15-740

Computer Arithmetic
A Programmer’s View
Oct. 6, 1998

Topics

• Integer Arithmetic
  – Unsigned
  – Two’s Complement

• Floating Point
  – IEEE Floating Point Standard
  – Alpha floating point
Notation

$W$: Number of Bits in “Word”

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Sun, etc.</th>
<th>Alpha</th>
</tr>
</thead>
<tbody>
<tr>
<td>long int</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>char</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Integers

- Lower case
- E.g., $x, y, z$

Bit Vectors

- Upper Case
- E.g., $X, Y, Z$
- Write individual bits as integers with value 0 or 1
- E.g., $X = x_{w-1}, x_{w-2}, \ldots, x_0$
  - Most significant bit on left
Encoding Integers

Unsigned

Two’s Complement

\[ B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i \]

\[ B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i \]

- C short 2 bytes long

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>15740</td>
<td>00111101 01111100</td>
</tr>
<tr>
<td>y</td>
<td>-15740</td>
<td>11000010 10000100</td>
</tr>
</tbody>
</table>

Sign Bit

- For 2’s complement, most significant bit indicates sign
  - 0 for nonnegative
  - 1 for negative
Numeric Ranges

Unsigned Values
- $UMin = 0$
  000...0
- $UMax = 2^w - 1$
  111...1

Two’s Complement Values
- $TMin = -2^{w-1}$
  100...0
- $TMax = 2^{w-1} - 1$
  011...1

Other Values
- Minus 1
  111...1

Values for $W = 16$

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$UMax$</td>
<td>65535</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>$TMax$</td>
<td>32767</td>
<td>7F FF</td>
<td>01111111 11111111</td>
</tr>
<tr>
<td>$TMin$</td>
<td>-32768</td>
<td>80 00</td>
<td>10000000 00000000</td>
</tr>
<tr>
<td>$-1$</td>
<td>-1</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>$0$</td>
<td>0</td>
<td>00 00</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>
## Values for Different Word Sizes

<table>
<thead>
<tr>
<th></th>
<th>W=8</th>
<th>W=16</th>
<th>W=32</th>
<th>W=64</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>255</td>
<td>65,535</td>
<td>4,294,967,295</td>
<td>18,446,744,073,709,551,615</td>
</tr>
<tr>
<td>Tmax</td>
<td>127</td>
<td>32,767</td>
<td>2,147,483,647</td>
<td>9,223,372,036,854,775,807</td>
</tr>
<tr>
<td>Tmin</td>
<td>−128</td>
<td>−32,768</td>
<td>−2,147,483,648</td>
<td>−9,223,372,036,854,775,808</td>
</tr>
</tbody>
</table>

### Observations
- $|TMin| = Tmax + 1$
  - Asymmetric range
- $UMax = 2 \times Tmax + 1$

### C Programming
- `#include <limits.h>`
  - K&R Appendix B11
- Declares constants, e.g.,
  - `ULONG_MAX`
  - `LONG_MAX`
  - `LONG_MIN`
- Values platform-specific
Unsigned & Signed Numeric Values

Example Values

- \( W = 4 \)

Equivalence

- Same encodings for nonnegative values

Uniqueness

- Every bit pattern represents unique integer value
- Each representable integer has unique bit encoding

⇒ Can Invert Mappings

- \( U2B(x) = B2U^{-1}(x) \)
  - Bit pattern for unsigned integer
- \( T2B(x) = B2T^{-1}(x) \)
  - Bit pattern for two's comp integer

<table>
<thead>
<tr>
<th>( x )</th>
<th>( B2U(x) )</th>
<th>( B2T(x) )</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>9</td>
<td>-7</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
<td>-6</td>
</tr>
<tr>
<td>1011</td>
<td>11</td>
<td>-5</td>
</tr>
<tr>
<td>1100</td>
<td>12</td>
<td>-4</td>
</tr>
<tr>
<td>1101</td>
<td>13</td>
<td>-3</td>
</tr>
<tr>
<td>1110</td>
<td>14</td>
<td>-2</td>
</tr>
<tr>
<td>1111</td>
<td>15</td>
<td>-1</td>
</tr>
</tbody>
</table>
Casting Signed to Unsigned

C Allows Conversions from Signed to Unsigned

```
short int x = 15740;
unsigned short int ux = (unsigned short) x;
short int y = -15740;
unsigned short int uy = (unsigned short) y;
```

Resulting Value

- No change in bit representation
- Nonnegative values unchanged
  
  \[ ux = 15740 \]

