Overview

- Floating Point Numbers
- Motivation: Decimal Scientific Notation
  - Binary Scientific Notation
- Floating Point Representation inside computer (binary)
  - Greater range, precision
- Decimal to Floating Point conversion, and vice versa
- Big Idea: Type is not associated with data
- MIPS floating point instructions, registers

Review of Numbers

- Computers are made to deal with numbers
- What can we represent in N bits?
  - Unsigned integers: 0 to $2^N - 1$
  - Signed Integers (Two’s Complement)
    - $-2^{(N-1)}$ to $2^{(N-1)} - 1$

Other Numbers

- What about other numbers?
  - Very large numbers? (seconds/century) $3,155,760,000_{10}$ ($3.15576_{10} \times 10^9$)
  - Very small numbers? (atomic diameter) $0.00000001_{10}$ ($1.0_{10} \times 10^{-8}$)
  - Rationals (repeating pattern)
    - $2/3$ ($0.666666666\ldots$)
  - Irrationals
    - $\sqrt{2}$ ($1.414213562373\ldots$)
  - Transcendentals
    - $e$ ($2.718\ldots$), $\pi$ ($3.141\ldots$)
- All represented in scientific notation
Scientific Notation Review

- Mantissa $6.02 \times 10^{23}$
- Exponent
- Decimal point
- Radix (base)

• Normalized form: no leadings 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 for Binary Numbers

- Mantissa $1.0_{two} \times 2^{-1}$
- Exponent
- Binary point
- Radix (base)

• Computer arithmetic that supports it called floating point, because it represents numbers where binary point is not fixed, as it is for integers
  - Declare such variable in C as float

Floating Point Representation [1]

• Normal format: $+1.xxxxxxxxx_{two} \times 2^{yyyy}_{two}$
• Multiple of Word Size (32 bits)

31 30 23 22 0
S Exponent Significand
1 bit 8 bits 23 bits

- $S$ represents Sign of significand (number)
  - Exponent represents y’s
  - Significand represents x’s
- Represent numbers as small as $2.0 \times 10^{-38}$ to as large as $2.0 \times 10^{38}$

Floating Point Representation [2]

• What if result too large? ($> 2.0 \times 10^{38}$)
  - Overflow!
  - Overflow $=>$ Exponent larger than represented in 8-bit Exponent field
• What if result too small? ($>0, < 2.0 \times 10^{-38}$)
  - Underflow!
  - Underflow $=>$ Negative exponent larger than represented in 8-bit Exponent field
• How to reduce chances of overflow or underflow?
### Double Precision Fl. Pt.

- **Next Multiple of Word Size (64 bits)**

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Significand</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30</td>
<td>20 19</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1 bit</th>
<th>11 bits</th>
<th>20 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Significand (cont’d)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>32 bits</td>
</tr>
</tbody>
</table>

- **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 $2.0 \times 10^{308}$
  - But primary advantage is greater accuracy due to larger significand


- **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
    - always true: $0 < \text{Significand} < 1$
  - **Note:** 0 has no leading 1, so reserve exponent value 0 just for number 0


- **Negative Exponent?**
  - 2’s comp? $1.0 \times 2^{-1}$ v. $1.0 \times 2^{+1}$ ($1/2$ v. 2)

<table>
<thead>
<tr>
<th>$1/2$</th>
<th>$01111111000000000000000000000000$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2$</td>
<td>$00000001000000000000000000000000$</td>
</tr>
</tbody>
</table>

  - This notation using integer compare of $1/2$ v. 2 makes $1/2 > 2!$  
  - Instead, pick notation $00000001$ is most negative, and $11111111$ is most positive
    - $1.0 \times 2^{-1}$ v. $1.0 \times 2^{+1}$ ($1/2$ v. 2)

<table>
<thead>
<tr>
<th>$1/2$</th>
<th>$00111111000000000000000000000000$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2$</td>
<td>$01000000000000000000000000000000$</td>
</tr>
</tbody>
</table>


- **Called Biased Notation**, where bias is number subtract to get real number
  - IEEE 754 uses bias of 127 for single precision
  - Subtract 127 from Exponent field to get actual value for exponent
  - 1023 is bias for double precision

- **Summary (single precision):**

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Significand</th>
</tr>
</thead>
<tbody>
<tr>
<td>$3130$</td>
<td>$2322$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1 bit</th>
<th>8 bits</th>
<th>23 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$01111111000000000000000000000000$</td>
<td></td>
</tr>
</tbody>
</table>

- **(-1)$^S$ x (1 + Significand) x $2^{(\text{Exponent}-127)}$**
  - Double precision identical, except with exponent bias of 1023 and significand of 52 bits
Understanding the Significand [1]

