two’s comp.
  - Range from -8 to +7

- **Wraps Around**
  - If sum $\geq 2^{w-1}$
    - Becomes negative
    - At most once
  - If sum $< -2^{w-1}$
    - Becomes positive
    - At most once
Characterizing TAdd

**Functionality**
- True sum requires \( w + 1 \) bits
- Drop off MSB
- Treat remaining bits as 2’s comp. integer

\[
TAdd_w(u, v) = \begin{cases} 
  u + v + 2^w & u + v < TMin_w \quad \text{(NegOver)} \\
  u + v & TMin_w \leq u + v \leq TMax_w \\
  u + v - 2^w & TMax_w < u + v \quad \text{(PosOver)}
\end{cases}
\]
Mathematical Properties of TAdd

- **Isomorphic Group to unsigneds with UAdd**
  - \( TAdd_w(u, v) = U2T(UAdd_w(T2U(u), T2U(v))) \)
  - Since both have identical bit patterns

- **Two’s Complement Under TAdd Forms a Group**
  - Closed, Commutative, Associative, 0 is additive identity
  - Every element has additive inverse

\[
TComp_w(u) = \begin{cases} 
-u & \text{if } u \neq TMin_w \\
TMin_w & \text{if } u = TMin_w 
\end{cases}
\]
Multiplication

- Computing Exact Product of $w$-bit numbers $x$, $y$
  - Either signed or unsigned

- Ranges
  - Unsigned: $0 \leq x \times y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
    - Up to $2w$ bits
  - Two’s complement min: $x \times y \geq (-2^{w-1}) \times (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$
    - Up to $2w-1$ bits
  - Two’s complement max: $x \times y \leq (-2^{w-1})^2 = 2^{2w-2}$
    - Up to $2w$ bits, but only for $(TMin_w)^2$

- Maintaining Exact Results
  - Would need to keep expanding word size with each product computed
  - Done in software by “arbitrary precision” arithmetic packages
Unsigned Multiplication in C

Operands: $w$ bits

True Product: $2w$ bits

Discard $w$ bits: $w$ bits

- Standard Multiplication Function
  - Ignores high order $w$ bits

- Implements Modular Arithmetic
  $$\text{UMult}_w(u, v) = u \cdot v \mod 2^w$$
Signed Multiplication in C

Operands: $w$ bits

True Product: $2w$ bits

Discard $w$ bits: $w$ bits

Standard Multiplication Function

- Ignores high order $w$ bits
- Some of which are different for signed vs. unsigned multiplication
- Lower bits are the same
Consider 3 bit #'s

- Two’s compliment: $100 = -4$
- $5 \times 3 = 15 \rightarrow 7$ because $001111 \rightarrow 111$
- $-3 \times 3$?
- $4 \times 7$?
- $-4 \times -1$?
How could you test for * overflow?

```c
int t_mult_ok(int x, int y){
    int p = x*y
    return ( .... )
}
```
Power-of-2 Multiply with Shift

Operation
- $u \ll k$ gives $u \times 2^k$
- Both signed and unsigned

Operands: $w$ bits
- $u \times 2^k$
- $2^k$

True Product: $w+k$ bits
- $u \cdot 2^k$

Discard $k$ bits: $w$ bits
- $UMult_w(u, 2^k)$
- $TMult_w(u, 2^k)$

Examples
- $u \ll 3 = u \times 8$
- $u \ll 5 - u \ll 3 = u \times 24$
- Most machines shift and add faster than multiply
  - Compiler generates this code automatically
Compiled Multiplication Code

C Function

```c
int mul12(int x)
{
    return x*12;
}
```

Compiled Arithmetic Operations

```
leal (%eax,%eax,2), %eax
sall $2, %eax
```

Explanation

```
t <- x+x*2
return t << 2;
```

- C compiler automatically generates shift/add code when multiplying by constant
• Google Glass
  – http://www.youtube.com/watch?v=9c6W4CCU9M4
  – http://www.youtube.com/watch?v=t3TAOYXT840
Decimal Numbers

• 1234.5678
Fractional Binary Numbers

Representation
- Bits to right of "binary point" represent fractional powers of 2
- Represents rational number: \[ \sum_{k=-j}^{i} b_k \cdot 2^k \]
# Fractional Binary Numbers: Examples

<table>
<thead>
<tr>
<th>Value</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-3/4</td>
<td>101.11₂</td>
</tr>
<tr>
<td>2-7/8</td>
<td>10.111₂</td>
</tr>
<tr>
<td>63/64</td>
<td>0.1111111₁₂</td>
</tr>
</tbody>
</table>

## Observations
- Divide by 2 by shifting right
- Multiply by 2 by shifting left
- Numbers of form $0.\overline{11111}_2$ are just below 1.0
  - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$
  - Use notation $1.0 - \varepsilon$

• Limitations?
Representable Numbers

Limitation
- Can only exactly represent numbers of the form \(x/2^k\)
- Other rational numbers have repeating bit representations

<table>
<thead>
<tr>
<th>Value</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/3</td>
<td>0.0101010101[01]..._2</td>
</tr>
<tr>
<td>1/5</td>
<td>0.001100110011[0011]..._2</td>
</tr>
<tr>
<td>1/10</td>
<td>0.0001100110011[0011]..._2</td>
</tr>
</tbody>
</table>

Example: Patriot Missile in 1\(^{st}\) Gulf War
Heard of them?

- 6-bit IEEE-like format
IEEE Floating Point

- IEEE Standard 754
  - Established in 1985 as uniform standard for floating point arithmetic
    - Before that, many idiosyncratic formats
  - Supported by all major CPUs

- Driven by numerical concerns
  - Nice standards for rounding, overflow, underflow
  - Hard to make fast in hardware
    - Numerical analysts predominated over hardware designers in defining standard
Floating Point Representation

- **Numerical Form:**
  
  \[(−1)^s \ M \ 2^E\]

  - Sign bit \(s\) determines whether number is negative or positive
  - Significand \(M\) normally a fractional value in range \([1.0,2.0)\).
  - Exponent \(E\) weights value by power of two

- **Encoding**
  
  - MSB \(s\) is sign bit \(s\)
  - \(exp\) field encodes \(E\) (but is not equal to \(E\))
  - \(frac\) field encodes \(M\) (but is not equal to \(M\))
Precisions

- **Single precision**: 32 bits
  - s: 1 bit
  - exp: 8 bits
  - frac: 23 bits

- **Double precision**: 64 bits
  - s: 1 bit
  - exp: 11 bits
  - frac: 52 bits

- **Extended precision**: 80 bits (Intel only)
  - s: 1 bit
  - exp: 15 bits
  - frac: 63 or 64 bits