CS 33

Data Representation (Part 3)
Byte-Oriented Memory Organization

- Programs refer to data by address
  - conceptually, envision it as a very large array of bytes
    » in reality, it’s not, but can think of it that way
  - an address is like an index into that array
    » pointer variables contain addresses

- Note: system provides private address spaces to each “process”
  - think of a process as a program being executed
  - so, a program can clobber its own data, but not that of others
Machine Words

• Any given computer has a “word size”
  – nominal size of integer-valued data
    » and of addresses
  – until a decade or so ago, most machines used 32 bits (4 bytes) as word size
    » limits addresses to 4GB ($2^{32}$ bytes)
    » became too small for memory-intensive applications
      • leading to emergence of computers with 64-bit word size
  – machines still support multiple data formats
    » fractions or multiples of word size
    » always integral number of bytes
Word-Oriented Memory Organization

- Addresses specify byte locations
  - address of first byte in word
  - addresses of successive words differ by 4 (32-bit) or 8 (64-bit)

<table>
<thead>
<tr>
<th>32-bit Words</th>
<th>64-bit Words</th>
<th>Bytes</th>
<th>Addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr = 0000</td>
<td>Addr = 0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>Addr = 0004</td>
<td></td>
<td>0001</td>
<td>0001</td>
</tr>
<tr>
<td>Addr = 0008</td>
<td></td>
<td>0002</td>
<td>0002</td>
</tr>
<tr>
<td>Addr = 0012</td>
<td></td>
<td>0003</td>
<td>0003</td>
</tr>
<tr>
<td></td>
<td>Addr = 0008</td>
<td>0004</td>
<td>0004</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0005</td>
<td>0005</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0006</td>
<td>0006</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0007</td>
<td>0007</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0008</td>
<td>0008</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0009</td>
<td>0009</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0010</td>
<td>0010</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0011</td>
<td>0011</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0012</td>
<td>0012</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0013</td>
<td>0013</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0014</td>
<td>0014</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0015</td>
<td>0015</td>
</tr>
</tbody>
</table>
Byte Ordering

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

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

Little-endian

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

Big-endian
Byte Ordering (2)

Big Endian

Little Endian
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?

char title[] = "This is where it begins!!!";
int array[] = {0x04030201, 0x05040302,
               0x06050403, 0x07060504};

int main() {
    for (int i=0; i<4; i++) {
        printf("%x\n", array[i]);
    }
    abort();
    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: \( \text{exp} \neq 000\ldots0 \) and \( \text{exp} \neq 111\ldots1 \)

• Exponent coded as 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: \( M = 1.\text{xxx}\ldots\text{x}_2 \)
  – \( \text{xxx}\ldots\text{x} \): 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 $F = 15213.0;$
  – $15213_{10} = 111011011011012$
    = $1.11011011011012 \times 2^{13}$

• Significand
  $M = 1.11011011011012$
  $frac = 1101101101101000000000000_2$

• Exponent
  $E = 13$
  $bias = 127$
  $exp = 140 = 10001100_2$

• Result:

```
0 10001100 110110110110100000000000000
s   exp  frac
```
Denormalized Values

- Condition: \( \text{exp} = 000...0 \)
- Exponent value: \( E = -\text{Bias} + 1 \) (instead of \( E = 0 - \text{Bias} \))
- Significand coded with implied leading 0:
  \( M = 0.xxx...x \)
  - \( xxx...x \): bits of \( \text{frac} \)

Cases
- \( \text{exp} = 000...0, \ \text{frac} = 000...0 \)
  » represents zero value
  » note distinct values: +0 and –0 (why?)
- \( \text{exp} = 000...0, \ \text{frac} \neq 000...0 \)
  » numbers closest to 0.0
  » equispaced
Special Values

• Condition: $exp = 111...1$

• Case: $exp = 111...1$, $frac = 000...0$
  – 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: $exp = 111...1$, $frac \neq 000...0$
  – 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

-∞ - Normalized - Denorm + Denorm + Normalized +∞

NaN -0 +0 NaN
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>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>

**Denormalized numbers**

**Normalized numbers**

- **Closest to zero**: 0
- **Largest denorm**: 7/8*1/64 = 7/512
- **Smallest norm**: 8/8*1/64 = 8/512
- **Closest to 1 below**: 14/8*1/2 = 14/16
- **Closest to 1 above**: 15/8*1/2 = 15/16
- **Largest norm**: 15/8*128 = 240

---

*CS33 Intro to Computer Systems*
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.

