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

0000

1000

1001000 base ten

Integer Multiplication

Convert to binary

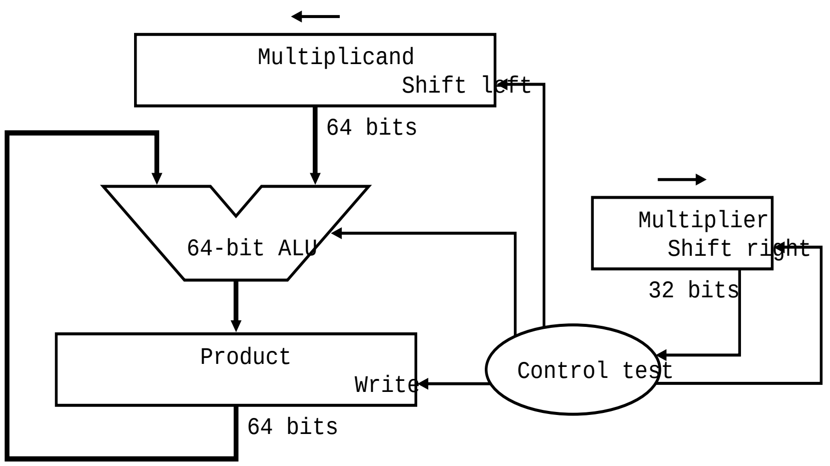
Use carry-save adders in a wallace tree

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

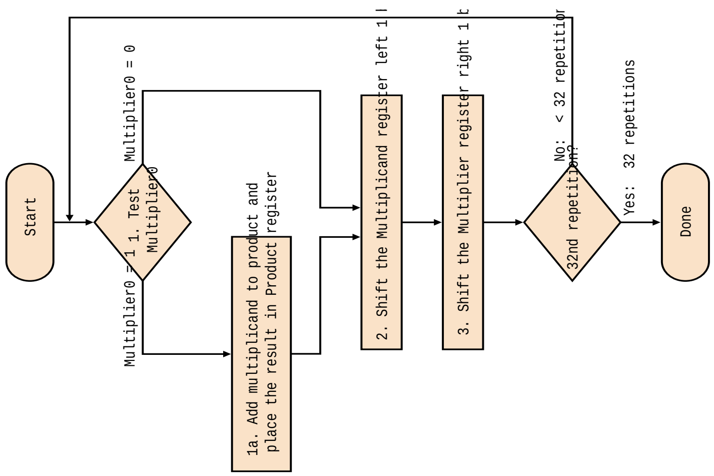
Example next (Figure 4.27)

- Multiplicand = 2 = 0010
- Multiplier = 3 = 0011
- Product = 6 = 0110

Example (Fig 4.25)



Example (Fig. 4.26)



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	

Integer Multiplication

What about negative multiplicand and/or multiplier

- grammar school
- Booth's encoding

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

Integer Multiplication

Booth encoding -- mind bending like carry-lookahead

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

```
    12345
*   09990    = -10 + 10000
      -123450  12345*10
 +123450000  12345*10000
  123326550
```

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

Booth's Encoding

In binary

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

current bit	bit to right	info	
1	0	start 1's	-1
1	1	middle of 1's	0
0	1	end of 1's	+1
0	0	middle of 0's	0

Booth's Encoding

```
-2k 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*512 + 1*128 + -1*16 + 1*8 + -1*4

-2*256 + 2 *64 + -1*16 + 1*4

all equivalent

Booth Encoding

1010 -> -6   8 bits = 11111010 -(-6) = 00000110

0110 -> +6   Boothenc = +1 0 -1 - = +0-0

