Divide: Paper & Pencil

\[
\begin{align*}
\text{Divisor} & \quad 1001010 \\
\text{Quotient} & \quad 1000 \\
\text{Dividend} & \quad 1001 \\
\text{Remainder (or Modulo result)} & \quad 1010
\end{align*}
\]

See how big a number can be subtracted, creating quotient bit on each step
Quotient bit = 1 if can be subtracted, 0 otherwise
Dividend = Quotient x Divisor + Remainder

3 versions of divide, successive refinement

Divide algorithm

- Main ideas:
  - Expand both divisor and dividend to twice their size
    - Expanded divisor = divisor (half bits, MSB) zeroes (half bits, LSB)
    - Expanded dividend = zeroes (half bits, MSB) dividend (half bits, LSB)
  - At each step, determine if divisor is smaller than dividend
    - Subtract the two, look at sign
    - If \(\geq 0\): dividend/divisor\(\geq 1\), mark this in quotient as "1"
    - If negative: divisor larger than dividend; mark this in quotient as "0"
  - Shift divisor right and quotient left to cover next power of two
  - Example: 7/2

DIVIDE HARDWARE Version 1

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

\[
\begin{align*}
\text{Divisor} & \quad 0s \\
64\text{-bit ALU} & \quad 64\text{ bits} \\
\text{Remainder} & \quad 0s \quad \text{Divid} \quad 64\text{ bits} \\
\text{Quotient} & \quad \text{Shift Left} \\
\text{Shift Right} & \quad \text{Write} \\
\text{Control} & \quad 32\text{ bits}
\end{align*}
\]
Divide Algorithm Version 1: 7/2

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

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

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

3. Shift the Divisor register right 1 bit.

Done

n+1 repetition?

No: < n+1 repetitions

Yes: n+1 repetitions (n = 4 here)

Start: Place Dividend in Remainder

Remainder Quotient Divisor
0000 0111 0000 0010 0000

Remainder >= 0 Test Remainder

Remainder < 0

Observations on Divide Version 1

° 1/2 bits in divisor always 0
  => 1/2 of 64-bit adder is wasted

° Instead of shifting divisor to right, shift remainder to left?

° 1st step will never produce a 1 in quotient bit (otherwise too big)
  => switch order to shift first and then subtract, can save 1 iteration

Divide Algorithm Version 1:
7 (0111) / 2 (0010) = 3 (0011) R 1 (0001)

Step Remainder Quotient Divisor Rem-Div

Initial 0000 0111 0000 0010 0000 < 0
1 0000 0111 0000 0001 0000 < 0
2 0000 0111 0000 0000 1000 < 0
3 0000 0111 0000 0000 0100 0000 0011 > 0
4 0000 0011 0001 0000 0010 0000 0001 > 0
5 0000 0001 0011 0000 0001

Final 1 3

First Rem-Dev always < 0
Always 0
DIVIDE HARDWARE Version 2

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

DIVIDE HARDWARE Version 3

- 32-bit Divisor reg, 32-bit ALU, 64-bit Remainder reg, (0-bit Quotient reg)

Observations on Divide Version 2

- Eliminate Quotient register by combining with Remainder as shifted left
  - Start by shifting the Remainder left as before.
  - Thereafter loop contains only two steps because the shifting of the Remainder register shifts both the remainder in the left half and the quotient in the right half
  - The consequence of combining the two registers together and the new order of the operations in the loop is that the remainder will shifted left one time too many.
  - Thus the final correction step must shift back only the remainder in the left half of the register

Divide Algorithm Version 2

1. Shift the Remainder register left 1 bit
2. Subtract the Divisor register from the left half of the Remainder register, & place the result in the left half of the Remainder register.
3. Test Remainder
   - Remainder >= 0
   - Remainder < 0
3a. Shift the Quotient register to the left setting the new rightmost bit to 1.
3b. Restore the original value by adding the Divisor register to the left half of the Remainder register, & place the sum in the left half of the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0.

nth repetition?
- No: < n repetitions
- Yes: n repetitions (n = 4 here)
Done
Divide Algorithm Version 3

1. Shift the Remainder register left 1 bit.
2. Subtract the Divisor register from the left half of the Remainder register, & place the result in the left half of the Remainder register.
3a. Shift the Remainder register to the left setting the new rightmost bit to 1.
3b. Restore the original value by adding the Divisor register to the left half of the Remainder register, & place the sum in the left half of the Remainder register. Also shift the Remainder register to the left, setting the new least significant bit to 0.

Start: Place Dividend in Remainder

Step | Remainder | Divisor | Rem-Div
--- | --- | --- | ---
Initial | 0000 0111 | 0010 | Always < 0
Shift | 0000 1110 | 0010 | < 0
1 | 0001 1100 | 0010 | < 0
2 | 0011 1000 | 0010 | 0011-0010 > 0
2 | 0001 1000 | 0010 | 
3 | 0011 0001 | 0010 | 0011-0010 > 0
3 | 0001 0001 | 0010 | 
4 | 0010 0011 | 0010 | 
Final | R1 3 |

Done. Shift left half of Remainder right 1 bit.