- Negative values change into (large) positive values
  
  \[ uy = 49796 \]
Relation Between 2’s Comp. & Unsigned

Two’s Complement → T2U → T2B → B2U → Unsigned

Maintain Same Bit Pattern

\[ \begin{align*}
ux &= \begin{cases} 
x & x \geq 0 \\
x + 2^w & x < 0 \end{cases} 
\end{align*} \]

\[ +2^{w-1} - 2^{w-1} = 2^w \]

\[ +2^{w-1} - 2^{w-1} = 2^w \]
Signed vs. Unsigned in C

Constants

• By default are considered to be signed integers
• Unsigned if have “U” as suffix
  \[ 0U, \; 4294967259U \]

Casting

• Explicit casting between signed & unsigned same as U2T and T2U
  \[
  \begin{align*}
    \text{int } \& \text{ ty; } \\
    \text{unsigned } \& \text{ ux, uy; } \\
    \text{tx } = \text{(int) } \& \text{ ux; } \\
    \text{uy } = \text{(unsigned) } \& \text{ ty; }
  \end{align*}
  \]
• Implicit casting also occurs via assignments and procedure calls
  \[
  \begin{align*}
    \text{tx } &= \text{ ux; } \\
    \text{uy } &= \text{ ty; }
  \end{align*}
  \]
Casting Surprises

Expression Evaluation

- If mix unsigned and signed in single expression, signed values implicitly cast to unsigned
- Including comparison operations <, >, ==, <=, >=
- Examples for $W = 32$

<table>
<thead>
<tr>
<th>Constant&lt;sub&gt;1&lt;/sub&gt;</th>
<th>Constant&lt;sub&gt;2&lt;/sub&gt;</th>
<th>Relation</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0U</td>
<td>==</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>&lt;</td>
<td>signed</td>
</tr>
<tr>
<td>-1</td>
<td>0U</td>
<td>&gt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>-2147483648</td>
<td>&gt;</td>
<td>signed</td>
</tr>
<tr>
<td>2147483647U</td>
<td>-2147483648</td>
<td>&lt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>-2</td>
<td>&gt;</td>
<td>signed</td>
</tr>
<tr>
<td>(unsigned) -1</td>
<td>-2</td>
<td>&gt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>2147483648U</td>
<td>&lt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>(int) 2147483648U</td>
<td>&gt;</td>
<td>signed</td>
</tr>
</tbody>
</table>
Explanation of Casting Surprises

2’s Comp. → Unsigned

- Ordering Inversion
- Negative → Big Positive
Sign Extension

Task:
• Given \( w \)-bit signed integer \( x \)
• Convert it to \( w+k \)-bit integer with same value

Rule:
• Make \( k \) copies of sign bit:
• \( X' = \underbrace{x_{w-1}, \ldots, x_{w-1}}_{\text{k copies of MSB}}, x_{w-1}, x_{w-2}, \ldots, x_0} \)
Justification For Sign Extension

Prove Correctness by Induction on $k$

- Induction Step: extending by single bit maintains value

- Key observation: $-2^{w-1} = -2^w + 2^{w-1}$
- Look at weight of upper bits:

\[
\begin{align*}
X &= -2^{w-1} x_{w-1} \\
X' &= -2^w x_{w-1} + 2^{w-1} x_{w-1} = -2^{w-1} x_{w-1}
\end{align*}
\]
Integer Operation C Puzzles

• Assume machine with 32 bit word size, two’s complement integers
• For each of the following C expressions, either:
  – Argue that is true for all argument values
  – Give example where not true

  

<table>
<thead>
<tr>
<th>C Expression</th>
<th>Implication</th>
</tr>
</thead>
<tbody>
<tr>
<td>x &lt; 0</td>
<td>((x \times 2) &lt; 0)</td>
</tr>
<tr>
<td>ux \geq 0</td>
<td></td>
</tr>
<tr>
<td>x &amp; 7 == 7</td>
<td>((x \ll 30) &lt; 0)</td>
</tr>
<tr>
<td>ux &gt; -1</td>
<td></td>
</tr>
<tr>
<td>x &gt; y</td>
<td>(-x &lt; -y)</td>
</tr>
<tr>
<td>x * x \geq 0</td>
<td></td>
</tr>
<tr>
<td>x &gt; 0 &amp;&amp; y &gt; 0</td>
<td>(x + y &gt; 0)</td>
</tr>
<tr>
<td>x \geq 0</td>
<td>(-x \leq 0)</td>
</tr>
<tr>
<td>x \leq 0</td>
<td>(-x \geq 0)</td>
</tr>
</tbody>
</table>

