Lecture 25

Floating Point
New-School Machine Structures (It’s a bit more complicated!)

**Software**

- Parallel Requests
  Assigned to computer
  e.g., Search “Katz”
- Parallel Threads
  Assigned to core
  e.g., Lookup, Ads
- Parallel Instructions
  >1 instruction @ one time
  e.g., 5 pipelined instructions
- Parallel Data
  >1 data item @ one time
  e.g., Add of 4 pairs of words
- Hardware descriptions
  All gates @ one time
- Programming Languages

**Hardware**

- Warehouse Scale Computer
- Harness Parallelism & Achieve High Performance
- Core
- Memory (Cache)
- Input/Output
- Instruction Unit(s)
- Functional Unit(s)
- Cache Memory
- Logic Gates

Smart Phone

Warehouse Scale Computer

Computer Science 61C Fall 2016

Friedland and Weaver
New-School Machine Structures
(It’s a bit more complicated!)

- **Parallel Requests**
  Assigned to computer
  e.g., Search “Katz”
- **Parallel Threads**
  Assigned to core
  e.g., Lookup, Ads
- **Parallel Instructions**
  >1 instruction @ one time
  e.g., 5 pipelined instructions
- **Parallel Data**
  >1 data item @ one time
  e.g., Add of 4 pairs of words
- **Hardware descriptions**
  All gates @ one time
- **Programming Languages**

Harness Parallelism & Achieve High Performance

Smart Phone

Warehouse Scale Computer

How do we know?
Review of Numbers

• What can we represent in N bits?

  • \(2^N\) things, and no more! They could be…
Review of Numbers

• What can we represent in N bits?
  • $2^N$ things, and no more! They could be…
  • Unsigned integers:
Review of Numbers

- What can we represent in N bits?
  - $2^N$ things, and no more! They could be…
  - Unsigned integers:
    - 0 to $2^N - 1$
      (for N=32, $2^{32} - 1 = 4,294,967,295$)
  - Signed Integers (Two’s Complement)
Review of Numbers

- What can we represent in N bits?
  - $2^N$ things, and no more! They could be…
  - Unsigned integers:
    
    $0$ to $2^N - 1$

    (for $N=32$, $2^{32}-1 = 4,294,967,295$)
  - Signed Integers (Two’s Complement)
    
    $-2^{(N-1)}$ to $2^{(N-1)} - 1$

    (for $N=32$, $2^{31} = 2,147,483,648$)
What about other numbers?

1. Very large numbers? (seconds/millennium)
   \[ \Rightarrow 31,556,926,000_{\text{ten}} \times 10^{10} \]

2. Very small numbers? (Bohr radius)
   \[ \Rightarrow 0.000000000529177_{\text{ten}} \times 10^{-11} \]

3. Numbers with both integer & fractional parts?
   \[ \Rightarrow 1.5 \]
What about other numbers?

1. Very large numbers? (seconds/millennium)
   \[ \Rightarrow 31,556,926,000_{\text{ten}} \times 10^{10} \]

2. Very small numbers? (Bohr radius)
   \[ \Rightarrow 0.0000000000529177_{\text{ten}} \times 10^{-11} \]

3. Numbers with \textbf{both} integer & fractional parts?
   \[ \Rightarrow 1.5 \]

First consider #3.

...our solution will also help with #1 and #2.
“Binary Point” like decimal point signifies boundary between integer and fractional parts:

Example 6-bit representation:

```
xx . yyy
```

- $2^1$
- $2^0$
- $2^{-1}$
- $2^{-2}$
- $2^{-3}$
- $2^{-4}$
Representation of Fractions

“Binary Point” like decimal point signifies boundary between integer and fractional parts:

Example 6-bit representation:

\[ xx . yyy \]

\[ 2^1 \quad 2^0 \quad 2^{-1} \quad 2^{-2} \quad 2^{-3} \quad 2^{-4} \]

\[ 10.1010_{\text{two}} = 1 \times 2^1 + 1 \times 2^{-1} + 1 \times 2^{-3} = 2.625_{\text{ten}} \]
"Binary Point" like decimal point signifies boundary between integer and fractional parts:

