Fixed Point Arithmetic with Rounding

A/Prof Lindsay Kleeman

Outline

- Fixed Point Representation
- Fixed Point Multiply and Add
- Rounding in conversion to less bits.
- Fixed Point Division
- Rounding in Division.
  - Integer Division and Biased errors
  - Rounded Integer Division

Fixed Point Representation

- Floating point arithmetic is not supported nor warranted in many applications.
- Fixed point arithmetic uses integer + - * / >> << operations
- The representation *i.f* uses *i+f* bit integers and reserves *i* bits for the integer part and *f* bits for the fraction part.
- The *i.f* format behaves like an integer in units of 2^{-f}
- 2's complement signed or unsigned representations can apply.

Examples

- Q. What is 10.78 in *8.8* fixed point binary format?
- Q. What is 10.78 in *8.8* + 3.75 in *4.4* format

Examples

- Q. What is 10.78 in *8.8* fixed point binary format?

- A. upper_byte.lower_byte

0	0	0	0	1	0	1	0	•	1	1	0	0	0	1	1	1

0.78*2 = 1.56
0.56*2 = 1.12
0.12*2 = 0.24
0.24*2 = 0.48
0.48*2 = 0.96
0.96*2 = 1.82
0.82*2 = 1.64
0.64*2 = 1.28

- Q. What is 10.78 in *8.8* + 3.75 in *4.4* format

-

Examples

- Q. What is 10.78 in *8.8* fixed point binary format?

- A. upper_byte.lower_byte

0 0 0 0 1 0 1 0 • 1 1 0 0 0 1 1 1

0.78*2 = 1.56
0.56*2 = 1.12
0.12*2 = 0.24
0.24*2 = 0.48
0.48*2 = 0.96
0.96*2 = 1.82
0.82*2 = 1.64
0.64*2 = 1.28

- Q. What is 10.78 in *8.8* + 3.75 in *4.4* format

- A. Convert 3.75 in *4.4* format is 0011.1100.
  In *8.8* (pad with 0s):
  00000011.11000000
  00001010.11000111 +

  00001110.10000111

- Check:
  14+1/2+1/64+1/128+1/256
  = 14.52734375
  (correct value is 14.53)

Fixed Point Add Multiply

- Addition/subtraction – each argument must have the same number fraction bits *f*
- Multiplication *i.f * i.f* results in *2i.2f*
  - Result has units of $2^{-f} * 2^{-f} = 2^{-2f}$
  - To convert to *i.f* format without rounding, truncate i bits from left and *f* bits from right
  - To convert to *i.f with rounding*:

    (*i.f* * *i.f* + (1 << (f-1))) >> f results in *i.f*

    Note that ***signed overflow*** occurs if the top *i+1* bits of *2i.2f* are not all 0s or all 1s!

    Any *1* in top *i* bits => ***unsigned overflow***.

Examples

- 1.75 * 1.50 = 2.625
  Calculate using 2.2 operands and 4.4 result.
- Convert result to 2.2 with and without rounding.
- Try 2.75*1.50 using 2.2 inputs and 2.2 result

Examples

- 1.75 * 1.50 = 2.625
  Calculate using 2.2 operands and 4.4 result.
- Convert result to 2.2 with and without rounding.
- In 2.2 inputs and 4.4 result:
  01.11 * 01.10 = 0010.1010
  = 2.625
  in decimal integers:
  7 * 6 = 42

  convert 0010.1010 to 2.2:
  no rounding 10.10 = 2.5
  rounding (00101010 + 10)>>2
  = 00101100 >> 2
  = 1011 (ie 2.75)
- Try 2.75*1.50 using 2.2 inputs and 2.2 result

Examples

- 1.75 * 1.50 = 2.625
  Calculate using 2.2 operands and 4.4 result.
- Convert result to 2.2 with and without rounding.
- In 2.2 inputs and 4.4 result:
  01.11 * 01.10 = 0010.1010
  = 2.625
  in decimal integers:
  7 * 6 = 42

  convert 0010.1010 to 2.2:
  no rounding 10.10 = 2.5
  rounding (00101010 + 10)>>2
  = 00101100 >> 2
  = 1011 (ie 2.75)

- Try 2.75*1.50 using 2.2 inputs and 2.2 result
- 10.11*01.10 = 0100.0010 4.125
  decimal 11*6 = 66

convert to 2.2: 00.00 Unsigned Overflow

Fixed Point Divide

- Division: *i.f1* / *i.f2*
- Quotient result has units of $2^{-f1} / 2^{-f2} = 2^{-(f1-f2)}$
- => quotient *i.{f1-f2}* number of fraction bits is *f1-f2*
- When dividing *i.f* by *i.f*, to achieve quotient of *i.f* add *f* bits (all 0s) to the right of dividend
  (ie shift left by *f* bits, so we do *i.2f / i.f*):

  (i.f << f)/i.f results in truncated quotient *i.f*

Fixed Point Divide Rounded

- *(i.f << f + den>>1)/i.f* has rounded quotient where *den* is the denominator in *i.f* format.

