Integer Arithmetic and Floating Point

Kai Shen

1

Unsigned Addition

- Operands: w bits
  - u + v
- True sum: w+1 bits
  - UAdd\textsubscript{u}(u, v)
- Discard carry: w bits

- Semantics: standard addition, but ignore overflowed carry
  - Still commutative and associative
  - Implements modular arithmetic

\[
UAdd\textsubscript{u}(u, v) = \begin{cases} \ & u + v & u + v < 2^w \\ \ & u + v - 2^w & u + v \geq 2^w \end{cases}
\]

2

Two’s Complement Addition

- Operands: w bits
  - u + v
- True sum: w+1 bits
  - TAdd\textsubscript{w}(u, v)
- Discard carry: w bits

- TAdd and UAdd have identical bit-level computation
  - Signed vs. unsigned addition in C:
    \[
    \text{int } s, t, u, v; \\
    s = \text{(int)} ((\text{unsigned}) u + (\text{unsigned}) v); \\
    t = u + v
    \]
  - Will give s == t
### TAdd Overflow

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

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

- **Formula**
  \[
  TAdd_w(u,v) = \begin{cases} 
  u + v + 2^w & u + v < TMin_w \\
  u + v & TMin_u \leq u + v \leq TMax_u \\
  u + v - 2^w & TMax_u < u + v \leq TMin_v
  \end{cases}
  \]

### Negation: Complement & Increment

- **Semantics of negation (\(-x\))**
  - Only meaningful for signed integer
  - \( TMIN \) is smallest negative integer, what is \(-TMIN\)?
- **Claim** for 2's complement
  \(-x + 1 = -x\)
- **Complement**
  - Observation: \(-x + x = 1111...111 = -1\)
  - \( x + \sim x = 01100010\)
  - \(-x = 10011111\)

### Unsigned Multiplication

- **Operands**: \( w \) bits
- **Semantics**: standard multiplication, but ignore high order \( w \) bits
- **Implements**: modular arithmetic
  \[\text{UMult}_w(u,v) = u \cdot v \mod 2^w\]

### Signed Multiplication

- **Under 2's complement**, same bit-level computation as in unsigned case
### Code Security Example

```c
void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size) {
    /* Allocate buffer for ele_cnt objects, each of ele_size bytes */
    void *result = malloc(ele_cnt * ele_size);
    if (result == NULL) {
        /* malloc failed */
        return NULL;
    }
    void *next = result;
    int i;
    for (i = 0; i < ele_cnt; i++) {
        /* Copy object i to destination */
        memcpy(next, ele_src[i], ele_size);
        /* Move pointer to next memory region */
        next += ele_size;
    }
    return result;
}
```

### Power-of-2 Multiply with Shift

- **Multiply** is slow on most machines
- **Operation**
  - \( u \ll k \) gives \( u \times 2^k \)
  - Both signed and unsigned
    - Operands: \( w \) bits
      - True product: \( w + k \) bits
      - Discard \( k \) bits: \( w \) bits
- **Examples**
  - \( u \ll 3 = u \times 8 \)
  - \( u \ll 5 - u \ll 3 = u \times 24 \)

### Compiled Multiplication Code

**C Function**

```c
int mull2(int x) {
    return x*12;
}
```

**Compiled Arithmetic Operations**

- `leal (%eax, %eax, 2), %eax`
- `sall 02, %eax`
-`t = x+x*2
- `return t << 2;`

- C compiler automatically generates shift/add code when multiplying by constant

### Division

- **Integer division**: divide one integer over another, output an integer
- **Semantics**:
  - Like standard division
  - No overflow problem (except divide by zero)
  - Round toward zero (round down on positive side, round up on negative side)
- **Implementation** for signed/unsigned division is very different
- Division is slower than multiply, so converting to shift etc. will help even more
Unsigned Power-of-2 Divide with Shift

- Quotient of unsigned by power of 2
  - \( u >> k \) gives \( \lfloor \frac{u}{2^k} \rfloor \)
  - Uses logical shift

