COORDINATE ROTATION DIGITAL COMPUTER

(CORDIC)

SENG440: EMBEDDED SYSTEMS

SUBMITTED AUG 5TH 2016

Yuhe Jin V00738443 - yuhejin@uvic.ca

Jia Xu V00810558 - xujia@uvic.ca

TABLE OF CONTENTS

1 INTRODUCTION 2

2 THEORETICAL BACKGROUND 2

2.1 motivation 2

2.2 coordinate rotation digital computer (cordic) 2

2.2.1 cordic rotation mode 3

2.2.2 cordic vectoring mode 3

3 DESIGN PROCESS 3

3.1 test bench c code 4

3.2 functional correct fixed point C code 4

3.3 Optimized C code 4

3.4 Optimized assembly code 4

4 UML DESCRIPTION 4

5 PERFORMANCE EVALUATION 6

6 CONCLUSIONS 7

7 BIBLIOGRAPHY 8

1 INTRODUCTION

The goal of this project is to apply various optimization methods to an ARM based implementation of CORDIC algorithm on both C level and assembly level to improve the execution efficiency of the algorithm. A test bench using math standard C library is built first. Then, the CORDIC algorithm is implemented in a fixed point C code, where only integer operations are used. The fixed point C code is then modified by using various optimization methods to improve the efficiency. After the optimized C code has been validated to ensure its functional correctness, assembly code is generated by using ARM-LINUX-GCC complier with O3 level optimization. The assembly code is also optimized manually to further increase its efficiency. The final complied code after several iteration of optimization has significantly increased execution efficiency comparing to original C code.

2 THEORETICAL BACKGROUND

2.1 MOTIVATION

CORDIC algorithm only utilize two simple operations, add and shift, to implement rotational matrix calculation, which allows fast execution for several complex computational tasks such as calculation of hyperbolic functions, trigonometric functions, logarithms estimation, and Eigen decomposition. This project will only focus on calculation of trigonometric functions.

The fundamental principle of using CORDIC algorithm to calculate trigonometric is all the trigonometric functions can be formalized in terms of vector rotations in a unit circle. Suppose v is vector from origin [0,0]T to a point [x,y]T on a unit circle, the angle θ for v is defined between the positive x axis and vector v . The sinθ and cosθ simply equal to x, and y respectively. CORDIC algorithm rotates vector following a set of rules to calculate trigonometric functions.

2.2 COORDINATE ROTATION DIGITAL COMPUTER (CORDIC)

As mentioned in the previous session, CORDIC uses a series of rotations to calculate trigonometric functions. The formula for rotating a vector is shown below.

$$\begin{cases} x' = x\cos\theta + y\sin\theta \\ y' = y\cos\theta - x\sin\theta \end{cases}$$

The above formula is modified to use for CORDIC algorithm in an iterative manner. The formula below is for (i+1)th iteration.

$$\begin{cases} x[i+1] = \cos\theta[i](x[i] + 2^{-i}y[i]) \\ y[i+1] = \cos\theta[i](y[i] + 2^{-i}x[i]) \end{cases}$$

The rotate angles are predefined by a sequence of elementary rotations. The tangent for the elementary rotations equal to 2^{-i} where i is from 0 to an arbitrary positive number. Increasing the range of i will

improve calculation accuracy. The elementary rotation table is the result from file test bench c code. The rotate angle can be obtained from the iteration of the elementary rotation angles:

$$\theta[i+1] = \theta[i] - \sigma[i]\arctan\left(2^{-i}\right)$$

$$\theta(i) = \arctan(2^{-i})$$

There are two modes for CORDIC algorithm: rotation mode for sine and cosine calculation, vector mode for arctangent calculation.

2.2.1 CORDIC ROTATION MODE

Given a rotation angle and an initial vector, the CORDIC rotation mode will rotate the initial vector by the given rotation angle in counter clock direction. The rotation mode equation shows below:

$$\begin{cases} x[i+1] = x[i] - \sigma[i]2^{-i}y[i] \\ y[i+1] = y[i] + \sigma[i]2^{-i}x[i] \\ z[i+1] = z[i] - \sigma[i]\arctan\left(2^{-i}\right) \end{cases} \quad \text{where} \quad \sigma[i] = \begin{cases} -1 & \text{if } z[i] < 0, \\ +1 & \text{otherwise.} \end{cases}$$

To calculate sinθ and cosθ, the initial vector is set to $[1,0]^T$ After rotating the initial vector by θ, a new vector $[x,y]^T$ is obtained. As noticed in the above equation, the cosθ term is dropped comparing to the original formula. Therefore the $[x,y]^T$ is scaled by dividing $\prod_{i=0}^{n}\sqrt{1+2^{-i}}$ to get $[x',y']^T$, where x' and y' are corresponding cosθ and sinθ respectively.