Example 6-bit representation: \[ \text{xx.yyyy} \]

\[ 10.1010_{\text{two}} = 1 \times 2^1 + 1 \times 2^{-1} + 1 \times 2^{-3} = 2.625_{\text{ten}} \]

If we assume "fixed binary point", range of 6-bit representations with this format:

0 to 3.9375 (almost 4)
## Fractional Powers of 2

<table>
<thead>
<tr>
<th>i</th>
<th>$2^{-i}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.0</td>
</tr>
<tr>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>2</td>
<td>0.25</td>
</tr>
<tr>
<td>3</td>
<td>0.125</td>
</tr>
<tr>
<td>4</td>
<td>0.0625</td>
</tr>
<tr>
<td>5</td>
<td>0.03125</td>
</tr>
<tr>
<td>6</td>
<td>0.015625</td>
</tr>
<tr>
<td>7</td>
<td>0.0078125</td>
</tr>
<tr>
<td>8</td>
<td>0.00390625</td>
</tr>
<tr>
<td>9</td>
<td>0.001953125</td>
</tr>
<tr>
<td>10</td>
<td>0.0009765625</td>
</tr>
<tr>
<td>11</td>
<td>0.00048828125</td>
</tr>
<tr>
<td>12</td>
<td>0.000244140625</td>
</tr>
<tr>
<td>13</td>
<td>0.0001220703125</td>
</tr>
<tr>
<td>14</td>
<td>0.00006103515625</td>
</tr>
<tr>
<td>15</td>
<td>0.000030517578125</td>
</tr>
</tbody>
</table>
What about addition and multiplication?

Addition is straightforward:

\[
\begin{array}{c}
01.100 \\
+ 00.100 \\
\hline
10.000 \\
\end{array}
\]

\[\text{ten}\]

\[= 2.0_{\text{ten}}\]

\[1.5_{\text{ten}}\]
What about addition and multiplication?

Addition is straightforward:

\[
\begin{align*}
01.100 & \quad 1.5_{\text{ten}} \\
+ & \quad 00.100 \quad 0.5_{\text{ten}} \\
\hline
10.000 & \quad 2.0_{\text{ten}} \quad 01.100 \quad 1.5_{\text{ten}}
\end{align*}
\]

Multiplication a bit more complex:

\[
\begin{align*}
01.100 & \quad 1.5_{\text{ten}} \\
\times & \quad 00.100 \quad 0.5_{\text{ten}} \\
\hline
00 & \quad 000 \\
000 & \quad 00 \\
0110 & \quad 0 \\
0000 & \\
0000 & \\
0000 & \\
00000 & \\
0000110000 &
\end{align*}
\]
What about addition and multiplication?

Addition is straightforward:

\[
\begin{align*}
01.100 & \quad 1.5_{\text{ten}} \\
+ \quad 00.100 & \quad 0.5_{\text{ten}} \\
\hline
10.000 & \quad 2.0_{\text{ten}}
\end{align*}
\]

\[
01.100_{\text{ten}}
\]

Multiplication a bit more complex:

\[
\begin{array}{c}
\hline
00.100 \\
+ 00.100 \\
\hline
00 000 \\
\hline
00110 \\
\hline
00000 \\
\hline
00000 \\
\hline
0000110000
\end{array}
\]

Where's the answer, \(0.11\)? (need to remember where point is)
What we really want is to “float” the binary point. Why?
What we really want is to “float” the binary point. Why?

Floating binary point most effective use of our limited bits (and thus more accuracy in our number representation):
Representation of Fractions: Floating Pt

What we really want is to “float” the binary point. Why?

Floating binary point most effective use of our limited bits (and thus more accuracy in our number representation):

Example: put $0.1640625_{10}$ into binary. Represent with 5-bits choosing where to put the binary point: $\ldots 00000.001010100000\ldots$

Store these bits and keep track of the binary point 2 places to the left of the MSB.
Any other solution would lose accuracy!
What we really want is to “float” the binary point. Why?