<table>
<thead>
<tr>
<th>Dividend</th>
<th>Divisor</th>
<th>Quotient</th>
</tr>
</thead>
<tbody>
<tr>
<td>15213</td>
<td>15213</td>
<td>1</td>
</tr>
<tr>
<td>7606.5</td>
<td>7606</td>
<td>1D B6</td>
</tr>
<tr>
<td>950.8125</td>
<td>950</td>
<td>03 B6</td>
</tr>
<tr>
<td>59.4257813</td>
<td>59</td>
<td>00 3B</td>
</tr>
</tbody>
</table>

Signed Power-of-2 Divide with Shift

- Quotient of signed by power of 2
  - \( x >> k \) gives \( \lfloor \frac{x}{2^k} \rfloor \)
  - Uses arithmetic shift
  - Rounds wrong direction when \( u < 0 \)

<table>
<thead>
<tr>
<th>Dividend</th>
<th>Divisor</th>
<th>Quotient</th>
</tr>
</thead>
<tbody>
<tr>
<td>-15213</td>
<td>-15213</td>
<td>C4 93</td>
</tr>
<tr>
<td>-7606.5</td>
<td>-7607</td>
<td>B2 49</td>
</tr>
<tr>
<td>-950.8125</td>
<td>-951</td>
<td>FC 49</td>
</tr>
<tr>
<td>-59.4257813</td>
<td>-60</td>
<td>FF C4</td>
</tr>
</tbody>
</table>

Correct Power-of-2 Divide

- Quotient of negative number by power of 2
  - Want \( \lceil \frac{x}{2^k} \rceil \) (Round Toward 0)
  - Compute as \( \lceil \frac{x}{2^k} \rceil \)
    - In C: \( (x + (1 << k) - 1) / 2^k \)

<table>
<thead>
<tr>
<th>Dividend</th>
<th>Divisor</th>
<th>Quotient</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x</td>
<td>foo()</td>
<td>int y</td>
</tr>
<tr>
<td>unsigned x</td>
<td>= x;</td>
<td>unsigned y</td>
</tr>
</tbody>
</table>

Initialization

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

- \( ux >= 0 \)
- \( ux > -1 \)
- \( x > 0 \&\& y > 0 \Rightarrow x + y > 0 \)
- \( ux >> 3 == ux/8 \)
- \( x >> 3 == x/8 \)
**Floating Point**
- Background: fractional binary numbers
- Bit representation: IEEE floating point standard
- Rounding, addition, multiplication

**Fractional Binary Numbers**
- What is 1011.1012?
- Representation
  - Bits to right of "binary point" represent fractional powers of 2
  - Represents rational number: \( \sum_{j} b_j \times 2^j \)

**Fractional Binary Numbers: Examples**
- **Value** | **Representation**
  - 5 & 3/4 | 101.112
  - 2 & 7/8 | 10.1112
  - 1 & 7/16 | 1.01112

**Representable Numbers**
- Limitation
  - Can only exactly represent numbers of the form \( x/2^k \)
  - Other rational numbers have repeating bit representations
- **Value** | **Representation**
  - 1/3 | 0.0101010101[01]…2
  - 1/5 | 0.001100110011[0011]…2
  - 1/10 | 0.000110011001[0011]…2

**Observations**
- Divide by 2 by shifting right
- Multiply by 2 by shifting left
- Numbers of form 0.111111...2 are just below 1.0
  - \( 1/2 + 1/4 + 1/8 + ... + 1/2^j + ... \rightarrow 1.0 \)
  - Use notation 1.0 – ε
Bits Representation of Fractional Binary Numbers

- How?
  - Think about the limited number of bits we have

IEEE Floating Point

- IEEE Standard 754
  - Established in 1985 as uniform standard for floating point representation
  - 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 exp frac
  - 1 8-bits 23-bits

