Back to Arithmetic

Before, we did

• Representation of integers
• Addition/Subtraction
• Logical ops

Forecast

• Integer Multiplication
• Integer Division
• Floating-point Numbers
• Floating-point Addition/Multiplication

Integer Multiplication

Recall decimal multiplication from grammar school (non negative)

multiplicand 1000 base ten
multiplier 1001 base ten
partial 1000
products

0000

1000

1001000 base ten

Example (Fig 4.25)

Use carry-save adders in a wallace tree

n bits times m bits = n+m bits (32 + 32 = 64)

Example next (Figure 4.27)

• Multiplicand = 2 = 0010
• Multiplier = 3 = 0011
• Product = 6 = 0110
Example (Fig. 4.26)

1. Test Multiplier
   1a. Add multiplicand to product and place the result in Product register
   2. Shift the Multiplicand register left 1
   3. Shift the Multiplier register right 1

32nd repetition?

Start

Multiplier0 = 0
Multiplier0 = 1

No:  < 32 repetition
Yes:  32 repetitions

Integer Multiplication

Two optimizations
- observation: upper-half of 64 bits are all zero
- use 32-bit ALU and shift product right
- instead of multiplicand left (multiplier still goes right)
- observation: only half of product is used
- put multiplier in not-yet-used part of product

Integer Multiplication

Combinational multiplier

1000 * 1001

1000

1 1000 AND bits to get partial products
0 0000 ADD PPs in tree to get product
0 0000 Use carry-save addition: 3 to 2 reduction every step
1 1000

1001000

Grammar school
- sign-prod = sign-mplicand XOR sign-mplier; negative = 0
- if multiplicand < 0 {multiplicand = -multiplicand; negative++}
- if multiplier < 0 {multiplier = -multiplier; negative++}
- product = multiplicand\*multiplier
- if negative == 1 product = -product

Booth’s encoding
Integer Multiplication

Booth encoding -- mind bending like carry-lookahead

Skipping over 9's in decimal - look for beginning and end of 9's

\[ 12345 \]
\[ \times \quad 09990 \quad = -10 + 10000 \]
\[ -123450 \quad 12345*10 \]
\[ +123450000 \quad 12345*10000 \]
\[ 123326550 \]

But in decimal only works for 9's - 1 less than base (10)

Booth's Encoding

-2\(k\) 1k 512 256 128 64 32 16 8 4 2 1 0

1 1 1 0 0 1 1 1 0 1 0 0 0
0 0 -1 0 +1 0 0 -1 +1 -1 0 0 0
0 -2 +2 -1 +1 0

-2048 + 1024 + 512 + 64 + 32 + 16 + 4

\(-1\times512 + 1\times128 + -1\times16 + 1\times8 + -1\times4\)

\(-2\times256 + 2 \times 64 + -1\times16 + 1\times4\)

all equivalent

Booth's Encoding

In binary

• works for 1’s - 1 less than 2
• we already are fast on zeroes

<table>
<thead>
<tr>
<th>current bit</th>
<th>bit to right</th>
<th>info</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>start 1’s</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>middle of 1’s</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>end of 1’s</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>middle of 0’s</td>
</tr>
</tbody>
</table>

Booth Encoding

1010 -> -6 8 bits = 11111010 -(-6) = 00000110
0110 -> +6 Boothenc = +1 0 -1 - = +0-0
11111010 *0 = 0
1111010_ *- = 00001100
11010_ *0 = 0
11010 *+ = 11010000
11011100 = -36
Booth Encoding

negative multiplier

1010 = -6

1110 = -2  booth enc 000-0

0000110_ = 00001100 = +12

\[ b \ast a_2 a_1 a_0 = \]
\[ (a_1 \ast a_2) \ast b \ast 2^2 + (a_0 \ast a_1) \ast b \ast 2^1 + (0-a_0) \ast b \ast 2^0 \]
\[ -a_2 \ast b \ast 2^2 + (2 \ast a_1 \ast a_1) \ast b \ast 2^1 + (2 \ast a_0 \ast a_0) \ast b \ast 2^0 \]
\[ [a_2 \ast -2^2 + a_1 \ast 2^1 + a_0 \ast 2^0] \]