Floating binary point most effective use of our limited bits (and thus more accuracy in our number representation):

Example: put $0.1640625_{\text{ten}}$ into binary. Represent with 5-bits choosing where to put the binary point: $\ldots \ 00000.001010100000\ldots$

Store these bits and keep track of the binary point 2 places to the left of the MSB.
Any other solution would lose accuracy!

With floating-point rep., each numeral carries an exponent field recording the whereabouts of its binary point. The binary point can be outside the stored bits, so very large and small numbers can be represented.
Scientific Notation (in Decimal)

- **Normalized form**: no leading 0s (exactly one digit to left of decimal point)
- **Alternatives to representing 1/1,000,000,000**
  - **Normalized**: \(1.0 \times 10^{-9}\)
  - **Not normalized**: \(0.1 \times 10^{-8}, 10.0 \times 10^{-10}\)
Scientific Notation (in Binary)

- Computer arithmetic that supports it called **floating point**, because it represents numbers where the binary point is not fixed, as it is for integers.
  - Declare such variable in C as `float`
  - `double` for double precision (64bit vs 32bit).
Floating-Point Representation (1/2)

- Normal format: $+1.\text{xxx…x}_{\text{two}} \times 2^\text{yyy…y}_{\text{two}}$

- Multiple of Word Size (32 bits)

  - $S$ represents Sign
  - Exponent represents $y$’s
  - Significand represents $x$’s (also called Mantissa)

- Represent numbers as small as $2.0_{\text{ten}} \times 10^{-38}$ to as large as $2.0_{\text{ten}} \times 10^{38}$
Floating-Point Representation (2/2)

- What if result too large?

\[ (> 2.0 \times 10^{38}, < -2.0 \times 10^{38} ) \]

- **Overflow!** ⇒ Exponent larger than represented in 8-bit Exponent field

- What if result too small?

\[ (>0 \& < 2.0 \times 10^{-38}, <0 \& > -2.0 \times 10^{-38} ) \]

- **Underflow!** ⇒ Negative exponent larger than represented in 8-bit Exponent field

What would help reduce chances of overflow and/or underflow?
IEEE 754 Floating-Point Standard (1/3)

IEEE 754 32 Bit Single Precision Float

- **Sign bit:** 1 means negative 0 means positive
- Significand in *sign-magnitude* format (not 2’s complement)
IEEE 754 Floating-Point Standard (1/3)

IEEE 754 32 Bit Single Precision Float

- **Sign bit:** 1 means negative 0 means positive
- **Significand in** *sign-magnitude* format (not 2’s complement)
  - To pack more bits, leading 1 implicit for normalized numbers
IEEE 754 Floating-Point Standard (1/3)

IEEE 754 32 Bit Single Precision Float

• Sign bit: 1 means negative 0 means positive

• Significand in \textit{sign-magnitude} format (not 2’s complement)
  • To pack more bits, leading 1 implicit for normalized numbers
  • 1 + 23 bits single, 1 + 52 bits double
  • always true: 0 < \textit{Mantissa} < 1 (for normalized numbers)

• Note: 0 has no leading 1, so reserve exponent value 0 just for number 0
IEEE 754 Floating Point Standard (2/3)

• IEEE 754 uses “biased exponent” representation: Use just magnitude and offset by half the range
IEEE 754 Floating Point Standard (2/3)

- IEEE 754 uses “biased exponent” representation: Use just magnitude and offset by half the range
IEEE 754 Floating Point Standard (2/3)

- IEEE 754 uses “biased exponent” representation: Use just magnitude and offset by half the range

- Designers wanted FP numbers to be used even if no FP hardware; e.g., sort records with FP numbers using integer compares
IEEE 754 uses “biased exponent” representation: Use just magnitude and offset by half the range

