CS 33

Data Representation (Part 2)
Unsigned Power-of-2 Divide with Shift

- Quotient of unsigned by power of 2
  - $u \gg k$ gives $\lfloor u / 2^k \rfloor$
  - uses logical shift

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>15213</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>$x \gg 1$</td>
<td>7606.5</td>
<td>1D B6</td>
<td>00011101 10110110</td>
</tr>
<tr>
<td>$x \gg 4$</td>
<td>950.8125</td>
<td>03 B6</td>
<td>00000011 10110110</td>
</tr>
<tr>
<td>$x \gg 8$</td>
<td>59.4257813</td>
<td>00 3B</td>
<td>00000000 00111011</td>
</tr>
</tbody>
</table>
Signed Power-of-2 Divide with Shift

- Quotient of signed by power of 2
  - \( x \gg k \) gives \( \lfloor x / 2^k \rfloor \)
  - uses arithmetic shift
  - rounds wrong direction when \( x < 0 \)

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y )</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>( y \gg 1)</td>
<td>-7606.5</td>
<td>E2 49</td>
<td>11100010 01001001</td>
</tr>
<tr>
<td>( y \gg 4)</td>
<td>-950.8125</td>
<td>FC 49</td>
<td>11111100 01001001</td>
</tr>
<tr>
<td>( y \gg 8)</td>
<td>-59.4257813</td>
<td>FF C4</td>
<td>11111111 11000100</td>
</tr>
</tbody>
</table>
Correct Power-of-2 Divide

- Quotient of negative number by power of 2
  - want $\left\lfloor \frac{x}{2^k} \right\rfloor$ (round toward 0)
  - compute as $\left\lfloor \frac{x+2^k-1}{2^k} \right\rfloor$
    » in C: $(x + (1<<k) - 1) >> k$
    » biases dividend toward 0

**Case 1: no rounding**

<table>
<thead>
<tr>
<th>dividend:</th>
<th>divisor:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u$</td>
<td>$2^k$</td>
</tr>
<tr>
<td>$u / 2^k$</td>
<td>$\left\lfloor \frac{x}{2^k} \right\rfloor$</td>
</tr>
</tbody>
</table>

Biasing has no effect
Correct Power-of-2 Divide (Cont.)

Case 2: rounding

<table>
<thead>
<tr>
<th>Dividend:</th>
<th>( x \times 2^k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( +2^k - 1 )</td>
<td>( 0 \cdots 001 \cdots 11 )</td>
</tr>
</tbody>
</table>

\[ \frac{x}{2^k} \]

<table>
<thead>
<tr>
<th>Divisor:</th>
<th>( \frac{x}{2^k} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \left\lfloor \frac{x}{2^k} \right\rfloor )</td>
<td>( 1 \cdots 111 \cdots )</td>
</tr>
</tbody>
</table>

Biases adds 1 to final result
Why Should I Use Unsigned?

• Don’t use just because number nonnegative
  – easy to make mistakes
    
    unsigned i;
    for (i = cnt-2; i >= 0; i--)
    a[i] += a[i+1];
  – can be very subtle
    
    #define DELTA sizeof(int)
    int i;
    for (i = CNT; i-DELTA >= 0; i-= DELTA)
      ...

• Do use when using bits to represent sets
  – logical right shift, no sign extension
Combining Bytes

• Data items of multiple sizes, usually powers of two
  – one-byte, two-byte, four-byte, eight-byte integers
  – four-byte and eight-byte floating-point numbers
• For example: four consecutive bytes interpreted as storing an integer (or a float)
  – for best performance, address of lowest byte should be a multiple of the size of the item (four in this case)
Word Size

• (Mostly) obsolete term
  – old computers had items of one size: the word size

• Now used to express the number of bits necessary to hold an address
  – 16 bits (really old computers)
  – 32 bits (old computers)
  – 64 bits (most current computers)
Byte Ordering

- Four-byte integer
  - 0x76543210
- Stored at location 0x100
  - which byte is at 0x100?
  - which byte is at 0x103?

<table>
<thead>
<tr>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>32</td>
<td>54</td>
<td>76</td>
</tr>
<tr>
<td>76</td>
<td>54</td>
<td>32</td>
<td>10</td>
</tr>
</tbody>
</table>

Little-endian

Big-endian
Byte Ordering (2)

Big Endian

Little Endian

00 00 00 01
Quiz 1

```c
int main() {
    long x=1;
    func((int *)&x);
    return 0;
}

void func(int *arg) {
    printf("%d\n", *arg);
}
```

What value is printed on a big-endian 64-bit computer?

