ESE 345 Computer Architecture

Floating point
Number Rep Revisited

- Given one word (32 bits), what can we represent so far?
  - Signed and Unsigned Integers
  - Characters (ASCII)
  - Instructions

- How do we encode the following:
  - Real numbers (e.g. 3.14159)
  - Very large numbers (e.g. $6.02 \times 10^{23}$)
  - Very small numbers (e.g. $6.626 \times 10^{-34}$)
  - Special numbers (e.g. $\infty$, NaN)
Goals of Floating Point

- Support a wide range of values
  - Both very small and very large
- Keep as much *precision* as possible
- Help programmer with errors in real arithmetic
  - Support $+\infty$, $-\infty$, Not-A-Number (NaN), exponent overflow and underflow
- Keep encoding that is somewhat compatible with two’s complement
  - e.g. 0 in FP is same as 0 in two’s complement
  - Make it possible to sort without needing to do a special floating point comparison
Representation of Fractions

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

Example 6-bit representation: \( xx \cdot yyyy \)

Example: \( 10.1010_{\text{two}} = 1 \times 2^1 + 1 \times 2^{-1} + 1 \times 2^{-3} = 2.625_{\text{ten}} \)
Scientific Notation (Decimal)

- **Normalized form**: no leading 0s (exactly one digit to the left of the decimal point)

- Alternatives to representing 1/1,000,000,000
  - Normalized: $1.0 \times 10^{-9}$
  - Not normalized: $0.1 \times 10^{-8}, 10.0 \times 10^{-10}$

Example: $6.02_{\text{ten}} \times 10^{23}$
Scientific Notation (Binary)

- Computer arithmetic that supports this called **floating point** due to the “floating” of the binary point

  - Declare such variable in C as `float`
Translating To and From Scientific Notation

- Consider the number $1.011_{\text{two}} \times 2^4$
- To convert to ordinary number, shift the binary point to the right by 4
  - Result: $10110_{\text{two}} = 22_{\text{ten}}$
- For negative exponents, shift the binary point to the left
  - $1.011_{\text{two}} \times 2^{-2} \Rightarrow 0.01011_{\text{two}} = 0.34375_{\text{ten}}$
- Go from ordinary number to scientific notation by shifting until in *normalized* form
  - $1101.001_{\text{two}} \Rightarrow 1.101001_{\text{two}} \times 2^3$
“Father” of the Floating Point Standard

IEEE Standard 754 for Binary Floating-Point Arithmetic.

1989 ACM Turing Award Winner!

Prof. Kahan
Floating Point Representation

- Use normalized, Base 2 scientific notation:
  \[ +1.\textit{xxx…x}_{two} \times 2^{\textit{yyy…y}_{two}} \]

- Split a 32-bit word into 3 fields:
  
  \[ \begin{array}{c}
    31 & 30 & 23 & 22 & 0 \\
    S & \text{Exponent} & \text{Significand} \\
  \end{array} \]

  - 1 bit 8 bits 23 bits
  
  - \( S \) represents \textit{Sign} (1 is negative, 0 positive)
  - \textit{Exponent} represents \( y \)'s
  - \textit{Significand} represents \( x \)'s
Exponent Comparison

• Which is smaller? (i.e. closer to $-\infty$)

  0 or $1 \times 10^{-127}$ ?

  $1 \times 10^{-126}$ or $1 \times 10^{-127}$ ?

  $-1 \times 10^{-127}$ or 0 ?

  $-1 \times 10^{-126}$ or $-1 \times 10^{-127}$ ?
Floating Point Overflow and Underflow

- What if result x is too large? \((\text{abs}(x) > 2^{128})\)
  - **Overflow**: Exponent is larger than can be represented

- What if result x too small? \((0 < \text{abs}(x) < 2^{-149})\)
  - **Underflow**: Negative exponent is larger than can be represented

What if result runs off the end of the significand?
- **Rounding** occurs
- FP has different *rounding modes*
  - truncate, round towards \(+\infty\), round towards \(-\infty\),
  - round to nearest even using guard, round , and sticky bits

How to reduce chances of overflow or underflow?
Double Precision FP Representation

- Next multiple of word size (64 bits)

- Double Precision (vs. Single Precision)
  - C variable declared as `double`
  - Represent numbers almost as small as $2.0 \times 10^{-308}$ to almost as large as $2.0 \times 10^{308}$
  - Primary advantage is greater precision due to larger significand
IEEE 754 Floating Point Standard (1/3)

• Single Precision, DP similar
• Sign bit: 1 means negative
  0 means positive
• Significand:
  • To pack more bits, leading 1 implicit
    for normalized numbers
  • 1 + 23 bits single, 1 + 52 bits double

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

Negative Exponent?
• 2’s comp? 1.0 x 2⁻¹ v. 1.0 x 2⁺¹ (1/2 v. 2)

1/2 0 1111 1111 000 0000 0000 0000 0000 0000 0000
 2 0 0000 0001 000 0000 0000 0000 0000 0000 0000

• This notation using integer compare of 1/2 v. 2 makes 1/2 > 2!