Initialization

```c
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```
Unsigned Addition

Operands: w bits

True Sum: w+1 bits

Discard Carry: w bits

Standard Addition Function
  • Ignores carry output

Implements Modular Arithmetic

\[ s = UAdd_w(u, v) = u + v \mod 2^w \]

\[
UAdd_w(u, v) = \begin{cases} 
  u + v & u + v < 2^w \\
  u + v - 2^w & u + v \geq 2^w 
\end{cases}
\]
Visualizing Integer Addition

Integer Addition

• 4-bit integers \( u \) and \( v \)
• Compute true sum \( \text{Add}_4(u, v) \)
• Values increase linearly with \( u \) and \( v \)
• Forms planar surface
Visualizing Unsigned Addition

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

True Sum

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

Modular Sum

Overflow

Overflow
Mathematical Properties

Modular Addition Forms an *Abelian Group*

- **Closed under addition**
  \[ 0 \leq \text{UAdd}_w(u, v) \leq 2^w - 1 \]
- **Commutative**
  \[ \text{UAdd}_w(u, v) = \text{UAdd}_w(v, u) \]
- **Associative**
  \[ \text{UAdd}_w(t, \text{UAdd}_w(u, v)) = \text{UAdd}_w(\text{UAdd}_w(t, u), v) \]
- **$0$ is additive identity**
  \[ \text{UAdd}_w(u, 0) = u \]
- **Every element has additive inverse**
  - Let \[ \text{UComp}_w(u) = 2^w - u \]
  \[ \text{UAdd}_w(u, \text{UComp}_w(u)) = 0 \]
Two’s Complement Addition

Operands: \( w \) bits

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

True Sum: \( w+1 \) bits

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

Discard Carry: \( w \) bits

\[
\text{TAdd}_w(u, v)
\]

TAdd and UAdd have Identical Bit-Level Behavior

- Signed vs. unsigned addition in C:
  \[
  \begin{align*}
  \text{int } & \text{s, t, u, v;} \\
  \text{s} & = (\text{int}) ((\text{unsigned}) \text{u} + (\text{unsigned}) \text{v}); \\
  \text{t} & = \text{u} + \text{v}
  \end{align*}
  \]
- Will give \( s == t \)
Characterizing TAdd

Functionality

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

True Sum

\[ \begin{align*}
0 & 111...1 \\
0 & 100...0 \ 
\quad 2^{w-1} \\
0 & 000...0 \ 
\quad 0 \\
1 & 100...0 \ 
\quad -2^{w-1} \\
1 & 000...0 \ 
\quad -2^w
\end{align*} \]

TAdd Result

\[ \begin{align*}
\text{PosOver} & \quad 011...1 \\
\text{NegOver} & \quad 000...0 \quad 100...0
\end{align*} \]

\[ TAdd_w(u,v) = \begin{cases} 
    u + v + 2^{w-1} & u + v < Tmin_w \quad \text{(NegOver)} \\
    u + v & Tmin_w \leq u + v \leq Tmax_w \\
    u + v - 2^{w-1} & Tmax_w < u + v \quad \text{(PosOver)}
\end{cases} \]
Visualizing 2’s Comp. 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
Mathematical Properties of TAdd

Isomorphic Algebra to 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
  - Let \( TComp_w(u) = U2T(UComp_w(T2U(u))) \)
  \( TAdd_w(u, TComp_w(u)) = 0 \)

\[
TComp_w(u) = \begin{cases} 
-u & u \neq TMin_w \\
TMin_w & u = TMin_w 
\end{cases}
\]
Two’s Complement Negation

Mostly like Integer Negation
• TComp(u) = −u

TMin is Special Case
• TComp(TMin) = TMin

Negation in C is Actually TComp

mx = −x
• mx = TComp(x)
• Computes additive inverse for TAdd

x + −x == 0
Negating with Complement & Increment

In C

\[ \sim x + 1 = -x \]

Complement

- Observation: \[ \sim x + x = 111\ldots1_2 = -1 \]

\[
\begin{array}{l}
x & |& 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\
\hline
\sim x & |& 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\
\hline
\sim x + 1 & |& 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

Warning: Be cautious treating int’s as integers

- OK here: We are using group properties of TAdd and TComp

\textit{class07.ppt}
Comparing Two’s Complement Numbers

Task

- **Given** signed numbers $u$, $v$
- Determine whether or not $u > v$
  - Return 1 for numbers in shaded region below

Bad Approach

- Test $(u - v) > 0$
  - $\text{TSub}(u, v) = \text{TAdd}(u, \text{TComp}(v))$
- Problem: Thrown off by either Negative or Positive Overflow
Comparing with TSub

Will Get Wrong Results

- **NegOver**: \( u < 0, v > 0 \)  
  - but \( u-v > 0 \)
