Example: Representing 1/3 in MIPS

\[
\frac{1}{3} = 0.33333\ldots_{10} = 0.25 + 0.0625 + 0.015625 + \ldots = 2^{-2} + 2^{-4} + 2^{-6} + 2^{-8} + \ldots
\]

- Sign: 0
- Exponent = -2 + 127 = 125 = 01111101
- Significand = 0101010101…

\[
0.111 \ 1101 \ 0101 \ 0101 \ 0101 \ 0101 \ 0101 \ 0101 \ 0101 \ 0101 \ 0101 \ 0101 \ \text{(2's complement)}
\]

Representation for ±\(\infty\)

- In FP, divide by 0 should produce ±\(\infty\), not overflow.
  - Why?
    - OK to do further computations with \(\infty\)
      E.g., \(X/0 > Y\) may be a valid comparison
    - Ask math majors
  - IEEE 754 represents ±\(\infty\)
    - Most positive exponent reserved for \(\infty\)
    - Significands all zeroes

Representation for Not a Number

- What is \(\sqrt{-4.0}\) 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: \(\text{op}(\text{NaN}, X) = \text{NaN}\)

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>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>???</td>
</tr>
</tbody>
</table>

- Professor Kahan had clever ideas; “Waste not, want not”
  - Exp=0,255 & Sig!=0 ...

Representation for Denorms (1/2)

- Problem: There’s a gap among representable FP numbers around 0
  - Smallest representable pos num:
    \(a = 1.0…, 2^{-126} = 2^{-126}\)
  - Second smallest representable pos num:
    \(b = 1.000…, 1, 2^{-126} = 2^{126} + 2^{149}\)

\[
\begin{align*}
a &= 2^{-126} \\
b &= 2^{-126}
\end{align*}
\]

- Gaps!

\[
\begin{align*}
&\infty \\
&\infty \\
&\infty \\
&\infty \\
\end{align*}
\]

- Normalization and implicit 1 is to blame!
Representation for Denorms (2/2)

- Solution:
  - We still haven’t used Exponent = 0, Significand nonzero
  - Denormalized number: no leading 1, implicit exponent = -126.
  - Smallest representable pos num: \( a = 2^{-149} \)
  - Second smallest representable pos num: \( b = 2^{-148} \)

Overview

- Reserve exponents, significands:
  - Exponent | Significand | Object
  - 0 | 0 | 0
  - 1-254 | anything | +/- fl. pt. #
  - 255 | 0 | +/- \( \infty \)
  - 255 | nonzero | NaN

Rounding

- Math on real numbers \( \Rightarrow \) we worry about rounding to fit result in the significant field.
- FP hardware carries 2 extra bits of precision, and rounds for proper value
- Rounding occurs when converting...
  - double to single precision
  - floating point # to an integer

IEEE Four Rounding Modes

- Round towards + \( \infty \)
  - ALWAYS round “up”: 2.1 \( \Rightarrow \) 3, -2.1 \( \Rightarrow \) -2
- Round towards - \( \infty \)
  - ALWAYS round “down”: 1.9 \( \Rightarrow \) 1, -1.9 \( \Rightarrow \) -2
- Truncate
  - Just drop the last bits (round towards 0)
- Round to (nearest) even (default)
  - Normal rounding, almost: 2.5 \( \Rightarrow \) 2, 3.5 \( \Rightarrow \) 4
  - Like you learned in grade school
  - Insures fairness on calculation
  - Half the time we round up, other half down

Integer Multiplication (1/3)

- Paper and pencil example (unsigned):
  
  \[
  \begin{array}{c|c|c|c|c|c|c|c|c}
  \text{Multiplicand} & 1000 & 8 \\
  \text{Multiplier} & \times & 1001 & 9 \\
  \hline
  1000 & \\
  0000 & \\
  0000 & \\
  +1000 & \\
  \hline
  01001000 & \\
  \end{array}
  \]

  \( m \text{ bits } \times n \text{ bits } = m + n \text{ bit product} \)

Integer Multiplication (2/3)