• Instead, pick notation 0000 0001 is most negative, and 1111 1111 is most positive

1.0 x 2⁻¹ v. 1.0 x 2⁺¹ (1/2 v. 2)

1/2 0 0111 1110 000 0000 0000 0000 0000 0000 0000
 2 0 1000 0000 000 0000 0000 0000 0000 0000 0000
IEEE 754 Floating Point Standard (3/3)

- Called Biased Notation, where bias is number subtract to get real number
- IEEE 754 uses bias of 127 for single prec.
- Subtract 127 from Exponent field to get actual value for exponent
- 1023 is bias for double precision
- Summary (single precision):
  \[ (-1)^s \times (1 + \text{Significand}) \times 2^{(\text{Exponent} - 127)} \]

Double precision identical, except with exponent bias of 1023
Single-precision Floating Point Encoding

\[ (-1)^s \times (1.\text{ Significand}) \times 2^{(\text{Exponent} - 127)} \]

- Note the implicit 1 in front of the Significand
  - Ex: 0011 1111 1100 0000 0000 0000 0000 0000\text{two} is read as 1.1\text{two} = 1.5\text{ten}, NOT 0.1\text{two} = 0.5\text{ten}
- Gives us an extra bit of precision
- Represent numbers as small as \(2.0 \times 10^{-38}\) to as large as \(2.0 \times 10^{38}\)
Representing Very Small Numbers

- Recall: What happened to zero?
  - Using standard encoding 0x00000000 is $1.0 \times 2^{-127} \neq 0$
  - Special case: Exp and Sig all zeros = 0
  - Two (positive and negative) zeros!

- Numbers closest to 0:
  - $a = 1.0\ldots0_{\text{two}} \times 2^{-126} = 2^{-126}$
  - $b = 1.0\ldots01_{\text{two}} \times 2^{-126} = 2^{-126} + 2^{-149}$
  - Normalization and implicit 1 are to blame
  - Special case: Exp = 0, Sig $\neq 0$ are denormalized numbers
Denorm Numbers

- Short for “denormalized numbers”
  - No leading 1
  - Careful! Implicit exponent is -126 even though Exp = 0x00 (not -127)

- Now what do the gaps look like?
  - Smallest norm: \( \pm 1.0...0_{\text{two}} \times 2^{-126} = \pm 2^{-126} \)
  - Largest denorm: \( \pm 0.1...1_{\text{two}} \times 2^{-126} = \pm (2^{-126} - 2^{-149}) \)
  - Smallest denorm: \( \pm 0.0...01_{\text{two}} \times 2^{-126} = \pm 2^{-149} \)
Other Special Cases

- Exp = 0xFF, Sig = 0: ±∞
  - e.g. division by 0
  - Still work in comparisons

- Exp = 0xFF, Sig ≠ 0: Not a Number (NaN)
  - e.g. square root of negative number, 0/0, −∞−∞
  - NaN propagates through computations
    - op(NaN,X) = NaN
    - Hope NaNs help with debugging?

- Largest value (besides ∞)?
  - Exp = 0xFF has now been taken!
    - Exp = 0xFE has largest: $1.1\ldots1_{\text{two}} \times 2^{127} = 2^{128} - 2^{104}$
### IEEE 754 Floating Point Numbers Summary

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Significand</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>± 0</td>
</tr>
<tr>
<td>0</td>
<td>non-zero</td>
<td>Denorm Num</td>
</tr>
<tr>
<td>1-254</td>
<td>anything</td>
<td>± FP Num</td>
</tr>
<tr>
<td>255</td>
<td>0</td>
<td>± ∞</td>
</tr>
<tr>
<td>255</td>
<td>non-zero</td>
<td>NaN</td>
</tr>
</tbody>
</table>
Floating Point Limitations

- FP addition is NOT associative!
  - You can find Big and Small numbers such that:
    Small + Big + Small ≠ Small + Small + Big
  - This is due to rounding errors: FP approximates results because it only has 23 bits for Significand