- a) 0
- b) 1
- c) $2^{32}$
- d) $2^{32}-1$
Which Byte Ordering Do We Use?

```c
int main() {
    unsigned int x = 0x03020100;
    unsigned char *xarray = (unsigned char *)&x;
    for (int i=0; i<4; i++) {
        printf("%02x", xarray[i]);
    }
    printf("\n");
    return 0;
}
```
Fractional binary numbers

• What is $1011.101_2$?
Fractional Binary Numbers

- Representation
  - bits to right of “binary point” represent fractional powers of 2
  - represents rational number: \[ \sum_{k=-j}^{i} b_k \times 2^k \]
Representable Numbers

• **Limitation #1**
  – can exactly represent only numbers of the form \( n/2^k \)
    » other rational numbers have repeating bit representations
  – value representation
    » 1/3 \( 0.0101010101[01]..._2 \)
    » 1/5 \( 0.001100110011[0011]..._2 \)
    » 1/10 \( 0.0001100110011[0011]..._2 \)

• **Limitation #2**
  – just one setting of decimal point within the \( w \) bits
    » limited range of numbers (very small values? very large?)
IEEE Floating Point

• IEEE Standard 754
  – established in 1985 as uniform standard for floating point arithmetic
    » before that, many idiosyncratic formats
  – supported on 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 \( M \))
Precision options

- **Single precision: 32 bits**
  
  \[
  \begin{array}{ccc}
  s & \text{exp} & \text{frac} \\
  1 & 8\text{-bits} & 23\text{-bits}
  \end{array}
  \]

- **Double precision: 64 bits**
  
  \[
  \begin{array}{ccc}
  s & \text{exp} & \text{frac} \\
  1 & 11\text{-bits} & 52\text{-bits}
  \end{array}
  \]

- **Extended precision: 80 bits (Intel only)**
  
  \[
  \begin{array}{ccc}
  s & \text{exp} & \text{frac} \\
  1 & 15\text{-bits} & 64\text{-bits}
  \end{array}
  \]
“Normalized” Values

• When: \( \exp \neq 000\ldots0 \) and \( \exp \neq 111\ldots1 \)

• Exponent coded as biased value: \( E = \Exp - \Bias \)
  – \( \exp \): unsigned value \( \exp \)
  – \( \bias = 2^{k-1} - 1 \), 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: \( M = 1.\,\text{xxx…x} \)
  – \( \text{xxx…x} \): bits of \( \text{frac} \)
  – minimum when \( \text{frac}=000\ldots0 \) (\( M = 1.0 \))
  – maximum when \( \text{frac}=111\ldots1 \) (\( M = 2.0 - \varepsilon \))
  – get extra leading bit for “free”
Normalized Encoding Example

• **Value:** float \( F = 15213.0; \)
  - \( 15213_{10} = 11101101101101_2 \)
  - \( = 1.1101101101101_2 \times 2^{13} \)

• **Significand**
  \[
  M = 1.1101101101101_2 \\
  frac = 1101101101010000000000000_2
  \]

• **Exponent**
  \[
  E = 13 \\
  bias = 127 \\
  exp = 140 = 10001100_2
  \]

• **Result:**

\[
\begin{array}{cccc}
\text{s} & \text{exp} & \text{frac} \\
0 & 10001100 & 1101101101101000000000000000
\end{array}
\]
Denormalized Values

• Condition: \( \text{exp} = 000\ldots0 \)
• Exponent value: \( E = -\text{Bias} + 1 \) (instead of \( E = 0 - \text{Bias} \))
• Significand coded with implied leading 0:
  \( M = 0.xxx\ldots x_2 \)
  – \( xxx\ldots x \): bits of \( \text{frac} \)
• Cases
  – \( \text{exp} = 000\ldots0, \text{frac} = 000\ldots0 \)
    » represents zero value
    » note distinct values: +0 and −0 (why?)
  – \( \text{exp} = 000\ldots0, \text{frac} \neq 000\ldots0 \)
    » numbers closest to 0.0
    » equispaced
Special Values