- Designers wanted FP numbers to be used even if no FP hardware; e.g., sort records with FP numbers using integer compares
- Wanted bigger (integer) exponent field to represent bigger numbers
IEEE 754 Floating Point Standard (2/3)

- IEEE 754 uses "biased exponent" representation: Use just magnitude and offset by half the range

- Designers wanted FP numbers to be used even if no FP hardware; e.g., sort records with FP numbers using integer compares
- Wanted bigger (integer) exponent field to represent bigger numbers
- 2’s complement poses a problem (because negative numbers look bigger)
IEEE 754 Floating Point Standard (3/3)

- Called **Biased Notation**, where bias is number subtracted to get final number
  - IEEE 754 uses bias of 127 for single precision
  - Subtract 127 from Exponent field to get actual value for exponent

$(-1)^S \times (1 + \text{Mantissa}) \times 2^{(\text{Exponent}-127)}$

- Double precision identical, except with exponent bias of 1023 (half, quad similar)
“Father” of the Floating point standard

IEEE Standard 754 for Binary Floating-Point Arithmetic.

http://www.cs.berkeley.edu/~wkahan/ieee754status/754story.html

Prof. Kahan
“Father” of the Floating point standard

IEEE Standard 754 for Binary Floating-Point Arithmetic.

1989
ACM Turing Award Winner!

http://www.cs.berkeley.edu/~wkahan/ieee754status/754story.html

Prof. Kahan
Wake-up: iClickers!!!

Guess this Floating Point number:

1 1000 0000 1000 0000 0000 0000 0000 0000

A: -1x 2^{128}
B: +1x 2^{-128}
C: -1x 2^1
D: +1.5x 2^{-1}
E: -1.5x 2^1
Representation for $\pm \infty$

- In FP, divide by 0 should produce $\pm \infty$, not overflow.
Representation for $\pm \infty$

- In FP, divide by 0 should produce $\pm \infty$, not overflow.
- Why?
Representation for $\pm \infty$

- In FP, divide by 0 should produce $\pm \infty$, not overflow.
- Why?
  - OK to do further computations with $\infty$. E.g., $X/0 > Y$ may be a valid comparison
Representation for $\pm \infty$

- In FP, divide by 0 should produce $\pm \infty$, not overflow.
- Why?
  - OK to do further computations with $\infty$. E.g., $X/0 > Y$ may be a valid comparison

- IEEE 754 represents $\pm \infty$
  - Most positive exponent reserved for $\infty$
  - Significant all zeroes
Representation for 0

• Represent 0?
  • exponent all zeroes
  • significant all zeroes
  • What about sign? Both cases valid

+0: 0 00000000 000000000000000000000000

−0: 1 00000000 000000000000000000000000
Special Numbers
## Special Numbers

### What have we defined so far?

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Significand</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>nonzero</td>
<td>???</td>
</tr>
<tr>
<td>1-254</td>
<td>anything</td>
<td>+/- fl. pt. #</td>
</tr>
<tr>
<td>255</td>
<td>0</td>
<td>+/- ∞</td>
</tr>
<tr>
<td>255</td>
<td>nonzero</td>
<td>???</td>
</tr>
</tbody>
</table>
Special Numbers

- What have we defined so far?

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Significand</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>nonzero</td>
<td>???</td>
</tr>
<tr>
<td>1-254</td>
<td>anything</td>
<td>+/- fl. pt. #</td>
</tr>
<tr>
<td>255</td>
<td>0</td>
<td>+/- ∞</td>
</tr>
<tr>
<td>255</td>
<td>nonzero</td>
<td>???</td>
</tr>
</tbody>
</table>

- Professor Kahan had clever ideas:
  - Wanted to use Exp=0, 255 & Sig!=0
Representation for Not a Number

• What do I get if I calculate \( \sqrt{-4.0} \) or \( 0/0 \)?
  
  • If \( \infty \) not an error, these shouldn’t be either
  
  • Called Not a Number (NaN)
  
  • Exponent = 255, Significand nonzero

• Why is this useful?
  
  • Hope NaNs help with debugging?
  