- Despite being seemingly “more accurate,” FP cannot represent all integers
  - Be careful when casting between int and float
Example: Convert FP to Decimal

\[
\begin{array}{cccccccccccc}
0 & 0110 & 1000 & 101 & 0101 & 0100 & 0011 & 0100 & 0010 \\
\end{array}
\]

- **Sign**: 0 means positive
- **Exponent**:
  - \(0110 \ 1000_{\text{two}} = 104_{\text{ten}}\)
  - Bias adjustment: \(104 - 127 = -23\)
- **Significand**:
  \[
  1.10101010100001101000010 \\
  = 1 + 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 0 \cdot 2^{-4} + 1 \cdot 2^{-5} + \cdots \\
  = 1 + 2^{-1} + 2^{-3} + 2^{-5} + 2^{-7} + 2^{-9} + 2^{-14} + 2^{-15} + 2^{-17} + 2^{-22} \\
  = 1.0 + 0.666115 \\
\]
- **Represents**: \(1.666115_{\text{ten}} \times 2^{-23} \approx 1.986 \times 10^{-7}\)
Think it over

```
1 1000 0010 111 0000 0000 0000 0000 0000
```

- What is the decimal equivalent of this floating point number?
  - 1: -1.75
  - 2: -3.5
  - 3: -3.75
  - 4: -7
  - 5: -7.5
  - 6: -15
  - 7: -7 * 2^129
  - 8: -129 * 2^7
What is the decimal equivalent of this floating point number?

\((-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent} - 127)} = \)
\((-1)^1 \times (1 + .111) \times 2^{(130 - 127)} = \)
\(-1 \times (1.111) \times 2^3 = \)
\(-1111.0 = -15\)
Example: Converting Decimal to FP

-2.340625 \times 10^1

1. Denormalize: -23.40625
2. Convert integer part:
   \[ 23 = 16 + 4 + 2 + 1 = 10111_{\text{two}} \]
3. Convert fractional part:
   \[ .40625 = .25 + .125 + .03125 = 0.01101_{\text{two}} \]
4. Put parts together and normalize:
   \[ 10111.01101 = 1.011101101 \times 2^4 \]
5. Convert exponent: \[ 4 + 127 = 1000011_{\text{two}} \]
6. \[ \begin{array}{cccccccccccc}
   1 & 1000 & 0011 & 011 & 1011 & 0100 & 0000 & 0000 & 0000
\end{array} \]
Example: Convert to FP

1/3

= 0.33333..._{ten}
= 0.25 + 0.0625 + 0.015625 + 0.00390625 + ...
= 1/4 + 1/16 + 1/64 + 1/256 + ...
= 2^{-2} + 2^{-4} + 2^{-6} + 2^{-8} + ...
= 0.0101010101..._{two} \times 2^0
= 1.0101010101..._{two} \times 2^{-2}

- Sign: 0
- Exponent = -2 + 127 = 125 = 01111101_{two}
- Significand = 0101010101..._{two}
Casting floats to ints and vice versa

**(int) floating_point_expression**

Coerces and converts it to the *nearest* integer
(C uses truncation)

\[ i = \text{(int)} (3.14159 \times f); \]

**(float) integer_expression**

Converts integer to *nearest* floating point

\[ f = f + \text{(float)} i; \]
MIPS Floating-Point Instructions

- MIPS instructions: .s for single, .d for double
  - FP addition: add.s (single) and add.d (double)
  - Similar instruction naming scheme for subtraction, multiplication, and division
  - Comparison and branching a little different; see next slide
- MIPS has 32 separate registers for floating point operations ($f\#$)
  - Use adjacent pairs (e.g. $f1$ and $f2$) for double
  - lwc1 (load word), swc1 (store word)
MIPS Floating-Point Instructions

### MIPS Floating-point operands

<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 floating-point registers</td>
<td>$f0$, $f1$, $f2$, ..., $f31</td>
<td>MIPS floating-point registers are used in pairs for double precision numbers.</td>
</tr>
<tr>
<td>2^31 memory words</td>
<td>Memory[0], Memory[4], ..., Memory[4294967292]</td>
<td>Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.</td>
</tr>
</tbody>
</table>