- **PosOver**: \( u > 0, v < 0 \)  
  - but \( u-v < 0 \)
Multiplication

Computing Exact Product of \( w \)-bit numbers \( x, y \)
- Either signed or unsigned

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

Maintaining Exact Results
- Would need to keep expanding word size with each product computed
- Done in software by “arbitrary precision” arithmetic packages
- Also implemented in Lisp, ML, and other “advanced” languages
Unsigned Multiplication in C

Operands: \( w \) bits

True Product: \( 2^w \) bits

Discard \( w \) bits: \( w \) bits

Standard Multiplication Function

- Ignores high order \( w \) bits

Implements Modular Arithmetic

\[
UMult_w(u, v) = u \cdot v \mod 2^w
\]
Unsigned vs. Signed Multiplication

Unsigned Multiplication

unsigned ux = (unsigned) x;
unsigned uy = (unsigned) y;
unsigned up = ux * uy

• Truncates product to \( w \)-bit number \( up = \text{UMult}_w(ux, uy) \)
• Simply modular arithmetic
  \[ up = ux \cdot uy \mod 2^w \]

Two’s Complement Multiplication

int x, y;
int p = x * y;

• Compute exact product of two \( w \)-bit numbers \( x, y \)
• Truncate result to \( w \)-bit number \( p = \text{TMult}_w(x, y) \)

Relation

• Signed multiplication gives same bit-level result as unsigned
  • \( up == (\text{unsigned}) p \)
Properties of Unsigned Arithmetic

Unsigned Multiplication with Addition Forms

Commutative Ring

• Addition is commutative group
• Closed under multiplication
  \[ 0 \leq \text{UMult}_w(u, v) \leq 2^w - 1 \]
• Multiplication Commutative
  \[ \text{UMult}_w(u, v) = \text{UMult}_w(v, u) \]
• Multiplication is Associative
  \[ \text{UMult}_w(t, \text{UMult}_w(u, v)) = \text{UMult}_w(\text{UMult}_w(t, u), v) \]
• 1 is multiplicative identity
  \[ \text{UMult}_w(u, 1) = u \]
• Multiplication distributes over addition
  \[ \text{UMult}_w(t, \text{UAdd}_w(u, v)) = \text{UAdd}_w(\text{UMult}_w(t, u), \text{UMult}_w(t, v)) \]
Properties of Two’s Comp. Arithmetic

Isomorphic Algebras

- Unsigned multiplication and addition
  - Truncating to $w$ bits
- Two’s complement multiplication and addition
  - Truncating to $w$ bits

Both Form Rings

- Isomorphic to ring of integers mod $2^w$

Comparison to Integer Arithmetic

- Both are rings
- Integers obey ordering properties, e.g.,
  \[ u > 0 \quad \Rightarrow \quad u + v > v \]
  \[ u > 0, \; v > 0 \quad \Rightarrow \quad u \cdot v > 0 \]
- These properties are not obeyed by two’s complement arithmetic
  \[ T_{Max} + 1 = T_{Min} \]
  \[ 15213 \times 30426 = -10030 \]
Integer C Puzzle Answers

- Assume machine with 32 bit word size, two’s complement integers
- $TMin$ makes a good counterexample in many cases

\[
\begin{align*}
\text{False: } & TMin \\
\text{True: } & 0 = UMin \\
\text{True: } & x_1 = 1 \\
\text{False: } & 0 \\
\text{False: } & -1, TMin \\
\text{False: } & 30426 \\
\text{False: } & TMax, TMax \\
\text{True: } & -TMax < 0 \\
\text{False: } & TMin
\end{align*}
\]
Floating Point Puzzles

• For each of the following C expressions, either:
  – Argue that is true for all argument values
  – Explain why not true

int x = ...;
float f = ...;
double d = ...;

Assume neither d nor f is NAN

• x == (int)(float) x
• x == (int)(double) x
• f == (float)(double) f
• d == (float) d
• f == -(-f);
• 2/3 == 2/3.0
• d < 0.0 ⇒ ((d*2) < 0.0)
• d > f ⇒ -f < -d
• d * d >= 0.0
• (d+f)−d == f
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 go fast
  - Numerical analysts predominated over hardware types in defining standard
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 Number 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>0.111111₁₂</td>
</tr>
</tbody>
</table>

Observation

- Divide by 2 by shifting right
- Numbers of form 0.1₁₁₁₁₁₁ᵰ₂ just below 1.0
  - Use notation 1.0 − ε

Limitation

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/3</td>
<td>0.0101010101₁₀₁₀₁₁₁₁ₙ₂</td>
</tr>
<tr>
<td>1/5</td>
<td>0.00110011₀₀₁₁₁₁ₙ₂</td>
</tr>
<tr>
<td>1/10</td>
<td>0.000₁₁₀₀₁₁₀₀₁₁₁₁ₙ₂</td>