- Alternative form:

  ({i.f << (f+1) } / i.f + 1) >> 1

  - Forms division with extra bit, add 1 and truncate extra bit (shift right). The added 1 before shift right then contributes a half.

Integer Divide x/y Biased Errors

FOR THE MATHEMATICALLY MINDED……..

Define

$q = (x \textbf{ div } y)$ where q, x and y are integers (1)

In C **div** is represented by the integer divide operator /

From (1) there exist integer r, $0 \le r \le y-1$ such that

$x = q*y + r,$

then $x/y = q + r/y$

$= q + error,$ where the real number *error* is $0 \le error \le 1-1/y$ (2)

Approximating the ***real number division*** of x/y with the integer division result q, gives a *biased* error. Assuming uniform distribution of x, then r is uniform and so the mean error from (2) is $½ - 1/2y$

Summary:

Truncated integer division (x ***div*** *y)*

mean error = ½ - 1/2y

Maximum error = 1-1/y

Rounding of x/y

- Motivation: we would like the ***nearest*** integer to the real number x/y where *x* and *y* are integers.

- Rounding of x/y is the same as
  truncate(x/y+0.5)
  ie remove fractional part
  ie find largest integer <= x/y+0.5
  = truncate((x+0.5y)/y)

- In C this can be approximated by:

```
(x+ (y>>1))/y;
```

Rounding (x + (y>>1))/y

Rounding gives about a half maximum error and little or no bias.
Rounding of x/y can be obtained with integer arithmetic by

q_r = (x + y **div** 2) **div** y

Analysis follows:

$$\begin{aligned} q\_r &= (x + y \operatorname{div} 2) \operatorname{div} y \\ &= (q \times y + r + y \operatorname{div} 2) \operatorname{div} y \\ &\quad \textit{where } 0 \le r < y \end{aligned}$$

When $r + y \operatorname{div} 2 < y \quad (1)$

then $q\_r = q$ with an x/y max error of $\frac{r}{y}$

Substituting for r from (1): $\frac{r}{y} < \frac{y - y \operatorname{div} 2}{y} \approx 0.5$

When $r + y \operatorname{div} 2 \ge y$ then $q\_r = q+1$ and

$x/y = q + r/y = q\_r + r/y - 1$ and so the error is $\frac{r}{y} - 1 \ge \frac{y - y \operatorname{div} 2}{y} - 1 \approx -0.5$

Rounded Integer Division

In summary, - (y div 2)/y ≤ error < 1- (y div 2)/y

When y is even:

-0.5 ≤ error < *0.5*
and mean error of *-1/(2y)* *ie small biased error*

When y odd y=2k+1:

-k/(2k+1) ≤ error ≤ *k/(2k+1)*

and mean error=0 *ie zero bias in error*

Rounded integer division: (*x* + (*y* **div** *2*)) **div** *y* has smaller errors and significantly less bias than *x* **div** *y*

Exercise

Improve the accuracy of CPUUsage% by using rounded integer division rather than truncating division.

*Denom = OSIdleCtrMax **div** 100*

```
/* improved:  */          Denom = (OSIdleCtrMax+50)/100;
```

*CPUUsage = 100 – OSIdleCtr **div** Denom;*

```
/* improved: */
    OSCPUUsage = 100 – (OSIdleCtr+(denom>>1))/denom;
```

Exercise

What is the improvement in accuracy?

Existing C implementation:

Max error in denom = OSIdleCtrMax/100 is 0.99, mean error 0.5

Max abs error OSCPUUsage:

Part 1: division by denom can result in error close to 1

Part 2: denom can be in error by a factor 0.99/denom so the
OSIdleCtr/denom can be in error by this factor
but OSIdleCtr/denom = 100-OSCPUUsage

so the worst case error is part1 + part2
= 1 + (0.99/denom)*(100-OSCPUUsage)

Mean error is -0.5 + (0.5/denom)*(100-OSCPUUsage)

Exercise (cont'd)

Improved version:

```
int denom = (OSIdleCtrMax+50)/100;

OSCPUUsage = 100 - OSIdleCtr+(denom>>1))/denom;
```

Max denom error is 0.5, mean denom error < 1/200 = 0.005

Max abs error OSCPUUsage
= 0.5 + (0.5/denom)*(100-OSCPUUsage)

Mean error < (0.005/denom)*(100-OSCPUUsage) ~= 0 (ie ~unbiased)

Summary

- Introduced *i.f* fixed point representation with *i* integer bits and *f* fractional bits.
- Fixed point arithmetic +, -, * and / can be used with rounding.
- Rounding achieved by effectively adding half lowest bit and truncating.
- Conversion between fixed point representations with rounding.
- Rounding and errors have been analysed