### MIPS floating-point assembly language

#### Arithmetic

<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP add single</td>
<td>add.s</td>
<td>$f2, f3, $f4, $f6</td>
<td>$f2 = $f4 + $f6</td>
<td>FP add (single precision)</td>
</tr>
<tr>
<td>FP subtract single</td>
<td>sub.s</td>
<td>$f2, f3, $f4, $f6</td>
<td>$f2 = $f4 - $f6</td>
<td>FP sub (single precision)</td>
</tr>
<tr>
<td>FP multiply single</td>
<td>mul.s</td>
<td>$f2, f3, $f4, $f6</td>
<td>$f2 = $f4 × $f6</td>
<td>FP multiply (single precision)</td>
</tr>
<tr>
<td>FP divide single</td>
<td>div.s</td>
<td>$f2, f3, $f4, $f6</td>
<td>$f2 = $f4 / $f6</td>
<td>FP divide (single precision)</td>
</tr>
<tr>
<td>FP add double</td>
<td>add.d</td>
<td>$f2, f3, $f4, $f6</td>
<td>$f2 = $f4 + $f6</td>
<td>FP add (double precision)</td>
</tr>
<tr>
<td>FP subtract double</td>
<td>sub.d</td>
<td>$f2, f3, $f4, $f6</td>
<td>$f2 = $f4 - $f6</td>
<td>FP sub (double precision)</td>
</tr>
<tr>
<td>FP multiply double</td>
<td>mul.d</td>
<td>$f2, f3, $f4, $f6</td>
<td>$f2 = $f4 × $f6</td>
<td>FP multiply (double precision)</td>
</tr>
<tr>
<td>FP divide double</td>
<td>div.d</td>
<td>$f2, f3, $f4, $f6</td>
<td>$f2 = $f4 / $f6</td>
<td>FP divide (double precision)</td>
</tr>
</tbody>
</table>

#### Data transfer

<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>load word compr.</td>
<td>lwcl</td>
<td>$f1, 100(32)</td>
<td>$f1 = Memory[3*2 + 100]</td>
<td>32-bit data to FP register</td>
</tr>
<tr>
<td>store word compr.</td>
<td>swcl</td>
<td>$f1, 100(32)</td>
<td>Memory[3*2 + 100] = $f1</td>
<td>32-bit data to memory</td>
</tr>
</tbody>
</table>

#### Conditional branch

<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch on FP true</td>
<td>bclt</td>
<td>25</td>
<td>if (cond == 1) go to PC + 4 + 100</td>
<td>PC-relative branch if FP cond.</td>
</tr>
<tr>
<td>branch on FP false</td>
<td>bclf</td>
<td>25</td>
<td>if (cond == 0) go to PC + 4 + 100</td>
<td>PC-relative branch if not FP cond.</td>
</tr>
<tr>
<td>FP compare single (eq, ne, lt, le, gt, ge)</td>
<td>c.lt.s</td>
<td>$f2, $f4</td>
<td>if ($f2 &lt; $f4) cond = 1; else cond = 0</td>
<td>FP compare less than single precision</td>
</tr>
<tr>
<td>FP compare double (eq, ne, lt, le, gt, ge)</td>
<td>c.lt.d</td>
<td>$f2, $f4</td>
<td>if ($f2 &lt; $f4) cond = 1; else cond = 0</td>
<td>FP compare less than single precision</td>
</tr>
</tbody>
</table>

### MIPS floating-point machine language

<table>
<thead>
<tr>
<th>Name</th>
<th>Format</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>add.s</td>
<td>R</td>
<td>17 16 6 4 2 0</td>
<td>add.s $f2, $f4, $f6</td>
</tr>
<tr>
<td>sub.s</td>
<td>R</td>
<td>17 16 6 4 2 1</td>
<td>sub.s $f2, $f4, $f6</td>
</tr>
<tr>
<td>mul.s</td>
<td>R</td>
<td>17 16 6 4 2 2</td>
<td>mul.s $f2, $f4, $f6</td>
</tr>
<tr>
<td>div.s</td>
<td>R</td>
<td>17 16 6 4 2 3</td>
<td>div.s $f2, $f4, $f6</td>
</tr>
<tr>
<td>add.d</td>
<td>R</td>
<td>17 17 6 4 2 0</td>
<td>add.d $f2, $f4, $f6</td>
</tr>
<tr>
<td>sub.d</td>
<td>R</td>
<td>17 17 6 4 2 1</td>
<td>sub.d $f2, $f4, $f6</td>
</tr>
<tr>
<td>mul.d</td>
<td>R</td>
<td>17 17 6 4 2 2</td>
<td>mul.d $f2, $f4, $f6</td>
</tr>
<tr>
<td>div.d</td>
<td>R</td>
<td>17 17 6 4 2 3</td>
<td>div.d $f2, $f4, $f6</td>
</tr>
<tr>
<td>lwcl</td>
<td>I</td>
<td>49 20 2 100</td>
<td>lwcl $f2, 100(32)</td>
</tr>
<tr>
<td>swcl</td>
<td>I</td>
<td>57 20 2 100</td>
<td>swcl $f2, 100(32)</td>
</tr>
<tr>
<td>bclt</td>
<td>I</td>
<td>17 8 0 25</td>
<td>bclt 25</td>
</tr>
<tr>
<td>bclf</td>
<td>I</td>
<td>17 8 0 25</td>
<td>bclf 25</td>
</tr>
<tr>
<td>c.lt.s</td>
<td>R</td>
<td>17 16 4 2 0 60</td>
<td>c.lt.s $f2, $f4</td>
</tr>
<tr>
<td>c.lt.d</td>
<td>R</td>
<td>17 17 4 2 0 60</td>
<td>c.lt.d $f2, $f4</td>
</tr>
</tbody>
</table>