• Method 1 (Fractions):
  - In decimal: 0.34010 => \( \frac{34010}{100010} \)
    
  - In binary: 0.1102 => \( \frac{1102}{10002} = \frac{6}{8} \)
    
  - Advantage: less purely numerical, more thought oriented; this method usually helps people understand the meaning of the significand better

Understanding the Significand [2]

• Method 2 (Place Values):
  - Convert from scientific notation
  - In decimal: 1.6732 = \( (1 \times 10^0) + (6 \times 10^{-1}) + (7 \times 10^{-2}) + (3 \times 10^{-3}) + (2 \times 10^{-4}) \)
  - In binary: 1.1001 = \( (1 \times 2^0) + (1 \times 2^{-1}) + (0 \times 2^{-2}) + (0 \times 2^{-3}) + (1 \times 2^{-4}) \)
  - Interpretation of value in each position extends beyond the decimal/binary point
  - Advantage: good for quickly calculating significand value; use this method for translating FP numbers

Ex: Converting Binary FP to Decimal

0110 1000 101 0101 0100 0011 0100 0010

• Sign: 0 => positive
• Exponent:
  - 0110 1000 \(_{\text{two}} = 104\)\(_{\text{ten}}\)
  - Bias adjustment: 104 - 127 = -23
• Significand:
  - \( 1 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5} + \ldots \)
  - \( = 1 + 2^{-1} + 2^{-3} + 2^{-5} + 2^{-7} + 2^{-9} + 2^{-14} + 2^{-15} + 2^{-17} + 2^{-22} \)
  - 1.0 \( + 0.666115 \)
• Represents: 1.666115\(_{\text{ten}} \times 2^{-23} \approx 1.986 \times 10^{-7} \)
  (about 2/10,000,000)

Continuing Example: Binary to ???

0011 0100 0101 0101 0100 0011 0100 0010

• Convert 2’s Comp. Binary to Integer:
  - \( 2^{29} + 2^{28} + 2^{26} + 2^{22} + 2^{20} + 2^{18} + 2^{16} + 2^{14} + 2^{9} + 2^{8} + 2^{6} + 2^{1} \)
  = 878,003,010\(_{\text{ten}}\)

• Convert Binary to Instruction:

• Convert Binary to ASCII:

  \( 0011 \ 0100 \ 0101 \ 0101 \ 0100 \ 0011 \ 0100 \ 0010 \)

  \( = 13 \ 2 \ 21 \ 17218 \)

  ori $s5, $v0, 17218

• Convert Binary to ASCII:

  \( 0011 \ 0100 \ 0101 \ 0101 \ 0100 \ 0011 \ 0100 \ 0010 \)

  \( = 4 \ U \ C \ B \)
**Big Idea: Type not associated with Data**

- What does bit pattern mean:
  - $1.986 \times 10^{-7}$
  - 878,003,010
  - “4UCB”?
  - ori $s5, s0, 17218$?

- Data can be anything; operation of instruction that accesses operand determines its type!
  - Side-effect of stored program concept: instructions stored as numbers

- Power/danger of unrestricted addresses/pointers: use ASCII as Fl. Pt., instructions as data, integers as instructions, ...

**Converting Decimal to FP [1]**

- Simple Case: If denominator is an exponent of 2 (2, 4, 8, 16, etc.), then it's easy.
- Show IEEE 32-bit representation of -0.75
  - $-0.75 = -3/4$
  - $-11_{two} / 100_{two} = -0.11_{two}$
  - Normalized to $-1.1_{two} \times 2^{-1}$
  - $(-1)^5 \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$
  - $(-1)^1 \times (1 + .100 \ 0000 \ ... \ 0000) \times 2^{(126-127)}$

  \[
  1 \ 0111 \ 1110 \ 100 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000
  \]

**Converting Decimal to FP [2]**

- Not So Simple Case: If denominator is not an exponent of 2.
  - Then we can't represent number precisely, but that's why we have so many bits in significand: for precision
  - Once we have significand, normalizing a number to get the exponent is easy.
  - So how do we get the significand of a neverending number?