  • They contaminate: \( \text{op}(\text{NaN}, X) = \text{NaN} \)
  
  • Can use the significand to identify which!
Representation for Denoms (1/2)

• Problem: There’s a gap among representable FP numbers around 0

• Smallest representable pos num:
  \[ a = 1.0\ldots_{\text{two}} \times 2^{-126} = 2^{-126} \]

• Second smallest representable pos num:
  \[ b = 1.000\ldots_{\text{two}} \times 2^{-126} = (1 + 0.00\ldots_{\text{two}}) \times 2^{-126} = (1 + 2^{-23}) \times 2^{-126} = 2^{-126} + 2^{-149} \]
  \[ a - 0 = 2^{-126} \]
  \[ b - a = 2^{-149} \]

Gaps!
Normalization and implicit 1 is to blame!
• **Solution:**
  - We still haven’t used Exponent = 0, Significand nonzero
  - **DEnormalized number:** no (implied) leading 1, implicit exponent = -126.
  - Smallest representable pos num:
    \[ a = 2^{-149} \]
  - Second smallest representable pos num:
    \[ b = 2^{-148} \]
## Special Numbers Summary

- **Reserve exponents, significands:**

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Significand</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>nonzero</td>
<td>Denorm</td>
</tr>
<tr>
<td>1-254</td>
<td>anything</td>
<td>+/- fl. pt. #</td>
</tr>
<tr>
<td>255</td>
<td>0</td>
<td>+/- $\infty$</td>
</tr>
<tr>
<td>255</td>
<td>nonzero</td>
<td>NaN</td>
</tr>
</tbody>
</table>
ICSILog: Let’s assume you want to calculate logarithms… FAST

IEEE 754 32 Bit Single Precision Float

General Equations:

value = Sign \times 2^{\text{Exponent}} \times \text{Mantissa}

\log_2(\text{value}) = \text{Exponent} + \log_2(\text{Mantissa}) \Rightarrow 
\log(\text{value}) = \log_2(\text{value}) \times 0.6931

See: Technical Report in reading for this lecture!

ICSILog: Idea

With \( q=8 \) usually \( \sim 30 \) CPU cycles
ICSILog Lookup Table in $
# A Quick Comparison

<table>
<thead>
<tr>
<th>Method</th>
<th>Speed up</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Log</td>
<td>1.0</td>
<td>0.00</td>
</tr>
<tr>
<td>AMD ACML Log</td>
<td>~1.7</td>
<td>1.10e-07</td>
</tr>
<tr>
<td>Fast-Math Log</td>
<td>~2.0</td>
<td>1.42e-07</td>
</tr>
<tr>
<td>FastLog (Taylor order 3)</td>
<td>~6-7</td>
<td>4.26e-03</td>
</tr>
<tr>
<td>ICSILog (q=0)</td>
<td>~1.3</td>
<td>~0.00</td>
</tr>
<tr>
<td>ICSILog (q=7)</td>
<td>~6</td>
<td>6.55e-06</td>
</tr>
<tr>
<td>ICSILog (q=8)</td>
<td>~6-8</td>
<td>0.0000131</td>
</tr>
<tr>
<td>ICSILog v2 (q=7)</td>
<td>~6</td>
<td>3.271E-06</td>
</tr>
<tr>
<td>ICSILog v2 (q=8)</td>
<td>~6-8</td>
<td>0.00000656</td>
</tr>
</tbody>
</table>

- ICSILog() ~6 times faster than log() for floats
- Arbitrarily accurate (depends on cache usage)
Conclusion

• Floating Point lets us:
  • Represent numbers containing both integer and fractional parts; makes efficient use of available bits.
  • Store approximate values for very large and very small #s.

• IEEE 754 Floating-Point Standard is most widely accepted attempt to standardize interpretation of such numbers (Every desktop or server computer sold since ~1997 follows these conventions)

• Knowledge of the inner workings of this can help efficiency even for a “pure" software developer.

www.h-schmidt.net/FloatApplet/IEEE754.html