• Condition: $\exp = 111\ldots1$

• Case: $\exp = 111\ldots1$, $\frac{\text{a}}{\text{b}} = 000\ldots0$
  – represents value $\infty$ (infinity)
  – operation that overflows
  – both positive and negative
  – e.g., $1.0/0.0 = -1.0/-0.0 = +\infty$, $1.0/-0.0 = -\infty$

• Case: $\exp = 111\ldots1$, $\frac{\text{a}}{\text{b}} \neq 000\ldots0$
  – not-a-number (NaN)
  – represents case when no numeric value can be determined
  – e.g., $\sqrt{-1}$, $\infty - \infty$, $\infty \times 0$
Visualization: Floating-Point Encodings

\[ \begin{align*}
-\infty & \quad -\text{Normalized} & \quad -\text{Denorm} & \quad +\text{Denorm} & \quad +\text{Normalized} & \quad +\infty \\
\text{NaN} & \quad -0 & \quad +0 & \text{NaN} 
\end{align*} \]
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>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000 000</td>
<td>-6</td>
<td>0</td>
<td>0</td>
<td>closest to zero</td>
</tr>
<tr>
<td>0</td>
<td>0000 001</td>
<td>-6</td>
<td>1/8*1/64 = 1/512</td>
<td>largest denorm</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000 010</td>
<td>-6</td>
<td>2/8*1/64 = 2/512</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000 110</td>
<td>-6</td>
<td>6/8*1/64 = 6/512</td>
<td>smallest norm</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000 111</td>
<td>-6</td>
<td>7/8*1/64 = 7/512</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0001 000</td>
<td>-6</td>
<td>8/8*1/64 = 8/512</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0001 001</td>
<td>-6</td>
<td>9/8*1/64 = 9/512</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0110 110</td>
<td>-1</td>
<td>14/8*1/2 = 14/16</td>
<td>closest to 1 below</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0110 111</td>
<td>-1</td>
<td>15/8*1/2 = 15/16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0111 000</td>
<td>0</td>
<td>8/8*1 = 1</td>
<td>closest to 1 above</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0111 001</td>
<td>0</td>
<td>9/8*1 = 9/8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0111 010</td>
<td>0</td>
<td>10/8*1 = 10/8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1110 110</td>
<td>7</td>
<td>14/8*128 = 224</td>
<td>largest norm</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1110 111</td>
<td>7</td>
<td>15/8*128 = 240</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1111 000</td>
<td>n/a</td>
<td>inf</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Denormalized numbers**

**Normalized numbers**
Distribution of Values

- **6-bit IEEE-like format**
  - \( e = 3 \) exponent bits
  - \( f = 2 \) fraction bits
  - bias is \( 2^{3-1}-1 = 3 \)

- Notice how the distribution gets denser toward zero.

---

<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3-bits</td>
<td>2-bits</td>
</tr>
</tbody>
</table>

- Denormalized
- Normalized
- Infinity
Distribution of Values (close-up view)

- 6-bit IEEE-like format
  - $e = 3$ exponent bits
  - $f = 2$ fraction bits
  - bias is 3

![Diagram showing the distribution of values with denormalized, normalized, and infinity markers.](image-url)
Quiz 2

- 6-bit IEEE-like format
  - e = 3 exponent bits
  - f = 2 fraction bits
  - bias is 3

What number is represented by 0 011 10?

- a) 12
- b) 1.5
- c) .5
- d) none of the above
Floating-Point Operations: Basic Idea

• $x +_f y = \text{Round}(x + y)$

• $x \times_f y = \text{Round}(x \times y)$

• Basic idea
  – first compute exact result
  – make it fit into desired precision
    » possibly overflow if exponent too large
    » possibly round to fit into frac
## Rounding

- **Rounding modes (illustrated with $ rounding)**

<table>
<thead>
<tr>
<th></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>towards zero</td>
<td>$1</td>
<td>$1</td>
<td>$1</td>
<td>$2</td>
<td>$–1</td>
</tr>
<tr>
<td>round down (−∞)</td>
<td>$1</td>
<td>$1</td>
<td>$1</td>
<td>$2</td>
<td>$–2</td>
</tr>
<tr>
<td>round up (+∞)</td>
<td>$2</td>
<td>$2</td>
<td>$2</td>
<td>$3</td>
<td>$–1</td>
</tr>
<tr>
<td>nearest integer</td>
<td>$1</td>
<td>$2</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>nearest even (default)</td>
<td>$1</td>
<td>$2</td>
<td>$2</td>
<td>$2</td>
<td>$–2</td>
</tr>
</tbody>
</table>
Creating a Floating Point Number