Redundant Representations

Normally

- \[ d_2 \ast b^2 + d_1 \ast b^1 + d_0 \ast b_0; \] b base, \( d_i \) usually \( 0, 1, \ldots \) base-1

Booth Encoding

- \( b = 2, d_i = \{-1, 0, +1\} \)

Carry-Save addition

- \( b = 2, d_i = \{0, 1, 2, 3\} \)

2-bit Booth Encoding

- \( b = 4, d_i = \{-2, -1, 0, +1, +2\} \)

2-bit Booth Encoding

n-bit encoding retires n multiplier bits at a time

Eg.,

1 1 1 0 0 1 1 1 0 1 0 0 “0”

0 0 -1 0 +1 0 0 -1 +1 -1 0 0 -------- 1 bit enc

0 0 -2 +2 -1 +1 0 -------- 2 bit enc

2-bit Booth Encoding

For each partial product, mux controlled by multiplier digits

-2 - 2’sC, shift left one bit

-1 - 2’sC

0

+1 pass through

+2 shift left one bit
Integer Division

Integer Division

divisor - 1000  dividend 1001010 - grammar school
1000)(1001010(1001 - quotient

1000
10
101
1010
1000
10 - remainder

But hardware can't inspect to see if divisor fits, so
Subtract
• if non-negative then set quotient to 1
• else set quotient to 0, add back the divisor (or “restore”)

Figure
Can do multiplication-like optimizations

Non-restoring division - a key optimization in division
Recall restoring division:
divisor 1000, 2'sC 1 . . . 11000

0010101
+ 11000  -divisor*2^2
11101 => < 0
+01000  +divisor*2^2
00101
001010  next bit down
+111000  -divisor*2^2
000010  (-d*2^2 + d*2^2) - d*2^1
**Integer Division**

Now non-restoring

0010101

+ 11000  \(-\text{divisor} \times 2^2\)

11101  \(\Rightarrow < 0\)

111010  next bit down

+001000  \(+\text{divisor} \times 2^1\)

000010  \((-\text{d} \times 2^2 + \text{d} \times 2^1) = -\text{d} \times 2^1\)

---

**Integer Division**

But quotient bits are \{1, \(\overline{1}\)\}

quotient bit = 1 if partial remainder is \(\geq 0\) (i.e., subtract)

quotient bit = \(\overline{1}\) if partial remainder is \(< 0\) (i.e., add)

convert the weird quotient into 2’s complement

for any 2’s complement negative numbers:

quotient bit = 1 if partial remainder and divisor are same sign

quotient bit = \(\overline{1}\) if partial remainder and divisor are opposite sign

---

**SRT Division and Pentium Bug**

Normalize so 1 \(\leq\) dividend, divisor < 2

Use radix 4 for divisor

- base = 2
- get 2 bits of quotient per iteration

Use redundant quotient representation

- digits \{-2, -1, 0, +1, +2\} instead of \{0,1,2,3\}

---

**Pentium Bug**

partial-remainder = dividend

loop {
  \cdot determine next quotient digit
  \cdot subtract quotient-digit \times divisor from partial-remainder (CSA)
  \cdot shift over 2 bits (radix 4)
}

---
Pentium Bug

Determine next quotient digit
conceptually - a table-lookup into table[partial-remainder, divisor]
guess next 2 quotient bits
some part of the table is not “accessible”
so optimized as don’t cares in PLA

But some of the don’t cares (5) actually occur in practice!