</tr>
</tbody>
</table>
Floating Point Representation

Numerical Form

- \(-1^s \times m \times 2^E\)
  - Sign bit \(s\) determines whether number is negative or positive
  - Mantissa \(m\) normally a fractional value in range \([1.0,2.0)\).
  - Exponent \(E\) weights value by power of two

Encoding

- MSB is sign bit
- Exp field encodes \(E\)
- Significand field encodes \(m\)

Sizes

- Single precision: 8 exp bits, 23 significand bits
  - 32 bits total
- Double precision: 11 exp bits, 52 significand bits
  - 64 bits total
“Normalized” Numeric Values

Condition

- \( \text{exp} \neq 000...0 \) and \( \text{exp} \neq 111...1 \)

Exponent coded as \textit{biased} value

- \( E = \text{Exp} - \text{Bias} \)
  - \( \text{Exp} \): unsigned value denoted by \texttt{exp}
  - \( \text{Bias} \): Bias value
    - Single precision: 127
    - Double precision: 1023

Mantissa coded with implied leading 1

- \( m = 1.xxx...x_2 \)
  - \( xxx...x \): bits of significand
  - Minimum when \( 000...0 \) (\( m = 1.0 \))
  - Maximum when \( 111...1 \) (\( m = 2.0 - \epsilon \))
  - Get extra leading bit for “free”
Normalized Encoding Example

Value

\[ \text{Float } F = 15740.0; \]
\[ 15740_{10} = 11110101111100_2 = 1.1101101101101_2 \times 2^{13} \]

Significand

\[ m = \underbrace{1.1101101101101}_2 \]
\[ \text{sig} = 11011011011010000000000000_2 \]

Exponent

\[ E = 13 \]
\[ \text{Bias} = 127 \]
\[ \text{Exp} = 140 = 10001100_2 \]

Floating Point Representation of 15740.0:

<table>
<thead>
<tr>
<th>Hex:</th>
<th>4</th>
<th>6</th>
<th>7</th>
<th>5</th>
<th>f</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary:</td>
<td>0100</td>
<td>0110</td>
<td>0111</td>
<td>0101</td>
<td>1111</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>140:</td>
<td>100</td>
<td>0110</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15740:</td>
<td>1111</td>
<td>0101</td>
<td>1111</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Denormalized Values

Condition
  • $\text{exp} = 000...0$

Value
  • Exponent value $E = -\text{Bias} + 1$
  • Mantissa value $m = \,0.\,xxx\ldots x_2$
    - $xxx\ldots x$: bits of significand

Cases
  • $\text{exp} = 000\ldots0$, $\text{significand} = 000\ldots0$
    - Represents value 0
    - Note that have distinct values $+0$ and $-0$
  • $\text{exp} = 000\ldots0$, $\text{significand} \neq 000\ldots0$
    - Numbers very close to 0.0
    - Lose precision as get smaller
    - “Gradual underflow”
# Interesting Numbers

<table>
<thead>
<tr>
<th>Description</th>
<th>Exp</th>
<th>Significand</th>
<th>Numeric Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>00…00</td>
<td>00…00</td>
<td>0.0</td>
</tr>
<tr>
<td>Smallest Pos. Denorm.</td>
<td>00…00</td>
<td>00…01</td>
<td>$2^{-{23,52}} \times 2^{-{126,1022}}$</td>
</tr>
<tr>
<td>• Single</td>
<td></td>
<td></td>
<td>$\approx 1.4 \times 10^{-45}$</td>
</tr>
<tr>
<td>• Double</td>
<td></td>
<td></td>
<td>$\approx 4.9 \times 10^{-324}$</td>
</tr>
<tr>
<td>Largest Denormalized</td>
<td>00…00</td>
<td>11…11</td>
<td>$(1.0 - \varepsilon) \times 2^{-{126,1022}}$</td>
</tr>
<tr>
<td>• Single</td>
<td></td>
<td></td>
<td>$\approx 1.18 \times 10^{-38}$</td>
</tr>
<tr>
<td>• Double</td>
<td></td>
<td></td>
<td>$\approx 2.2 \times 10^{-308}$</td>
</tr>
<tr>
<td>Smallest Pos. Normalized</td>
<td>00…01</td>
<td>00…00</td>
<td>$1.0 \times 2^{-{126,1022}}$</td>
</tr>
<tr>
<td>• Just larger than largest denormalized</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>One</td>
<td>01…11</td>
<td>00…00</td>
<td>1.0</td>
</tr>
<tr>
<td>Largest Normalized</td>
<td>11…10</td>
<td>11…11</td>
<td>$(2.0 - \varepsilon) \times 2^{127,1023}$</td>
</tr>
<tr>
<td>• Single</td>
<td></td>
<td></td>
<td>$\approx 3.4 \times 10^{38}$</td>
</tr>
<tr>
<td>• Double</td>
<td></td>
<td></td>
<td>$\approx 1.8 \times 10^{308}$</td>
</tr>
</tbody>
</table>
Memory Referencing Bug Example

Demonstration of corruption by out-of-bounds array reference