#### Field size

| Field size | 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits | All MIPS instructions 32 bits |

CA: Floating Point
Floating-Point Addition

1. Compare the exponents of the two numbers; shift the smaller number to the right until its exponent would match the larger exponent

2. Add the significands

3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent

4. Round the significand to the appropriate number of bits

Overflow or underflow? Yes

No

Still normalized?

Yes

No

Exception

Done

Sign  Exponent  Fraction

Small ALU

Exponent difference

0 1

0 1

0 1

Big ALU

Shift right

Shift left or right

Increment or decrement

Rounding hardware

Sign  Exponent  Fraction

Control

Compare exponents

Shift smaller number right

Add

Normalize

Round

CA: Floating Point
Extra bits for Rounding

How many extra bits?

IEEE: As if computed the result exactly and rounded.

Addition:

\[
\begin{array}{ccc}
1.xxxxx & 1.xxxxx & 1.xxxxxx \\
+ & 1.xxxxx & 0.001xxxxx & 0.01xxxxx \\
\hline
1x.xxxxxy & 1.xxxxxxxyy & 1x.xxxxxxyyy \\
post-normalization & pre-normalization & pre and post
\end{array}
\]

- **Guard Digits**: digits to the right of the first p digits of significand to guard against loss of digits – can later be shifted left into first P places during normalization.
Guard and Round Digits

normalized result, but some non-zero digits to the right of the significand --> the number should be rounded

one round digit must be carried to the right of the guard digit so that after a normalizing left shift, the result can be rounded, according to the value of the round digit

**IEEE Standard:**

four rounding modes: round to nearest even (default)
round towards plus infinity
round towards minus infinity
round towards 0

round to nearest:
round digit < B/2 then truncate
> B/2 then round up (add 1 to ULP: unit in last place)
= B/2 then round to nearest even digit

*it can be shown that this strategy minimizes the mean error introduced by rounding*
Sticky Bit

Additional bit to the right of the round digit to better fine tune rounding.

\[
\begin{array}{cccccccccc}
  & & & & d_0 & d_1 & d_2 & d_3 & \ldots & d_{p-1} & 0 & 0 & 0 \\
+ & 0 & 0 & 0 & X & \ldots & X & X & X & S \\
  & X & X & S
\end{array}
\]

Sticky bit: set to 1 if any 1 bits fall off the end of the round digit.

Rounding Summary:

Normal operations in +, -, *, / require one carry/borrow bit + one guard digit.

One round digit needed for correct rounding.

Sticky bit needed when round digit is B/2 for max accuracy.

Rounding to nearest has mean error = 0 if uniform distribution of digits are assumed.
Floating-Point Multiplication

1. Add the biased exponents of the two numbers, subtracting the bias from the sum to get the new biased exponent

2. Multiply the significands

3. Normalize the product if necessary, shifting it right and incrementing the exponent

4. Round the significand to the appropriate number of bits

5. Set the sign of the product to positive if the signs of the original operands are the same; if they differ make the sign negative

Done
In Conclusion

- Floating Point numbers **approximate** values that we want to use
- 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):
  
  \[
  (-1)^s \times (1 \cdot \text{Significand}) \times 2^{(\text{Exponent} - 127)}
  \]
  - Double precision identical, bias of 1023
Acknowledgements

- These slides contain material developed and copyright by:
  - Morgan Kauffmann (Elsevier, Inc.)
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)
  - Justin Hsia (UCB)