Pentium Bug

Incomplete testing did not expose,
• since the algorithm self-corrects
• as long as the partial-remainder is “in range”
incorrect quotient for some dividend, divisor pairs
1.14*10^{-10} fail on random
Max error in 5th significant digit,
• because you can’t get out of range for many iterations

Booth 2-bit Encoding

<table>
<thead>
<tr>
<th>curr bits</th>
<th>bit to right</th>
<th>info</th>
<th>Op</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>mid of 0’s</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>end of 1’s</td>
<td>+1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>single 1</td>
<td>+1</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>end of 1’s</td>
<td>+2</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>beg of 1’s</td>
<td>-2</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>single 0</td>
<td>-1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>beg of 1’s</td>
<td>-1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>mid of 1’s</td>
<td>0</td>
</tr>
</tbody>
</table>
Non-restoring Division

Final step may need correction if
- remainder and dividend opp signs, correction needed
- dividend, divisor same sign, remainder += D, quotient -=ulp
- dividend, divisor opp sign, emainder -= D, quotient +=ulp

convert weird quotient to 2'sC : 1 is 1, Ŵ is 0
shift left by one bit
complement MSB
shift 1 into LSB

Floating-Point Numbers

want to represent real numbers
But uncountably infinite
Recall scientific notation
- $3.15576 \times 10^9$ (#seconds in a century!)
- 3,155,760,000
- exponent says where the decimal point “float”
Recall normalization
- use $3.14 \times 10^{10}$ NOT $0.314 \times 10^{11}$ or $31.4 \times 10^9$
- MSD is [1,9] except for 0.0

Floating-Point Numbers

computer floating-point is similar except binary
- number is $-1^s \times f \times 2^e$ (note base is not stored)
- IEEE 754 uses base 2
  - reduce relative error (wobble)
  - most significant bit is always 1, so don’t store it

For IEEE FP, store s, e,f as S, E, F
- S E F range n bias
- 1 8 23 single-precision $2^{*10^+/-38}$ 23 127
- 1 11 52 double-precision $2^{*10^+/-308}$ 52 1023
Floating-Point Numbers

Exceptions

- $S$ $E$ $F$ number
- 0 0 0 0
- 0 max 0 +inf
- 1 max 0 -inf
- $x$ max !=0 NaN
- $x$ 0 !=0 denorm $f = 0 + F/2^n$

see book for table

Floating-Point Addition

Like scientific notation

$9.997 \times 10^2$

+$4.631 \times 10^{-1}$

First step: align decimal points, second step: add

$9.997 \times 10^2$ + $0.004631 \times 10^2$

$10.001631 \times 10^2$

Floating-Point Subtraction

Subtraction similar

- when adding different signs
- subtracting same signs

Floating-Point Addition

Third step: normalize the result

- often already normalized
- otherwise move only one digit

$1.0001631 \times 10^3$

Example presumes infinite precision; with FP must round

Figure
Floating-Point Multiplication

Example:

- $3.0 \times 10^1$
- $5.0 \times 10^2$
- algorithm: multiple mantissas, add exponents
- check exponent in bounds --> exception
- normalize (and round)
- set sign

Floating-Point Multiplication

Hardware: Figure
Exponent:

$e_+ = e_1 + e_2$
$E_+ = e_+ + 1023 = E_1 - 1023 + E_2 - 1023 + 1023$
$E_+ = E_1 + E_2 - 1023$

$-1023 = -(1111111111) = 0000000000 + 1 = +1$

With 2's C $E_+ = E_1 + E_2 + \text{carryin!}$

Floating-Point Multiplication

Significand

23 or 52 bit non-negative integer multiplier

carry save adders in a Wallace tree

a shifter to normalize

Floating-Point Division

$E/ = E_1 - E_2 + 1023 = E_1 - (E_2 - 1) = E - (1'sC(E_2))$

For significand, use integer SRT with radix 4 or 16 (ia Pentium)
Rounding

6-9 up
5 to even to make unbiased
1-4 down
0 unchanged
xxxx.1...1.. up
xxxx.10000 to even
xxx.0...1... down
xxx.000000000 unchanged

IEEE FP bounds error to 1/2 “units of the last place” ULP

Keeping error small and unbiased is important
  • can accumulate after billions of operations

other rounding modes
Mixing small and large numbers in FP

(3.1415 ... + 6 *10^{22} ) - 6 *10^{22} != 3.1415 .. + (6 *10^{22} - 6 *10^{22})