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

- Extended precision: 80 bits (Intel only)
  - s exp frac
  - 1 15-bits 63 or 64-bits
Normalized 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 \( \text{exp} \)
  - \( \text{Bias} = 2^{k-1} - 1 \), where \( k \) is number of exponent bits
    - Single precision: 127 (Exp: 1...254, E: -126...127)
    - Double precision: 1023 (Exp: 1...2046, E: -1022...1023)
- Significand coded with implied leading 1: \( M = 1 \times \text{x...x} \)
  - Minimum when 000...0 (\( M = 1.0 \))
  - Maximum when 111...1 (\( M = 2.0 – \varepsilon \))
  - Get extra leading bit for "free"

Normalized Encoding Example

- \( \text{float F} = 15213.0; \)
  - 1521310 = 111011011011012 = 1 1101101101101 x 2^{13}
  - Significand \( M = 1.1101101101101 \)
  - Fraction \( \text{frac} = 110110110110100000000002 \)
  - Exponent \( E = 13 \)
  - Bias = 127
  - \( \text{Exp} = 140 = 1001100, \)
- Result:
  \[ \begin{array}{cccc}
  \text{s} & \text{exp} & \text{frac} \\
  \hline
  0 & 10001100 & 11011011011010000000000 \\
  \end{array} \]

Denormalized Values

- Condition: \( \text{exp} = 000…0 \)
- Significand coded with implied leading 0: \( M = 0.x...x \)
  - Equispaced leading to 0
- Exponent value: \( E = 1 – \text{Bias} \) (instead of \( E = 0 – \text{Bias} \))
  - Smooth transition between largest value of \( \text{exp} = 00...00 \) and smallest value of \( \text{exp} = 00...01 \)
- Special case: \( \text{exp} = 000…0, \text{frac} = 000…0 \)
  - Represents zero value (distinct +0 and –0)

Special Values

- Condition: \( \text{exp} = 111...1 \)
- Case: \( \text{exp} = 111...1, \text{frac} = 000...0 \)
  - 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
- Case: \( \text{exp} = 111...1, \text{frac} \neq 000...0 \)
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., sqrt(-1), \( \infty \cdot \infty, \infty \times 0 \)
Tiny Floating Point Example

- 8-bit floating point representation
  - the sign bit is in the most significant bit
  - the next four bits are the exponent, with a bias of 7
  - the last three bits are the frac
- Same general form as IEEE Format
  - normalized, denormalized
  - representation of 0, NaN, infinity

Dynamic Range (Positive Only)

<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
<th>E</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>000</td>
<td>-6</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>001</td>
<td>-6</td>
<td>1/8*1/64 = 1/512</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>010</td>
<td>-6</td>
<td>2/8*1/64 = 2/512</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>110</td>
<td>-6</td>
<td>6/8*1/64 = 6/512</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>111</td>
<td>-6</td>
<td>7/8*1/64 = 7/512</td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td>000</td>
<td>-6</td>
<td>8/8*1/64 = 8/512</td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td>001</td>
<td>-6</td>
<td>9/8*1/64 = 9/512</td>
</tr>
<tr>
<td>0</td>
<td>0010</td>
<td>110</td>
<td>-1</td>
<td>14/8*1/2 = 14/16</td>
</tr>
<tr>
<td>0</td>
<td>0010</td>
<td>111</td>
<td>-1</td>
<td>15/8*1/2 = 15/16</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>000</td>
<td>0</td>
<td>8/8*1 = 1</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>001</td>
<td>0</td>
<td>9/8*1 = 9/8</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>010</td>
<td>0</td>
<td>10/8*1 = 10/8</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td>110</td>
<td>7</td>
<td>14/8*128 = 224</td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td>111</td>
<td>7</td>
<td>15/8*128 = 240</td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>000</td>
<td>n/a</td>
<td>inf</td>
</tr>
</tbody>
</table>

Interesting Numbers

- Single, double

<table>
<thead>
<tr>
<th>Description</th>
<th>exp</th>
<th>frac</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) x 2^(-126,1022)</td>
</tr>
<tr>
<td>Single</td>
<td>1.4 x 10^-45</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double</td>
<td>4.9 x 10^-324</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 - e) x 2^(127,1023)</td>
</tr>
<tr>
<td>Single</td>
<td>3.4 x 10^38</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double</td>
<td>1.8 x 10^38</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Special Properties of Encoding