• **Steps**
  – normalize to have leading 1
  – round to fit within fraction
  – postnormalize to deal with effects of rounding

<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4-bits</td>
<td>3-bits</td>
</tr>
</tbody>
</table>

• **Case study**
  – convert 8-bit unsigned numbers to tiny floating-point format

**example numbers**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>10000000</td>
</tr>
<tr>
<td>13</td>
<td>00001101</td>
</tr>
<tr>
<td>33</td>
<td>00010001</td>
</tr>
<tr>
<td>35</td>
<td>00010011</td>
</tr>
<tr>
<td>138</td>
<td>10001010</td>
</tr>
<tr>
<td>63</td>
<td>00111111</td>
</tr>
</tbody>
</table>
Normalize

- **Requirement**
  - set binary point so that numbers of form 1.xxxxx
  - adjust all to have leading one
    » decrement exponent as shift left

<table>
<thead>
<tr>
<th>Value</th>
<th>Binary</th>
<th>Fraction</th>
<th>Exponent</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>10000000</td>
<td>1.00000000</td>
<td>7</td>
</tr>
<tr>
<td>13</td>
<td>00001101</td>
<td>1.10100000</td>
<td>3</td>
</tr>
<tr>
<td>17</td>
<td>00010001</td>
<td>1.00010000</td>
<td>4</td>
</tr>
<tr>
<td>19</td>
<td>00010011</td>
<td>1.00110000</td>
<td>4</td>
</tr>
<tr>
<td>138</td>
<td>10001010</td>
<td>1.00010010</td>
<td>7</td>
</tr>
<tr>
<td>63</td>
<td>00111111</td>
<td>1.11111000</td>
<td>5</td>
</tr>
</tbody>
</table>
Rounding

1. BBGRXXX

Guard bit: LSB of result
Sticky bit: OR of remaining bits
Round bit: 1st bit removed

• Round-up conditions
  – round = 1, sticky = 1 ⇒ > 0.5
  – guard = 1, round = 1, sticky = 0 ⇒ round up to even

<table>
<thead>
<tr>
<th>Value</th>
<th>Fraction</th>
<th>GRS</th>
<th>Incr?</th>
<th>Rounded</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>1.00000000</td>
<td>000</td>
<td>N</td>
<td>1.000</td>
</tr>
<tr>
<td>13</td>
<td>1.10100000</td>
<td>100</td>
<td>N</td>
<td>1.101</td>
</tr>
<tr>
<td>17</td>
<td>1.00010000</td>
<td>010</td>
<td>N</td>
<td>1.000</td>
</tr>
<tr>
<td>19</td>
<td>1.00110000</td>
<td>110</td>
<td>Y</td>
<td>1.010</td>
</tr>
<tr>
<td>138</td>
<td>1.00010100</td>
<td>011</td>
<td>Y</td>
<td>1.001</td>
</tr>
<tr>
<td>63</td>
<td>1.11111000</td>
<td>111</td>
<td>Y</td>
<td>10.000</td>
</tr>
</tbody>
</table>
Postnormalize

- **Issue**
  - rounding may have caused overflow
  - handle by shifting right once & incrementing exponent

<table>
<thead>
<tr>
<th>Value</th>
<th>Rounded</th>
<th>Exp</th>
<th>Adjusted</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>1.000</td>
<td>7</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>1.101</td>
<td>3</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>1.000</td>
<td>4</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>1.010</td>
<td>4</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>138</td>
<td>1.001</td>
<td>7</td>
<td>134</td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>10.000</td>
<td>5</td>
<td>1.000*2^6</td>
<td>64</td>
</tr>
</tbody>
</table>
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 (or underflow)
  – round M to fit \(\frac{\text{precision}}{\text{frac}}\)

• Implementation
  – biggest chore is multiplying significands
Floating-Point Addition

• $(-1)^{s_1} M_1 \ 2^{E_1} \ + \ (-1)^{s_2} M_2 \ 2^{E_2}$
  – 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 frac precision