```c
main ()
{
    long int a[2];
    double d = 3.14;
    a[2] = 1073741824; /* Out of bounds reference */
    printf("d = %.15g\n", d);
    exit(0);
}
```

<table>
<thead>
<tr>
<th>Alpha</th>
<th>MIPS</th>
<th>Sun</th>
</tr>
</thead>
<tbody>
<tr>
<td>-g</td>
<td>5.30498947741318e-315</td>
<td>3.13999998664856</td>
</tr>
<tr>
<td>-O</td>
<td>3.14</td>
<td>3.14</td>
</tr>
</tbody>
</table>
Referencing Bug on Alpha

Alpha Stack Frame (-g)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td></td>
</tr>
<tr>
<td>a[1]</td>
<td></td>
</tr>
</tbody>
</table>
| a[0]| long int a[2];
     | double d = 3.14;
     | a[2] = 1073741824;

Optimized Code

- Double d stored in register
- Unaffected by errant write

Alpha -g

- 1073741824 = 0x40000000 = 2^{30}
- Overwrites all 8 bytes with value 0x00000000040000000
- Denormalized value 2^{30} X (smallest denorm 2^{-1074}) = 2^{-1044}
- \approx 5.305 \times 10^{-315}
Referencing Bug on MIPS

MIPS Stack Frame (-g)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>d</td>
</tr>
<tr>
<td>a[1]</td>
<td></td>
</tr>
<tr>
<td>a[0]</td>
<td></td>
</tr>
</tbody>
</table>

long int a[2];
double d = 3.14;
a[2] = 1073741824;

MIPS -g

- Overwrites lower 4 bytes with value 0x40000000
- Original value 3.14 represented as 0x40091eb851eb851f
- Modified value represented as 0x40091eb840000000
- Exp = 1024  E = 1024−1023 = 1
- Mantissa difference: .0000011eb851f_{16}
- Integer value: 11eb851f_{16} = 300,647,711_{10}
- Difference = 2^1 X 2^{-52} X 300,647,711 \approx 1.34 \times 10^{-7}
- Compare to 3.140000000 − 3.139999866 = 0.000000134
Special Values

Condition

• \( \exp = 111\ldots1 \)

Cases

• \( \exp = 111\ldots1, \text{significand} = 000\ldots0 \)
  – Represents value \( \infty \) (infinity)
  – Operation that overflows
  – Both positive and negative
  – E.g., \( 1.0/0.0 = -1.0/-0.0 = +\infty, \ 1.0/-0.0 = -\infty \)

• \( \exp = 111\ldots1, \text{significand} \neq 000\ldots0 \)
  – Not-a-Number (NaN)
  – Represents case when no numeric value can be determined
  – E.g., \( \sqrt{-1}, \infty - \infty \)
  – No fixed meaning assigned to significand bits
Special Properties of Encoding

FP Zero Same as Integer Zero
  • All bits = 0

Can (Almost) Use Unsigned Integer Comparison
  • Must first compare sign bits
  • NaNs problematic
    – Will be greater than any other values
    – What should comparison yield?
  • Otherwise OK
    – Denorm vs. normalized
    – Normalized vs. infinity
Floating Point Operations

Conceptual View

- First compute exact result
- Make it fit into desired precision
  - Possibly overflow if exponent too large
  - Possibly round to fit into significand

Rounding Modes (illustrate with $ rounding)

<table>
<thead>
<tr>
<th>Rounding Mode</th>
<th>$1.40</th>
<th>$1.60</th>
<th>$1.50</th>
<th>$2.50</th>
<th>−$1.50</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>$1.00</td>
<td>$2.00</td>
<td>$1.00</td>
<td>$2.00</td>
<td>−$1.00</td>
</tr>
<tr>
<td>−∞</td>
<td>$1.00</td>
<td>$2.00</td>
<td>$1.00</td>
<td>$2.00</td>
<td>−$2.00</td>
</tr>
<tr>
<td>+∞</td>
<td>$1.00</td>
<td>$2.00</td>
<td>$2.00</td>
<td>$3.00</td>
<td>−$1.00</td>
</tr>
<tr>
<td>Nearest Even (default)</td>
<td>$1.00</td>
<td>$2.00</td>
<td>$2.00</td>
<td>$2.00</td>
<td>−$2.00</td>
</tr>
</tbody>
</table>
A Closer Look at Round-To-Even

Default Rounding Mode

• Hard to get any other kind without dropping into assembly
• All others are statistically biased
  – Sum of set of positive numbers will consistently be over- or under-estimated

Applying to Other Decimal Places

• When exactly halfway between two possible values
  – Round so that least significant digit is even
• E.g., round to nearest hundredth
  
  1.2349999 1.23  (Less than half way)
  1.2350001 1.24  (Greater than half way)
  1.2350000 1.24  (Half way—round up)
  1.2450000 1.24  (Half way—round down)
# Rounding Binary Numbers

## Binary Fractional Numbers
- “Even” when least significant bit is 0
- Half way when bits to right of rounding position = $100..._2$

## Examples
- Round to nearest 1/4 (2 bits right of binary point)

<table>
<thead>
<tr>
<th>Value</th>
<th>Binary</th>
<th>Rounded</th>
<th>Action</th>