- Floating point zero same as integer zero
  - All bits = 0
- Can (almost) use unsigned integer comparison
  - Must first compare sign bits
  - Must consider ~0 = 0
  - NaNs problematic
    - Will be greater than any other values
    - What should comparison yield?
  - Otherwise OK
    - Denorm vs. normalized
    - Normalized vs. infinity
Floating Point Operations: Basic Idea

- \( x + y = \text{Round}(x + y) \)
- \( x \times y = \text{Round}(x \times y) \)

Basic idea:
- First compute exact result
- Make it fit into desired precision
- Possibly overflow if exponent too large
- Possibly round to fit into \( \text{frac} \)

Rounding

- Rounding modes (illustrate with \$\text{rounding}\$

  - Towards zero
  - Round down \((-\infty)\)
  - Round up \((+\infty)\)
  - Nearest even (default)

Nearest even:
- Round to nearest acceptable point
- When exactly halfway between two adjacent points, round so that least significant digit is even

Why?
- Statistically unbiased, even digit is easier to represent

Floating Point Addition

\[
(-1)^s M_1 \, 2^{E_1} + (-1)^r M_2 \, 2^{E_2}
\]

Assume \( E_1 > E_2 \)

- Exact result: \((-1)^s M \, 2^E\)
  - Sign \( s \), significand \( M \):
  - Result of signed align & add
  - Exponent \( E \):
    - \( E_1 \)

Fixing:
- If \( M \geq 2 \), shift \( M \) right, increment \( E \)
- If \( M < 1 \), shift \( M \) left \( k \) positions, decrement \( E \) by \( k \)
- Overflow if \( E \) out of range
- Round \( M \) to fit \( \text{frac} \) precision

Properties of FP Add

- Commutative?
- Associative?
  - Overflow and inexactness of rounding
    - \((1e20 + -1e20) + 3.14 = 3.14\)
    - \(1e20 + (-1e20 + 3.14) = ??\)
- Monotonicity: \( a \geq b \Rightarrow a+c \geq b+c \)?
  - Except for infinities & NaNs
Floating Point Multiplication

- \((-1)^{s_1} M_1 2^{E_1} \times (-1)^{s_2} M_2 2^{E_2}\)
- Exact result: \((-1)^s M 2^E\)
  - Sign \(s\): \(s_1 \land s_2\)
  - Significand \(M\): \(M_1 \times M_2\)
  - Exponent \(E\): \(E_1 + E_2\)

Fixing
- If \(M \geq 2\), shift \(M\) right, increment \(E\)
- If \(E\) out of range, overflow
- Round \(M\) to fit \texttt{frac} precision

Implementation
- Biggest chore is multiplying significands

Mathematical Properties of FP Mult

- Multiplication is commutative?
- Multiplication is associative?
  - Possibility of overflow, inexactness of rounding
- Monotonicity: \(a \geq b \land c \geq 0 \Rightarrow a \times c \geq b \times c\)
  - Except for infinities & NaNs

Floating Point in C

- C guarantees two levels
  - \texttt{float} single precision
  - \texttt{double} double precision
- Conversions/casting
  - Casting between \texttt{int}, \texttt{float}, and \texttt{double} changes bit representation
  - \texttt{double/float \to int}
    - Truncates fractional part
    - Like rounding toward zero
    - Not defined when out of range or NaN: Generally sets to TMIn
  - \texttt{int \to double}
    - Exact conversion, as long as \texttt{int} has \(\leq 53\) bit word size
  - \texttt{int \to float}
    - Will round according to rounding mode

Disclaimer

These slides were adapted from the CMU course slides provided along with the textbook of “Computer Systems: A programmer’s Perspective” by Bryant and O’Hallaron. The slides are intended for the sole purpose of teaching the computer organization course at the University of Rochester.