Observations on Divide Version 3

- Same Hardware as Multiply: just need ALU to add or subtract, and 64-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 e.g., -7 ÷ 2 = -3, remainder = -1

Floating-Point

- What can be represented in N bits?
  - Unsigned: 0 to 2^N
  - 2's Complement: -2^(N-1) to 2^(N-1)-1
- Integer numbers useful in many cases; must also consider “real” numbers with fractions
  - E.g. 1/2 = 0.5
  - Very large: 9,349,398,989,000,000,000,000,000,000,000,000
  - Very small: 0.0000000000000000000000000000045691
Recall Scientific Notation

- **Decimal point**
- **Exponent**
- **Radix (base)**
- **Mantissa**

| 6.02 x 10^23 | 1.673 x 10^-24 |

**Issues:**
- Arithmetic (+, -, *, /)
- Representation, normalized form (e.g., x.xxx * 10^y)
- Range and Precision
- Rounding
- Exceptions (e.g., divide by zero, overflow, underflow)
- Errors

---

**Normalised notation using powers of two**

- **Base 10:** single non-zero digit left of the decimal point.
- **Base 2:** normalized numbers can also be represented as: 1.xxx * 2^y, where x and y are binary
- **Example:** -0.75
  - -75/100, or -3/4
  - -3 in binary: -11.0
  - Divided by 4 -> binary point moves left two positions, -0.11
  - Normalized: -1.1 * 2^(-1)

---

*Review from Prerequisites: Floating-Point Arithmetic*

**Representation of floating point numbers in IEEE 754 standard:**

<table>
<thead>
<tr>
<th>Single precision</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sign</strong></td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

- Actual exponent is  e = E - 127 (bias)
- Normalized: ± 1.M * 2^e

<table>
<thead>
<tr>
<th>N = (-1)^S * 2^E * 1.M</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 &lt; E &lt; 255 (bias makes &lt; &gt; comparisons easy)</td>
</tr>
</tbody>
</table>

**Example:** -0.75
- -1.1 * 2^(-1)
- Sign bit: 1
- Exponent: e-127=-1 so e=126 (01111110)
- Mantissa: 1000...00 (Remember, for 1.x, the 1 is implicit so not in M)
- Single-precision representation: 1011111101000...00

**Single- and double-precision**

- **Single-precision:** 32 bits
  - (sign + 8 exponent + 23 fraction)
- **Double-precision:** 64 bits
  - (sign + 11 exponent + 52 fraction)
- Increases reach of large/small numbers by 3 powers, but most noticeable improvement is in the number of bits used to represent fraction

**Example:** -0.75
- -1.1 * 2^(-1)
- Sign bit: 1
- Exponent: e-127=-1 so e=126 (01111110)
- Mantissa: 1000...00 (Remember, for 1.x, the 1 is implicit so not in M)
- Single-precision representation: 1011111101000...00

Magnitude of numbers that can be represented is in the range:

- 2^-126 (1.0) to 2^127 (2 - 2^23)

which is approximately:

- 1.8 x 10^-38 to 3.4 x 10^38

(integer comparison valid on IEEE Fl.Pt. numbers of same sign)
Operations with floating-point numbers

° Addition/subtraction:
  • Need to have both operands with the same exponent
    - “small” ALU calculates exponent difference
    - Shift number with smaller exponent to the right
  • Add/subtract the mantissas

° Multiplication/division
  • Add/subtract the exponents
  • Multiply/divide mantissas

° Normalize, round, (re-normalize)

Addition example

° 99.99 + 0.161
° Scientific notation, assume only 4 digits can be stored
  • 9.999E+1, 1.610E-1
° Must align exponents:
  • 1.610E-1 = 0.0161E+1
° Can only represent 4 digits: 0.016E+1
° Sum: 10.015E+1
° Not normalized; adjust to 1.0015E+2
° Can only represent 4 digits; must round (0 to 4 down, 5 to 9 up)
  • 1.002E+2
° It can happen that after rounding result is no longer normalized
  • E.g. if the sum was 9.9999E+2, normalize again
Multiplication

Example: 1.110E10 * 9.200E-5

Add exponents: 10 + (-5) = 5

- Remember: in IEEE format, the number stored in the FP bits is "e", but the actual exponent is (e-127) (subtract the bias). To compute the exponent of the result, you have to add the "e" bits from both operands, and then subtract 127 to adjust
- E.g. exponent +10 is stored as 137; -5 as 122
- 137+122 = 259
- 259-127 = 132, which represents exponent +5

