Divide: Paper & Pencil

\[
\begin{array}{c|c}
\text{Divisor} & 1001 \\
1000 & \text{Quotient} \\
\hline
1001010 & \text{Dividend} \\
\hline
-1000 & \\
10 & \\
101 & \\
1010 & \\
-1000 & \\
10 & \text{Remainder}
\end{array}
\]

- See how big a number can be subtracted, creating quotient bit on each step
  - Binary \(\Rightarrow 1 \times \text{divisor}\) or \(0 \times \text{divisor}\)
- Dividend = Quotient \(\times\) Divisor + Remainder

DIVIDE HARDWARE

Version 1

- 64-bit Divisor reg, 64-bit ALU, 64-bit Remainder reg, 32-bit Quotient reg

CSE 141 Allan Snavey
Divide Algorithm
Version 1
• Takes $n+1$ steps for $n$-bit Quotient & Rem.

<table>
<thead>
<tr>
<th>Quotient</th>
<th>Divisor</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0011</td>
<td>0000</td>
</tr>
</tbody>
</table>

1. Subtract the Divisor register from the Remainder register, and place the result in the Remainder register.

Test Remainder

Remainder $\geq 0$

2a. Shift the Quotient register to the left setting the new rightmost bit to 1.

Remainder $< 0$

2b. Restore the original value by adding the Divisor register to the Remainder register, and place the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0.

3. Shift the Divisor register right 1 bit.

3rd repetition?

No: < 33 repetitions

Yes: 33 repetitions

Done

DIVIDE HARDWARE
Version 3
• 32-bit Divisor reg, 32-bit ALU, 64-bit Remainder reg, (0-bit Quotient reg)
Observations on Divide
Version 3

• Same Hardware as Multiply: just need ALU to add or subtract, and 63-bit register to shift left or shift right
• Hi and Lo registers in MIPS combine to act as 64-bit register for multiply and divide
• Signed Divides: Simplest is to remember signs, make positive, and complement quotient and remainder if necessary
  – Note: Dividend and Remainder must have same sign
  – Note: Quotient negated if Divisor sign & Dividend sign disagree

So Far

• Can do logical, add, subtract, multiply, divide, ...
• But........
  – what about fractions?
  – what about really large numbers?
Binary Fractions

1011_2 = 1x2^3 + 0x2^2 + 1x2^1 + 1x2^0
so...
101.011_2 = 1x2^2 + 0x2^1 + 1x2^0 + 0x2^{-1} + 1x2^{-2} + 1x2^{-3}
e.g.,
.75 = 3/4 = 3/2^2 = 1/2 + 1/4 = .11

Recall Scientific Notation

Issues:
° Arithmetic (+, -, *, /)
° Representation, Normal form
° Range and Precision
° Rounding
° Exceptions (e.g., divide by zero, overflow, underflow)
° Errors
° Properties (negation, inversion, if A = B then A - B = 0)
Floating-Point Numbers

Representation of floating point numbers in IEEE 754 standard:

### Single Precision

<table>
<thead>
<tr>
<th>sign</th>
<th>8</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exponent:</td>
<td>excess 127 binary integer</td>
<td></td>
</tr>
<tr>
<td>Mantissa:</td>
<td>sign + magnitude, normalized binary significand w/ hidden integer bit: 1.M</td>
<td></td>
</tr>
</tbody>
</table>

(actual exponent is $e = E - 127$)

$$N = (-1)^S 2^{E-127} (1.M)$$

$0 = 0.00000000 \ldots 0$  
$-1.5 = 1.01111111 10 \ldots 0$  
$325 = 101000101 \times 2^8 = 1.01000101 \times 2^8$  
$.02 = .0011001101100 \ldots X 2^8 = 1.1001101100 \ldots X 2^8$  
$.02 = 0.1111100 1001101100 \ldots$  

- range of about $2 \times 10^{-38}$ to $2 \times 10^{38}$
- always normalized (so always leading 1, thus never shown)
- special representation of 0 (E = 00000000) (why?)
- can do integer compare for greater-than, sign

Double Precision Floating Point

Representation of floating point numbers in IEEE 754 standard:

<table>
<thead>
<tr>
<th>sign</th>
<th>11</th>
<th>20</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exponent:</td>
<td>excess 1023 binary integer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mantissa:</td>
<td>sign + magnitude, normalized binary significand w/ hidden integer bit: 1.M</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(actual exponent is $e = E - 1023$)

$$N = (-1)^S 2^{E-1023} (1.M)$$

- 52 (+1) bit mantissa
- range of about $2 \times 10^{-308}$ to $2 \times 10^{308}$
Floating Point Addition

• How do you add in scientific notation?
  \[9.962 \times 10^4 + 5.231 \times 10^2\]

• Basic Algorithm
  1. Align
  2. Add
  3. Normalize
  4. Round
Floating Point Multiplication

- How do you multiply in scientific notation?
  \[(9.9 \times 10^4)(5.2 \times 10^2) = 5.148 \times 10^7\]

- Basic Algorithm
  1. Add exponents
  2. Multiply
  3. Normalize
  4. Round
  5. Set Sign

FP Accuracy

- Extremely important in scientific calculations
- Very tiny errors can accumulate over time
- IEEE 754 FP standard has four rounding modes
  - always round up (toward \(\pm \infty\))
  - always round down (toward \(-\infty\))
  - truncate
  - round to nearest
    \(\Rightarrow\) in case of tie, round to nearest even
- Requires extra bits in intermediate representations
Extra Bits for FP Accuracy

- *Guard bits* -- bits to the right of the least significant bit of the significand computed for use in normalization (could become significant at that point) and rounding.
- IEEE 754 has three extra bits and calls them *guard*, *round*, and *sticky*.

Key Points

- Multiplication and division take much longer than addition, requiring multiple addition steps.
- Floating Point extends the range of numbers that can be represented, at the expense of precision (accuracy).
- FP operations are very similar to integer, but with pre- and post-processing.
- Rounding implementation is critical to accuracy over time.