CS 61C:
Great Ideas in Computer Architecture

*Floating Point Arithmetic*

Instructors:
Vladimir Stojanovic & Nicholas Weaver

http://inst.eecs.berkeley.edu/~cs61c/
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
  - How do we know?

Harness Parallelism & Achieve High Performance

How do we know?

Computer

- Core
- Core
- Core
- Cache
- Memory
- Instruction Unit(s)
- Functional Unit(s)
- Logic Gates
- Cache Memory

Smart Phone
Review of Numbers

• Computers are made to deal with numbers

• What can we represent in N bits?
  • $2^N$ things, and no more! They could be…
  • Unsigned integers:
    
    \[
    0 \quad \text{to} \quad 2^N - 1
    \]

    (for \(N=32\), \(2^{32} - 1 = 4,294,967,295\))

  • Signed Integers (Two’s Complement)
    
    \[
    -2^{(N-1)} \quad \text{to} \quad 2^{(N-1)} - 1
    \]

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

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

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

3. Numbers with both integer & fractional parts?
   \[1.5\]

First consider #3.

...our solution will also help with #1 and #2.
Representation of Fractions

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

Example 6-bit representation:

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

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:

```
  01.100
+ 00.100
  10.000
```

```
  1.5_{ten}
+ 0.5_{ten}
  2.0_{ten}
```

```
  01.100
+ 00.100
  00.000
```

```
  1.5_{ten}
+ 0.5_{ten}
  2.0_{ten}
```

Multiplication a bit more complex:

```
  000 00
  0110 0
  00000
  00000
```

```
  0000110000
```

Where’s the answer, 0.11? (need to remember where point is)
Representation of Fractions

So far, in our examples we used a “fixed” binary point. 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.

... 00000.001010100000...

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)

mantissa \(6.02_{\text{ten}}\times10^{23}\)

exponent

decimal point

radix (base)

- 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)

mantissa

1.01\textsubscript{two} \times 2^{-1}

"binary point"

exponent

radix (base)

• Computer arithmetic that supports it called \textbf{floating point}, because it represents numbers where the binary point is not fixed, as it is for integers

  • Declare such variable in C as \texttt{float}
    • \texttt{double} for double precision.
Floating-Point Representation (1/2)

- Normal format: \(+1.xxx...x_{\text{two}} \times 2^{y_{\text{two}}}\)

- Multiple of Word Size (32 bits)

<table>
<thead>
<tr>
<th>S</th>
<th>Exponent</th>
<th>Significand</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 bit</td>
<td>8 bits</td>
<td>23 bits</td>
</tr>
</tbody>
</table>

- \(S\) represents Sign
- Exponent represents \(y\)’s
- Significand represents \(x\)’s

- 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)

Single Precision (Double Precision similar):

\[ \begin{array}{cccccc}
31 & 30 & 23 & 22 & 0 \\
S & \text{Exponent} & \text{Significand} \\
\end{array} \]

1 bit 8 bits 23 bits

- **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
  - 1 + 23 bits single, 1 + 52 bits double
  - always true: \(0 < \text{Significand} < 1\) (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
  - 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)
    - Use just magnitude and offset by half the range
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 prec.
  - Subtract 127 from Exponent field to get actual value for exponent

• **Summary** (single precision):
  
  | 31 30 23 22 | 0 |
  | S | Exponent | Significand |
  | 1 bit | 8 bits | 23 bits |
  
  \((-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}\)

• 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.

1989 ACM Turing Award Winner!

Prof. Kahan

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

• Guess this Floating Point number:
  1 1000 0000 1000 0000 0000 0000 0000 0000 000

A: -1x 2^{128}
B: +1x 2^{-128}
C: -1x 2^1
D: +1.5x 2^{-1}
E: -1.5x 2^1
Administrivia

• Project 3-2 extended until 03/20 @ 23:59:59

• Guerrilla Session: Caches/ Proj 3-2 OH
  – Sat 3/19 1 - 3 PM @ 521 Cory
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 $\infty$
  - Significands all zeroes
Representation for 0

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

+0: 0 00000000 0000000000000000000000000

−0: 1 00000000 0000000000000000000000000
Special Numbers

• What have we defined so far? (Single Precision)

<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!
• 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:
    \[ b = 1.000\ldots_{\text{two}} \times 2^{-126} = (1 + 0.00\ldots_{\text{two}}) \times 2^{-126} = (1 + 2^{-23}) \times 2^{-126} = 2^{-126} + 2^{-149} \]

\[ a - 0 = 2^{-126} \]
\[ b - a = 2^{-149} \]

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>+/- ∞</td>
</tr>
<tr>
<td>255</td>
<td>nonzero</td>
<td>NaN</td>
</tr>
</tbody>
</table>
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)

Summary (single precision):

<table>
<thead>
<tr>
<th>31 30</th>
<th>23 22</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Exponent</td>
<td>Significand</td>
</tr>
</tbody>
</table>

1 bit 8 bits 23 bits

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

Double precision identical, except with exponent bias of 1023 (half, quad similar)