Multiply significands

1.110*9.200 = 10.212000

Normalize: 1.0212E+6

Check exponent for overflow (too large positive exponent) and underflow (too large negative exponent)

Round to 4 digits: 1.021E+6

Infinity and NaNs

result of operation overflows, i.e., is larger than the largest number that can be represented

overflow (too large of an exponent) is not the same as divide by zero
Both generate +/-Inf as result; but raise different exceptions

+/− infinity [S] 1 . . . 1 0 . . . 0

It may make sense to do further computations with infinity
e.g., X=Inf > Y may be a valid comparison

Not a number, but not infinity (e.q. sqrt(-4))
invalid operation exception (unless operation is = or ≠)

NaN [S] 1 . . . 1 non-zero

NaNs propagate: f(NaN) = NaN

Guard, round and sticky bits

# of bits in floating-point fraction is fixed
- During an operation, can keep additional bits around to improve precision in rounding operations
- Guard and round bits are kept around during FP operation and used to decide direction to round

Sticky bits: flag whether any bits that are not considered in an operation (they have been shifted right) are 1

Can be used as another factor to determine the direction of rounding
Guard and round bits

- E.g. $2.56 \times 10^0 + 2.34 \times 10^2$
- 3 significant decimal digits
- With guard and round digits:
  - $2.3400 +$
  - $0.0256$
  - --------
  - $2.3656$
  - 0 to 49: round down, 50 to 99: round up -> 2.37
- Without guard and round digits:
  - $2.34 +$
  - $0.02$
  - ------
  - $2.36$

Floating-point in MIPS

- Use different set of registers
  - 32 32-bit floating point registers, $f0 - f31$
- Individual registers: single-precision
- Two registers can be combined for double-precision
  - $f0$ ($f0, f1), f2 ($f2, f3)$
- add, sub, mul, div
  - .s for single, .d for double precision
- Load and store memory word to 32-bit FP register
  - Lwcl, swcl (cl refers to co-processor 1 when separate FPU used in past)
- Instructions to branch on floating point conditions (e.g. overflow), and to compare FP registers

Floating-point in x86

- First introduced with 8087 FP co-processor
- Primarily a stack architecture:
  - Loads push numbers into stack
  - Operations find operands on two top slots of stack
  - Stores pop from stack
  - Similar to HP calculators 2+3 -> 23+
- Also supports one operand to come from either FP register below top of stack, or from memory
- 32-bit (single-precision) and 64-bit (double-precision) support

Floating point in x86

- Data movement:
  - Load, load constant, store
- Arithmetic operations:
  - Add, subtract, multiply, divide, square root
- Trigonometric/logarithmic operations
  - Sin, cos, log, exp
- Comparison and branch
SSE2 extensions

° Streaming SIMD extension 2
  • Introduced in 2001
  • SIMD: single-instruction, multiple data
  • Basic idea: operate in parallel on elements within a wide word
    - e.g. 128-bit word can be seen as 4 single-precision FP numbers, or 2 double-precision

° Eight 128-bit registers
  • 16 in the 64-bit AMD64/EM64T

° No stack – any register can be referenced for FP operation

Differences between x86 FP approaches

° 8087-based:
  • Registers are 80-bit (more accuracy during operations); data is converted to/from 64-bit when moving to/from memory
  • Stack architecture
  • Single operand per register

° SSE2:
  • Registers are 128-bit
  • Register-register architecture
  • Multiple operands per register

° Differences in internal representation can cause differences in results for the same program
  • 80-bit representation used in operations
  • Truncated to 64-bit during transfers
  • Differences can accumulate, effected by when loads/stores occur

Floating point operations

° Number of bits is limited and small errors in individual FP operations can compound over large iterations
  • Numerical methods that perform operations such as to minimize accumulation of errors are needed in various scientific applications

° Operations may not work as you would expect
  • E.g. floating-point add is not always associative
  • \[ x + (y+z) = (x+y) + z \] ?
  • \[ x = -1.5 \times 10^{38}, y = 1.5 \times 10^{38}, z = 1.0 \]
  • \[ (x+y) + z = (-1.5 \times 10^{38} + 1.5 \times 10^{38}) + 1.0 = (0.0) + 1.0 = 1.0 \]
  • \[ x + (y+z) = -1.5 \times 10^{38} + (1.5 \times 10^{38} + 1.0) = -1.5 \times 10^{38} + 1.5 \times 10^{38} = 0.0 \]

  \[ 1.5 \times 10^{38} \text{ is so much larger than 1, that sum is just } 1.5 \times 10^{38} \text{ due to rounding during the operation } \]

Summary

° Bits have no inherent meaning: operations determine whether they are really ASCII characters, integers, floating point numbers
° Divide can use same hardware as multiply: Hi & Lo registers in MIPS
° Floating point basically follows paper and pencil method of scientific notation using integer algorithms for multiply and divide of significands
° IEEE 754 requires good rounding; special values for NaN, Infinity