Example: Representing 1/3 in MIPS

\[
1/3 = 0.33333_{10} = 0.25 + 0.0625 + 0.015625 + 0.00390625 + \ldots = 2^{-2} + 2^{-4} + 2^{-6} + 2^{-8} + \ldots = 0.0101010101_{2} * 2^{0} = 1.0101010101_{2} * 2^{-2}
\]

\[
\text{Sign: } 0, \quad \text{Exponent } = -2 + 127 = 125 = 01111101, \quad \text{Significand } = 0101010101_{2}
\]

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 \(\pm \infty\)
  - Most positive exponent reserved for \(\infty\)
  - Significands all zeroes

Representation for 0

- Represent 0?
  - Exponent all zeroes
  - Significand all zeroes too
  - What about sign?
    - +0: 0 0000000 00000000000000000000000
    - -0: 1 0000000 00000000000000000000000
- Why two zeroes?
  - Helps in some limit comparisons
  - Ask math majors

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>+/- (\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 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: op(NaN, X) = NaN
**Representation for Denoms (1/2)**

- **Problem:** There's a gap among representable FP numbers around 0
  - Smallest representable pos num: \(a = 1.0 \ldots 2 \cdot 2^{-126}\)
  - Second smallest representable pos num: \(b = 1.000 \ldots 1 \cdot 2^{-126} = 2^{-126} + 2^{-149}\)
  
  \[
  a - 0 = 2^{-126} \\
  b - a = 2^{-149}
  \]

  **Gaps!**

  Normalization and implicit 1 is to blame!

  **RQ answer!**

**Representation for Denoms (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: \(b = 2^{148}\)

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

  **RQ answer!**

**Rounding**

- Math on real numbers ⇒ we worry about rounding to fit result in the significant field. **RQ answer!**
- 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 ⇒ 3, -2.1 ⇒ -2
- Round towards \(-\infty\)
  - ALWAYS round “down”: 1.9 ⇒ 1, -1.9 ⇒ -2
- Truncate
  - Just drop the last bits (round towards 0)
- Round to (nearest) even (default)
  - Normal rounding, almost: 2.5 ⇒ 2, 3.5 ⇒ 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):
  
  \[
  \begin{array}{c}
  \text{Multiplicand} \\
  1000 \\
  \times 9
  \end{array}
  \begin{array}{c}
  100 \\
  000 \\
  9000 \\
  10000
  \end{array}
  \]

  \[
  \begin{array}{c}
  m \text{ bits} \times n \text{ bits} = m + n \text{ bit product}
  \end{array}
  \]

**Integer Multiplication (2/3)**

- In MIPS, we multiply registers, so:
  - 32-bit value x 32-bit value = 64-bit value
- Syntax of Multiplication (signed):
  - 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 mfhi register & mflo register to move from hi, lo to another register
### Integer Multiplication (3/3)

- **Example:**
  - In C: \(a = b \times c;\)
  - 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)
    - \text{mult} \, s2, s3 \quad \# \text{b} \times \text{c}
    - \text{mfhi} \, s0 \quad \# \text{upper half of product into } s0
    - \text{mflo} \, s1 \quad \# \text{lower half of product into } s1
  - **Note:** Often, we only care about the lower half of the product.

### Integer Division (1/2)

- **Paper and pencil example (unsigned):**
  
  \[
  \begin{array}{c|c}
  \text{Divisor} & 1000 \\
  \text{Dividend} & 10101010 \\
  \text{Quotient} & 101 \\
  \text{Remainder} & 1000 \\
  \end{array}
  \]

- **Example in C:** 
  \[a = c / d;\]
  \[b = c \mod d;\]

### Integer Division (2/2)

- **Syntax of Division (signed):**
  - \text{div} \, \text{reg1}, \text{reg2}
  - Divides 32-bit register 1 by 32-bit register 2:
    - puts remainder of division in hi, quotient in lo
  - **Example in C:** 
    
  - In MIPS:
    - \text{div} \, s2, s3 \quad \# \text{c} / \text{d}
    - \text{mfhi} \, s0 \quad \# \text{get quotient}
    - \text{mflo} \, s1 \quad \# \text{get remainder}

### Unsigned Instructions & Overflow

- MIPS also has versions of \text{mult}, \text{div} for unsigned operands:
  - \text{mulu}
  - \text{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

### 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 (1/4)

- Separate floating point instructions:
  - Single Precision:
    - \text{add.s}, \text{sub.s}, \text{mul.s}, \text{div.s}
  - Double Precision:
    - \text{add.d}, \text{sub.d}, \text{mul.d}, \text{div.d}

  - These are far more complicated than their integer counterparts
    - Can take much longer to execute
MIPS Floating Point Architecture (2/4)

• 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 (3/4)

• 1990 Solution: Make a completely separate chip that handles only FP.
  • Coprocessor 1: FP chip
    - contains 32 32-bit registers: $f0, f1, ...$
    - most of the registers specified in .s and .d instruction refer to this set
    - separate load and store: lwc1 and swc1 (“load word coprocessor 1”, “store ...”)
    - Double Precision: by convention, even/odd pair contain one DP FP number: $f0/f1, f2/f3, ..., f30/f31$
      - Even register is the name

MIPS Floating Point Architecture (4/4)

• 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, 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. 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: $(x+y)+z = x+(y+z)$

As Promised, the way to remember #s

• What is $2^{34}$? How many bits addresses (i.e., what’s ceil log₂ = log₁₀ of) 2.5 TB?
  • Answer! $2^{XY}$ means...
    X=0 $\Rightarrow$ 0  Y=0 $\Rightarrow$ 1
    X=1 $\Rightarrow$ Kilo $\sim$ 10³  Y=1 $\Rightarrow$ 2
    X=2 $\Rightarrow$ Mega $\sim$ 10⁶  Y=2 $\Rightarrow$ 4
    X=3 $\Rightarrow$ Giga $\sim$ 10⁹  Y=3 $\Rightarrow$ 8
    X=4 $\Rightarrow$ Terra $\sim$ 10¹²  Y=4 $\Rightarrow$ 16
    X=5 $\Rightarrow$ Peta $\sim$ 10¹⁵  Y=5 $\Rightarrow$ 32
    X=6 $\Rightarrow$ Exa $\sim$ 10¹⁸  Y=6 $\Rightarrow$ 64
    X=7 $\Rightarrow$ Zetta $\sim$ 10²¹  Y=7 $\Rightarrow$ 128
    X=8 $\Rightarrow$ Yotta $\sim$ 10²⁴ Y=8 $\Rightarrow$ 256
    X=9 $\Rightarrow$ 512

MEMORIZE!

“And in conclusion…”

• 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>+/- infinite</td>
</tr>
<tr>
<td>255</td>
<td>nonzero</td>
<td>NaN</td>
</tr>
</tbody>
</table>

• Integer multi, div uses hi, lo regs
  • mfhi and mflo copies out.

• Four rounding modes (to even default)

• MIPS FL ops complicated, expensive