```
11111010 *0 = 0
1111010_ *-  = 00001100
111010__ *0 = 0
11010___ *+ = 11010000
  11011100 = -36
```

Booth Encoding

negative multiplier

1010 = -6

1110 = -2 booth enc 000-0

0000110_ = 00001100 = +12

$b * a_2\ a_1\ a_0$ =

- $(a_1-a_2)*b*2^2 + (a_0-a_1)*b*2^1 + (0-a_0)*b*2^0$
- $-a_2*b*2^2 + (2*a_1-a_1)*b*2^1 + (2*a_0-a_0)*b*2^0$
- $[a_2*-2^2 + a_1*2^1 + a_0*2^0]$!!

Redundant Representations

Normally

- $d_2*b^2 + d_1*b^1 + d_0*b_0$; b base, d_i usually (0, 1, . . . 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        -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

divisor - 1000 dividend 1001010 - grammar school

1000)1001010(1001 - quotient

1000

10

101

1010

1000

10 - remainder

Integer Division

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

Integer Division

Non-restoring division - a key optimization in division

Recall restoring division:

divisor 1000, 2'sC 1 . . . 11000

Integer Division

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 -divisor*2^2

11101 => < 0

111010 next bit down

+001000 +divisor*2^1

000010 $(-d*2^2+d*2^1) == -d*2^1$

Integer Division

But quotient bits are $\{1, \bar{1}\}$

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

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

convert the weird quotient into 2'sC

for any 2'sC negative numbers:

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

quotient bit = $\bar{1}$ if partial remainder and divisor are opposite sign

SRT Division and Pentium Bug

Normalize so 1 <= 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 {

- determine next quotient digit
- subtract quotient-digit*divisor from partial-remainder (CSA)
- 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

Pentium Bug

Analysis

- There are are actually much worse errors in Pentium
- Errata book (and other microprocessors)
- These can cause completely incorrect results
- People believe hardware is always perfect
- (for software you pay for their bugs!!)
- Pentium bug caught public attention
- and Intel handled poorly

Booth 2-bit Encoding

curr bits	bit to right	info	Op
00	0	mid of 0's	0
00	1	end of 1's	+1
01	0	single 1	+1
01	1	end of 1's	+2
10	0	beg of 1's	-2
10	1	single 0	-1
11	0	beg of 1's	-1
11	1	mid of 1's	0

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 wierd quotient to 2'sC : 1 is 1, $\bar{1}$ 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 $*10^9$ (#seconds in a century!)
- 3,155,760,000
- exponent says where the decimal point "float"

Recall normalization

- use $3.14*10^{10}$ NOT $0.314*10^{11}$ or $31.4*10^9$
- MSD is [1,9] except for 0.0

Floating-Point Numbers

computer floating-point is similar except binary

- number is $-1^S * f * 2^e$ (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

usually

- s = S
- e = E - bias
- $f = 1+ F/2^n$
- e.g., $-1^S * (1.F) * 2^{(E-1023)}$

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

$9.997 * 10^2$

$+ 4.631 * 10^{-1}$

First step: align decimal points, second step: add

$9.997 * 10^2$

$\underline{+ 0.004631 * 10^2}$

$10.001631 * 10^2$

Floating-Point Addition

Third step: normalize the result

- often already normalized
- otherwise move only one digit

$1.0001631 * 10^3$

Example presumes infinite precision; with FP must round

Figure

Floating-Point Subtraction

Subtraction similar

- when adding different signs
- subtracting same signs

Floating-Point Multiplication

Example:

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

Floating-Point Multiplication

Hardware: Figure

Exponent:

e+ = e1 + e2

E+ = e+ + 1023 = E1 - 1023 + E2 -1023 + 1023

E+ = E1 + E2 - 1023

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

With 2'sC E+ = E1 + E2 + 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/ = E1 - E2 + 1023 = E1 - (E2 - 1) = E - (1'sC(E2))

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

Rounding

6-9 up

5 to even to make unbiased

1-4 down

0 unchanged

xxxx.1 . . . 1 .. up

xxxxx.10000 to even

xxx.0 .. . . 1 .. . down

xxx.0000000000 unchanged

Rounding

Need infinite bits? No - hold least significant bits

- guard bits - used for normalization - one bit right of LSB
- round bit - main round bit - one bit right of guard bit
- sticky - logical OR of all less significant bits
- round sticky
- 1 1 round up
- 1 0 round even
- 0 1 round down
- 0 0 no round

Rounding

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

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