UNC CHARLOTTE

Efficient Micro Mathematics

Multiplication and Division Techniques for MCUs

By Kripasagar Venkat

Presented by Adam Wickersham

Introduction

There are 2 types of processors fixed and floating point

- Fixed point processors only support integers and not factions
  - Fixed point suffer from the effects of:
    - Finite word length
    - Round-off
    - Truncation
- Most low cost microcontrollers do not have a multiplier module

Horner's Algorithm

- Several algorithms have been devised for fast multiplication and division using only shifts and adds Horner's is one of them
- Attempts to reduce error and improve accuracy
- Innovative scaling free method to implement integer-real multiplications
- Based on position of the bits with a value of 1 and their distance to neighboring 1s
- Relies on dedicated code

Fractional Multiplier

- The bits of value 1 are identified in the multiplier then shifted and added
- Starting at the rightmost 1 and moving left
- 2^{-1} is a right shift and 2^{1} is a left shift
- Advantages and Disadvantages
  - Much more accurate (shown in example), suffers less from finite word length
  - Need defined code for each multiplier

Fractional Multiplier Example Problem

x*M -> x = 0.2468, M = 0.1357
Conventional Math = 0.0333251953125
Correct Answer using floating point math = 0.03349076
Absolute error = 0.0001655646875 or 5.4 LSB

Horner's for M = 0.1357
Position of 1s in multiplier $\{2^{-14}, 2^{-13}, 2^{-12}, 2^{-11}, 2^{-9}, 2^{-7}, 2^{-3}\}$
Distance to closest binary 1 to the left for each bit (Used for shifting) {1, 1, 1, 1, 2, 2, 2, 4}
$x * 2^{-1} + x = x_1$
$x_1 * 2^{-1} + x = x_2$
$x_2 * 2^{-1} + x = x_3$
$x_3 * 2^{-2} + x = x_4$
$x_4 * 2^{-2} + x = x_5$
$x_5 * 2^{-4} + x = x_6$
Final Product = $x_6 * 2^{-3}$
The absolute error for Horner's = 0.000012976796875 or .42522368 LSB

Integer Multiplier

- Easily extended from factional multiplication
- Instead search from leftmost bit to rightmost
- Must make sure that result does not exceed range

- Example M = 77 = 1001101_b

$x * 2^3 + x = x_1$

$x_1 * 2^1 + x = x_2$

$x_2 * 2^2 + x = x_3$

Final Product = $x_3 * 2^0$

Real Multiplier

- Can use either the fractional or integer multiplication
- Have to scale the real number up or down to either pure fractional or pure integer
- The result then must be scaled again back to the original

Canonical Signed Digit (CSD)

- Based off of Horner's Algorithm
- Uses ternary set {-1, 0, 1} compared to a binary set [0, 1]
- Attempts to reduce the number of 1s present in the multiplier by grouping 1s and replacing them with a combination of the ternary set
- This reduces number of add operations

$M = 0.1357\ 0.0010001010191111 0_b$ Red text is grouped 1s $\bar{1} = -1$

$= 0.0010001011000\bar{1}0_b = 0.0010001101000\bar{1}0_b = 0.0010\bar{1}0\bar{1}0\bar{1}000\bar{1}0_{CSD}$

$= 2^{-3} + 2^{-6} - 2^{-8} - 2^{-10} - 2^{-14}$

Reduced the number of adds by 2

$M = 891 = 1101111011_b = 1101111101_b = 1110000101_b = 1001000010 1_{CSD}$

$= 2^{10} - 2^7 - 2^2 - 2^0 = 1024 - 128 - 4 - 1 = 891$

Reduced the number of adds by 4

Implementation on the MSP430

- Performance increased in code size, CPU cycles, and final result error

The table shows an integer-real multiply of 711 * (14.98789 scaled down by 16 = .936743125), then the result was scaled back up by 16

Methods	CPU Cycles	Code Size	Results	Absolute Error
Horner's Method	33	68 bytes	10656	0.38979
Horner with CSD	27	56 bytes	10656	0.38979
Existing Method (rounded to 14)	107	54 bytes	9954	702.38979
Existing Method (rounded to 15)	107	54 bytes	10665	8.61021
C Floating-point library	427	322 bytes	10656.38979	0

Conclusion

- This method shows superior performance in both code size, speed, and error reduction
- Do not need hardware multiplier to perform multiplication and division
- Can get very good precision with a fixed point processor
- And with memory getting cheaper the code size does not pose limitation as much as speed