2.2.2 CORDIC VECTORING MODE

Given an initial vector, CORDIC vectoring mode will rotate the initial vector to align the vector to the positive x axis and record the angle it rotates.

$$\begin{cases} x[i+1] = x[i] - \sigma[i]2^{-i}y[i] \\ y[i+1] = y[i] + \sigma[i]2^{-i}x[i] \\ z[i+1] = z[i] - \sigma[i]\arctan\left(2^{-i}\right) \end{cases} \quad \text{where} \quad \sigma[i] = \begin{cases} -1 & \text{if } y[i] \geq 0, \\ +1 & \text{otherwise.} \end{cases}$$

To calculate arctan$x$, the initial vector is set to [1, arctan$x$]. After go through an sequence of elementary rotations, the vector become $[r,0]^T$ where is r is an arbitrary number. The sum of the elementary rotations equals to arctan$x$.

3 DESIGN PROCESS

The design process is involved developing test bench C code, functional correct fixed point C code, optimized C code and optimized assembly code.

3.1 TEST BENCH C CODE

The test bench code is developed by using standard math library in C to calculate elementary rotation angle table, which will be used in CORDIC algorithm.

3.2 FUNCTIONAL CORRECT FIXED POINT C CODE

Functional correct fixed point C code implement both rotation mode and vector mode in CORDIC to calculate trigonometric functions. The main function contains several test cases for sine, cosine, and arctangent evaluation. Each test case is converted and scaled to integer first by shifting left 14 bits, and then passed into corresponding CORDIC functions. Only integer operation is used in CORDIC functions.

3.3 OPTIMIZED C CODE

To improve the execution efficiency of the C code, varies optimization method has been implemented.

- Loop unrolling: the loop in the CORDIC functions has been unrolled once. Originally, the loop executes 14 times. After unrolling the loop executes 7 times.
- Loop counter initialization: the initialization of loop counter changed to i=i-I instead of i=0 to avoid memory access.
- Use local variable: elementary rotation table has been moved inside the CORDIC functions to become a local variable.
- Use register specifier: the frequently used variable in CORDIC function is defined by using register specifier to make sure it stored in the register.
- Combine two 16 bits angles into a 32 bit word: each elementary angle is stored as a 16 bits integer. By combining two 16 bits angles into a 32 bit word, half of the memory accesses can be eliminated.
- Question mark operator: all the if-then-else statements in the CORDIC functions have been replaced by question mark operator.

3.4 OPTIMIZED ASSEMBLY CODE

Assembly is generated from c code by using GCC complier with O3 level of optimization. It is observed that the loop has been unrolled entirely by complier. There are 19 pairs of mutually exclusive load and move operations. Each pair of operations is merged into a single operation.

4 UML DESCRIPTION

UML sequence diagrams below illustrate how the CORDIC algorithm implemented using integer arithmetic. The four UML sequence diagrams are for sine function evaluation, cosine function evaluation, arctan(x) function evaluation, and arctan(x,y) function evaluation respectively.