<th>Rounded Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-3/32</td>
<td>10.00011₂</td>
<td>10.00₂</td>
<td>(&lt;1/2—down)</td>
<td>2</td>
</tr>
<tr>
<td>2-3/16</td>
<td>10.00110₂</td>
<td>10.01₂</td>
<td>(&gt;1/2—up)</td>
<td>2-1/4</td>
</tr>
<tr>
<td>2-7/8</td>
<td>10.11100₂</td>
<td>11.00₂</td>
<td>(1/2—up)</td>
<td>3</td>
</tr>
<tr>
<td>2-5/8</td>
<td>10.10100₂</td>
<td>10.10₂</td>
<td>(1/2—down)</td>
<td>2-1/2</td>
</tr>
</tbody>
</table>
FP Multiplication

Operands

$(-1)^{s_1} m_1 \ 2^{E_1}$

$(-1)^{s_2} m_2 \ 2^{E_2}$

Exact Result

$(-1)^s m \ 2^E$

- **Sign** $s$: $s_1 \land s_2$
- **Mantissa** $m$: $m_1 \times m_2$
- **Exponent** $E$: $E_1 + E_2$

Fixing

- Overflow if $E$ out of range
- Round $m$ to fit significand precision

Implementation

- Biggest chore is multiplying mantissas
FP Addition

Operand

\[(\neg1)^{s1} m1 \ 2^{E1}\]
\[(\neg1)^{s2} m2 \ 2^{E2}\]

- Assume \(E1 > E2\)

Exact Result

\[(-1)^s m \ 2^E\]

- Sign \(s\), mantissa \(m\):  
  - Result of signed align & add
- Exponent \(E\): \(E1 - E2\)

Fixing

- Shift \(m\) right, increment \(E\) if \(m \geq 2\)
- Shift \(m\) left \(k\) positions, decrement \(E\) by \(k\) if \(m < 1\)
- Overflow if \(E\) out of range
- Round \(m\) to fit significand precision
Mathematical Properties of FP Add

Compare to those of Abelian Group

- Closed under addition? YES
  - But may generate infinity or NaN
- Commutative? YES
- Associative? NO
  - Overflow and inexactness of rounding
- 0 is additive identity? YES
- Every element has additive inverse ALMOST
  - Except for infinities & NaNs

Montonicity

- $a \leq b \Rightarrow a+c \leq b+c$? ALMOST
  - Except for infinities & NaNs
Algebraic Properties of FP Mult

Compare to Commutative Ring

• Closed under multiplication? YES
  – But may generate infinity or NaN

• Multiplication Commutative? YES

• Multiplication is Associative? NO
  – Possibility of overflow, inexactness of rounding

• 1 is multiplicative identity? YES

• Multiplication distributes over addition? NO
  – Possibility of overflow, inexactness of rounding

Montonicity

• \( a \leq b \) & \( c \geq 0 \) \Rightarrow a \times c \leq b \times c \) ? ALMOST
  – Except for infinities & NaNs
Floating Point in C

C Supports Two Levels

- float single precision
- double double precision

Conversions

- Casting between int, float, and double changes numeric values
- Double or float to int
  - Truncates fractional part
  - Like rounding toward zero
  - Not defined when out of range
    » Generally saturates to Tmin or Tmax
- int to double
  - Exact conversion, as long as int has \( \leq 54 \) bit word size
- int to float
  - Will round according to rounding mode
Answers to Floating Point Puzzles

int x = ...;
float f = ...;
double d = ...;

• \( x == (\text{int})(\text{float}) \ x \)
  No: 24 bit mantissa

• \( x == (\text{int})(\text{double}) \ x \)
  Yes: 53 bit mantissa

• \( f == (\text{float})(\text{double}) \ f \)
  Yes: increases precision

• \( d == (\text{float}) \ d \)
  No: looses precision

• \( f == -(-f); \)
  Yes: Just change sign bit

• \( 2/3 == 2/3.0 \)
  No: 2/3 == 1

• \( d < 0.0 \Rightarrow ((d*2) < 0.0) \)
  Yes!

• \( d > f \Rightarrow -f < -d \)
  Yes!

• \( d * d >= 0.0 \)
  Yes!

• \( (d+f)-d == f \)
  No: Not associative

Assume neither \( d \) nor \( f \) is NAN
Alpha Floating Point

Implemented as Separate Unit
- Hardware to add, multiply, and divide
- Floating point data registers
- Various control & status registers

Floating Point Formats
- S_Floating (C float): 32 bits
- T_Floating (C double): 64 bits

Floating Point Data Registers
- 32 registers, each 8 bytes
- Labeled $f0$ to $f31$
- $f31$ is always 0.0

Return Values
- Procedure arguments
  - $f0$ to $f1$, $f2$ to $f3$, $f4$ to $f5$
  - $f6$ to $f7$, $f8$ to $f9$
  - $f10$ to $f11$, $f12$ to $f13$
  - $f14$ to $f15$, $f16$ to $f17$
  - $f18$ to $f19$, $f20$ to $f21$
  - $f22$ to $f23$, $f24$ to $f25$
  - $f26$ to $f27$, $f28$ to $f29$
  - $f30$
  - $f31$

Callee Save Temporaries:
- Always 0.0

Caller Save Temporaries:
- Procedure arguments
- Always 0.0

Callee Save Temporaries:
Floating Point Code Example

Compute Inner Product of Two Vectors

- Single precision arithmetic