**Converting Decimal to FP [3]**

- Fact: All rational numbers have a repeating pattern when written out in decimal.
- Fact: This still applies in binary.
- To finish conversion:
  - Write out binary number with repeating pattern.
  - Cut it off after correct number of bits (different for single vs. double precision).
  - Derive Sign, Exponent and Significand fields.
**Hairy Example [1]**

- How to represent 1/3 in IEEE 32-bit?
  - $1/3$
  - $= 0.3333\ldots_{10}$
  - $= 0.25 + 0.0625 + 0.015625 + 0.00390625$
  - $+ 0.0009765625 + \ldots$
  - $= 1/4 + 1/16 + 1/64 + 1/256 + 1/1024 + \ldots$
  - $= 2^{-2} + 2^{-4} + 2^{-6} + 2^{-8} + 2^{-10} + \ldots$
  - $= 0.0101010101\ldots \times 2^{0}$
  - $= 1.0101010101\ldots \times 2^{-2}$

**Hairy Example [2]**

- Sign: 0
- Exponent $= -2 + 127 = 125_{10} = 01111101_2$
- Significand $= 0101010101\ldots$

**Representation for +/- Infinity**

- In FP, divide by zero should produce +/- infinity, not overflow.
- Why?
  - OK to do further computations with infinity
  - e.g., $X/0 > Y$ may be a valid comparison
- IEEE 754 represents +/- infinity
  - Most positive exponent reserved for infinity
  - Significands all zeroes

**Representation for 0**

- Represent 0?
  - exponent all zeroes
  - significand all zeroes too
  - What about sign?
    - +0: 0 00000000 00000000000000000000000
    - -0: 1 00000000 00000000000000000000000
- Why two zeroes?
  - Helps in some limit comparisons
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>Denormalized</td>
</tr>
<tr>
<td>1-254</td>
<td>anything</td>
<td>+/- fl. pt. #</td>
</tr>
<tr>
<td>255</td>
<td>0</td>
<td>+/- infinity</td>
</tr>
<tr>
<td>255</td>
<td>nonzero</td>
<td>???</td>
</tr>
</tbody>
</table>

FP Addition

- Much more difficult than with integers
- Can’t just add significands
- How do we do it?
  - De-normalize to match exponents
  - Add significands to get resulting significand
  - Keep the same exponent
  - Normalize (possibly changing exponent)
- Note: If signs differ, just perform a subtract instead.

FP Subtraction

- Similar to addition
- How do we do it?
  - De-normalize to match exponents
  - Subtract significands
  - Keep the same exponent
  - Normalize (possibly changing exponent)

FP Addition/Subtraction

- Problems in implementing FP add/sub:
  - If signs differ for add (or same for sub), what will be the sign of the result?
- Question: How do we integrate this into the integer arithmetic unit?
- Answer: We don’t!
**Fl. Pt. Architecture [1]**

- Separate floating point instructions:
  - Single Precision:  
    - eg. `add.s`, `sub.s`, `mul.s`, `div.s`
  - Double Precision:  
    - eg. `add.d`, `sub.d`, `mul.d`, `div.d`
- These instructions are far more complicated than their integer counterparts, so they can take much longer.

**Fl. Pt. Architecture [2]**

- Problems:
  - It’s inefficient to have different instructions take vastly differing amounts of time.
  - Generally, a particular piece of data will not change from FP to int, or vice versa, within a program. So only one type of instruction will be used on it.
  - Some programs do no floating point calculations
  - It takes lots of hardware relative to integers to do Floating Point fast

**Fl. Pt. Architecture [3]**

- 1990 Solution: Make a completely separate chip that handles only FP.
- Coprocessor 1: FP chip
  - contains 32 32-bit registers: `$f0$, $f1$, ...
  - most registers specified in `.s` and `.d` instruction refer to this set
  - separate load and store: `lwcl` and `swcl` (“load word coprocessor 1”, “store ...”)
  - Double Precision: by convention, even/odd pair contain one DP FP number: `$f0$/f1, $f2$/f3, ..., $f30$/f31

**Fl. Pt. Architecture [4]**

- 1990+ Computer actually contains multiple separate chips:
  - Processor: handles all the normal stuff
  - Coprocessor 1: handles FP and only FP;
  - more coprocessors?... Yes, later
  - Today, cheap chips may leave out FP HW
- Instructions to move data between main processor and coprocessors:
  - `mfc0`, `mtc0`, `mfcl`, `mtcl`, etc.
- Many, many more FP operations.
Things to Remember

- 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.
- FP registers (e.g., $f0$-$f31$) & instruct.: 
  - Single Precision (32 bits, $2 \times 10^{-38}$... $2 \times 10^{38}$): 
    e.g., `add.s`, `sub.s`, `mul.s`, `div.s`
  - Double Precision (64 bits, $2 \times 10^{-308}$... $2 \times 10^{308}$): 
    e.g., `add.d`, `sub.d`, `mul.d`, `div.d`
- Type is not associated with data, bits have no meaning unless given in context.