![Diagram showing distribution of values with 8 values and labels for denormalized, normalized, and infinity.]
Distribution of Values (close-up view)

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

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

example numbers

<table>
<thead>
<tr>
<th>Number</th>
<th>Binary</th>
<th>Fraction</th>
<th>Exponent</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>100000000</td>
<td>100000000000000</td>
<td>00000000</td>
</tr>
<tr>
<td>13</td>
<td>00001101</td>
<td>0000110100000000</td>
<td>00000000</td>
</tr>
<tr>
<td>33</td>
<td>00010001</td>
<td>0001000100000000</td>
<td>00000000</td>
</tr>
<tr>
<td>35</td>
<td>00010011</td>
<td>0001001100000000</td>
<td>00000000</td>
</tr>
<tr>
<td>138</td>
<td>10001010</td>
<td>1000101000000000</td>
<td>00000000</td>
</tr>
<tr>
<td>63</td>
<td>00111111</td>
<td>0011111100000000</td>
<td>00000000</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>000001101</td>
<td>1.10100000</td>
<td>3</td>
</tr>
<tr>
<td>17</td>
<td>00010001</td>
<td>1.0001000</td>
<td>4</td>
</tr>
<tr>
<td>19</td>
<td>00010011</td>
<td>1.0011000</td>
<td>4</td>
</tr>
<tr>
<td>138</td>
<td>10001010</td>
<td>1.0001010</td>
<td>7</td>
</tr>
<tr>
<td>63</td>
<td>00111111</td>
<td>1.1111100</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 \(\Rightarrow\) > 0.5
  - guard = 1, round = 1, sticky = 0 \(\Rightarrow\) 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>1.0001010</td>
<td>011</td>
<td>Y</td>
<td>1.001</td>
</tr>
<tr>
<td>63</td>
<td>1.1111100</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></td>
<td>128</td>
</tr>
<tr>
<td>13</td>
<td>1.101</td>
<td>3</td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>17</td>
<td>1.000</td>
<td>4</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>19</td>
<td>1.010</td>
<td>4</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>138</td>
<td>1.001</td>
<td>7</td>
<td></td>
<td>134</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: \(s_1 \wedge s_2\)
  – 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 \(\text{frac}\) precision

• 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 \texttt{frac}\ precision
Floating Point in C

• C guarantees two levels
  –float single precision
  –double double precision

• Conversions/casting
  –casting between int, float, and double changes bit representation
  –double/float \rightarrow int
    » truncates fractional part
    » like rounding toward zero
    » not defined when out of range or NaN: generally sets to Tmin
  –int \rightarrow double
    » exact conversion, as long as int has \leq 53-bit word size
  –int \rightarrow float
    » will round according to rounding mode
Quiz 3

Suppose $f$, declared to be a `float`, is assigned the largest possible floating-point positive value (other than $+\infty$). What is the value of $g = f+1.0$?

a) $f$
b) $+\infty$
c) NAN
d) 0
Float is not Rational …

• Floating addition
  – commutative: a +f b = b +f a
    » yes!
  – associative: a +f (b +f c) = (a +f b) +f c
    » no!
    • 2 +f (1e20 +f -1e20) = 2
    • (2 +f 1e20) +f -1e20 = 0
Float is not Rational …

• Multiplication
  – commutative: $a \times f b = b \times f a$
    » yes!
  – associative: $a \times f (b \times f c) = (a \times f b) \times f c$
    » no!
    - $1e20 \times f (1e20 \times f 1e-20) = 1e20$
    - $(1e20 \times f 1e20) \times f 1e-20 = +\infty$
Float is not Rational …

• More …
  – multiplication distributes over addition:
    \[ a \times_f (b +_f c) = (a \times_f b) +_f (a \times_f c) \]
    » no!
    » \(1e20 \times_f (1e20 +_f -1e20) = 0\)
    » \((1e20 \times_f 1e20) +_f (1e20 \times_f -1e20) = \text{NaN}\)
  – loss of significance:
    x=y+1
    z=2/(x-y)
    z==2?
    » not necessarily!
      • consider \(y = 1e20\)