- In MIPS, we multiply registers, so:
  - 32-bit value x 32-bit value = 64-bit value
- Syntax of Multiplication (signed):
  - \text{mult} register1, register2
  - Multiplies 32-bit values in those registers & puts 64-bit product in special result regs:
    - puts product upper half in hi, lower half in lo
  - hi and lo are 2 registers separate from the 32 general purpose registers
- Use \text{mfhi} register & \text{mflo} register to move from hi, lo to another register
Integer Multiplication (3/3)

• Example:
  • in C: `a = b * c;
  • in MIPS:
    - let b be $s2; let c be $s3; and let a be $s0 and $s1 (since it may be up to 64 bits)
    - mult $s2,$s3 # b*c
      of
    - mthi $s0 # upper half
    - mflo $s1 # lower half of
      product
  • Note: Often, we only care about the lower half of the product.

Integer Division (2/2)

• Syntax of Division (signed):
  • `div register1, register2`
  • Divides 32-bit register 1 by 32-bit register 2:
    • puts remainder of division in $hi$, quotient in $lo$
  • Implements C division (/) and modulo (%)
• Example in C: `a = c / d; b = c % d;`
• in MIPS:
  - `div $s2,$s3 # lo=c/d, hi=cd`
  - `mthi $s0 # get quotient`
  - `mflo $s1 # get remainder`

Integer Division (1/2)

• Paper and pencil example (unsigned):
  ```
  Divisor 100010100100
  -1000
  1010
  -1000
  =10 Remainder
  ```
  (or Modulo result)
• Dividend = Quotient x Divisor + Remainder

Unsigned Instructions & Overflow

• MIPS also has versions of `mul`, `div` for unsigned operands:
  • `multu`
  • `divu`
  • Determines whether or not the product and quotient are changed if the operands are signed or unsigned.
• MIPS does not check overflow on ANY signed/unsigned multiply, divide instr
• Up to the software to check $hi$

MIPS Floating Point Architecture

• Separate floating point instructions:
  • Single Precision:
    • `add.s`, `sub.s`, `mul.s`, `div.s`
  • Double Precision:
    • `add.d`, `sub.d`, `mul.d`, `div.d`
• These are far more complicated than their integer counterparts
• Can take much longer to execute

FP Addition & Subtraction

• Much more difficult than with integers (can’t just add significands)
• How do we do it?
  • De-normalize to match larger exponent
  • Add significands to get resulting one
  • Normalize (check for under/overflow)
  • Round if needed (may need to renormalize)
• If signs ≠, do a subtract. (Subtract similar)
  • If signs ≠ for add (or = for sub), what’s ans sign?
• Question: How do we integrate this into the integer arithmetic unit? [Answer: We don’t!]
MIPS Floating Point Architecture (1)

Problems:
- Inefficient to have different instructions take vastly differing amounts of time.
- Generally, a particular piece of data will not change FP \( \leftrightarrow \) int within a program.
  - Only 1 type of instruction will be used on it.
- Some programs do no FP calculations
- It takes lots of hardware relative to integers to do FP fast

MIPS Floating Point Architecture (2)

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, FP coprocessor integrated with CPU, or cheap chips may leave out FP HW
- Instructions to move data between main processor and coprocessors:
  * mfc0, mtc0, mfc1, mtc1, etc.
- Appendix contains many more FP ops

Peer Instruction 1

1. Let \( X \) = # of floats between 1 and 2
2. Let \( Y \) = # of floats between 2 and 3

Peer Instruction 2

1. Converting float \( \rightarrow \) int \( \rightarrow \) float produces same float number
2. Converting int \( \rightarrow \) float \( \rightarrow \) int produces same int number
3. FP add is associative:
   \((x+y)+z = x+(y+z)\)

“And in conclusion…”

- Reserve exponents, significands:
  - Exponent  Significand  Object
  - 0 0 0 nonzero Denorm
  - 1-254 anything +/- fl. pt. #
  - 255 0 +/- \( \infty \)
  - 255 nonzero NaN
- Integer mult, div uses hi, lo regs
- mfhi and mflo copies out.
- Four rounding modes (to even default)
- MIPS FL ops complicated, expensive