```c
float inner_prodF
    (float x[], float y[],
     int n)
{
    int i;
    float result = 0.0;
    for (i = 0; i < n; i++) {
        result += x[i] * y[i];
    }
    return result;
}
```

```assembly
    cpys $f31,$f31,$f0 # result = 0.0
    bis $31,$31,$3   # i = 0
    cmplt $31,$18,$1  # 0 < n?
    beq $1,$102    # if not, skip loop
    .align 5
    $104:
    s4addq $3,0,$1  # $1 = 4 * i
    addq $1,$16,$2  # $2 = &x[i]
    addq $1,$17,$1  # $1 = &y[i]
    lds $f1,0($2)  # $f1 = x[i]
    lds $f10,0($1)  # $f10 = y[i]
    muls $f1,$f10,$f1  # $f1 = x[i] * y[i]
    adds $f0,$f1,$f0  # result += $f1
    addl $3,1,$3   # i++
    cmplt $3,$18,$1  # i < n?
    bne $1,$104    # if so, loop
    $102:
    ret $31,($26),1  # return
```
Numeric Format Conversion

Between Floating Point and Integer Formats

- Special conversion instructions \texttt{cvttq}, \texttt{cvtqt}, \texttt{cvtts}, \texttt{cvtst}, ...
- Convert source operand in one format to destination in other
- Both source & destination must be FP register
  - Transfer to and from GP registers via memory store/load

\begin{center}
\begin{tabular}{|l|l|}
\hline
C Code & Conversion Code \\
\hline
\texttt{float double2float(double d)} & \texttt{cvtts \$f16,\$f0} \\
\hspace{1cm} \{ \hspace{1cm} & \hspace{1cm} \texttt{[Convert T\_Floating to S\_Floating]} \}
\texttt{return (float) d;} & \texttt{stq \$16,0(\$30)} \\
& \texttt{ldt \$f1,0(\$30)} \\
& \texttt{cvtqt \$f1,\$f0} \\
\texttt{\} & \texttt{[Pass through stack and convert]} \\
\hline
double long2double(long i) & \}
\texttt{\{ \hspace{1cm} \{ return (double) i; } \\
& \}
\texttt{\} } \\
\hline
\end{tabular}
\end{center}
Getting FP Bit Pattern

double bit2double(long i)
{
    union {
        long i;
        double d;
    } arg;
    arg.i = i;
    return arg.d;
}

double long2double(long i)
{
    return (double) i;
}

- Union provides direct access to bit representation of double
- bit2double generates double with given bit pattern
  - NOT the same as (double) i
  - Bypasses rounding step
## Alpha 21164 Arithmetic Performance

### Integer

<table>
<thead>
<tr>
<th>Operation</th>
<th>Latency</th>
<th>Issue Rate</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>1</td>
<td>2 / cycle</td>
<td>Two integer pipes</td>
</tr>
<tr>
<td>LW Multiply</td>
<td>8</td>
<td>1 / 8 cycles</td>
<td>Unpipelined</td>
</tr>
<tr>
<td>QW Multiply</td>
<td>16</td>
<td>1 / 16 cycles</td>
<td>Unpipelined</td>
</tr>
<tr>
<td>Divide</td>
<td>∞</td>
<td>0 / cycle</td>
<td>Not implemented</td>
</tr>
</tbody>
</table>

### Floating Point

<table>
<thead>
<tr>
<th>Operation</th>
<th>Latency</th>
<th>Issue Rate</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>4</td>
<td>1 / cycle</td>
<td>Fully pipelined</td>
</tr>
<tr>
<td>Multiply</td>
<td>4</td>
<td>1 / cycle</td>
<td>Fully pipelined</td>
</tr>
<tr>
<td>SP Divide</td>
<td>10</td>
<td>1 / 10 cycle</td>
<td>Unpipelined</td>
</tr>
<tr>
<td>DP Divide</td>
<td>23</td>
<td>1 / 23 cycle</td>
<td>Unpipelined</td>
</tr>
</tbody>
</table>