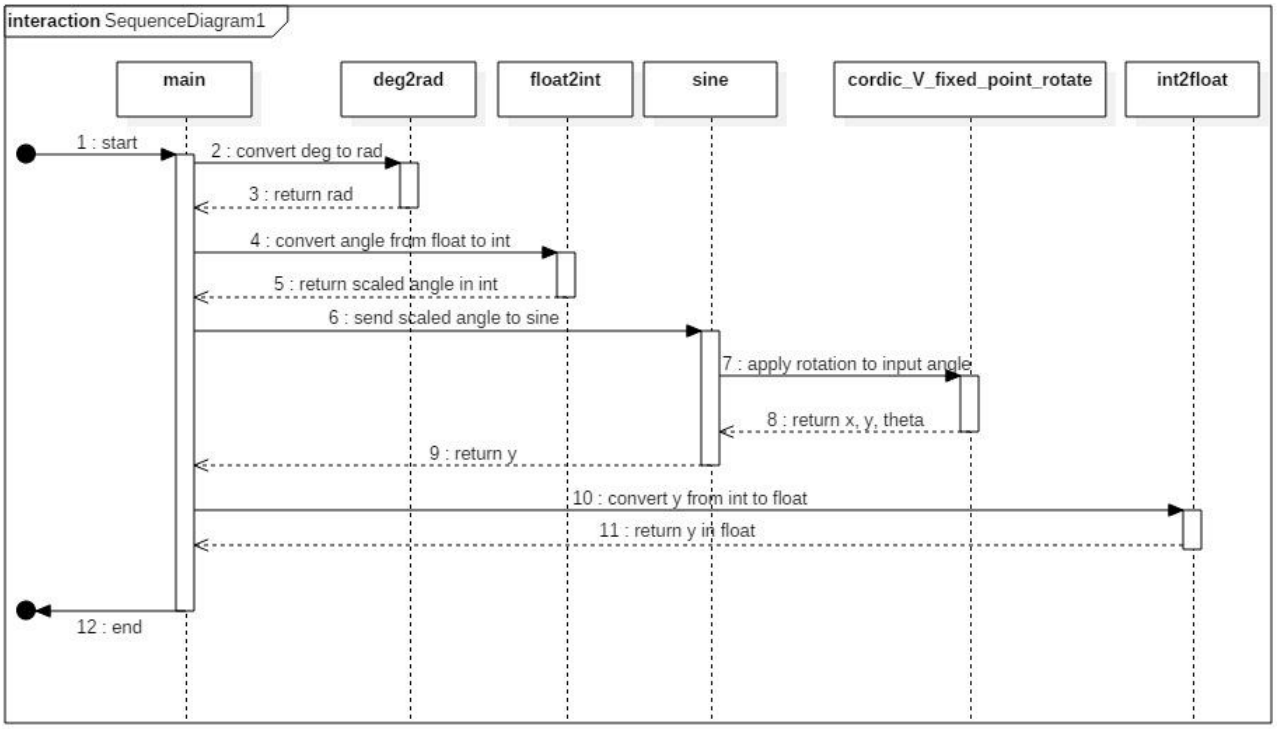

Figure 1: Sequence diagram for sine evaluation

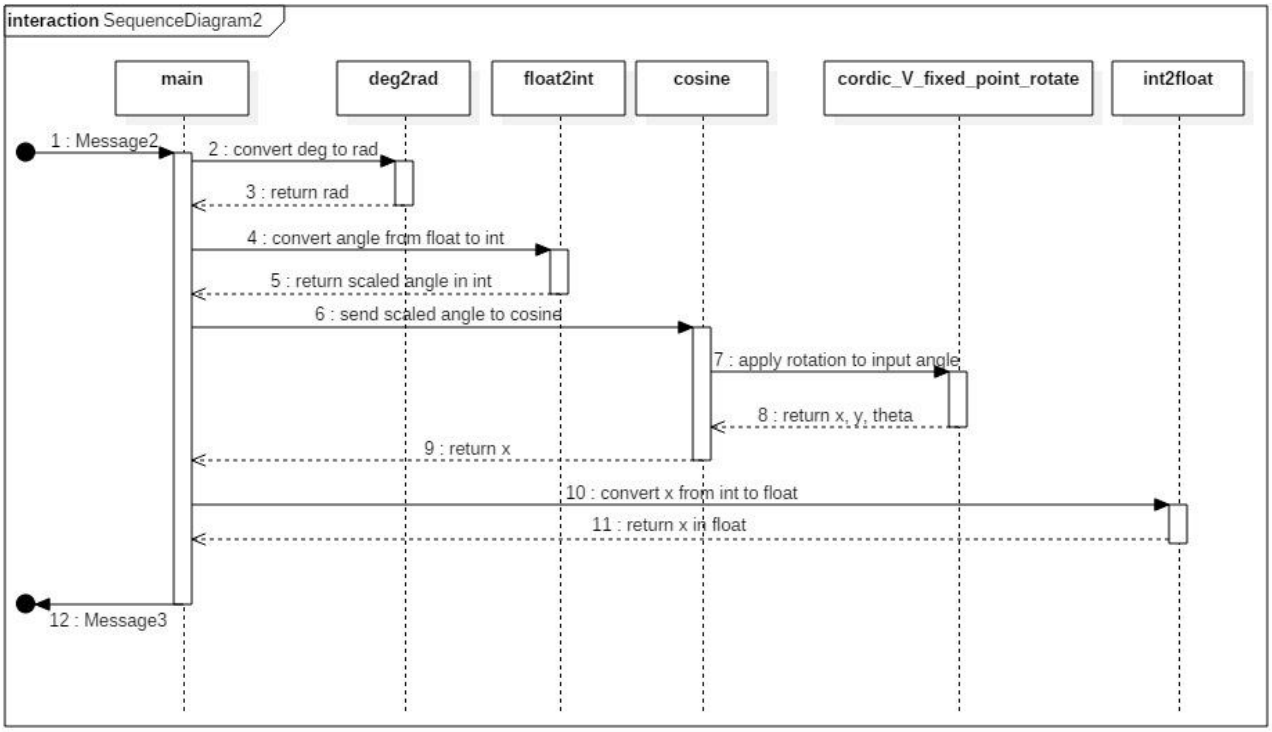

Figure 2: Sequence diagram for cosine evaluation

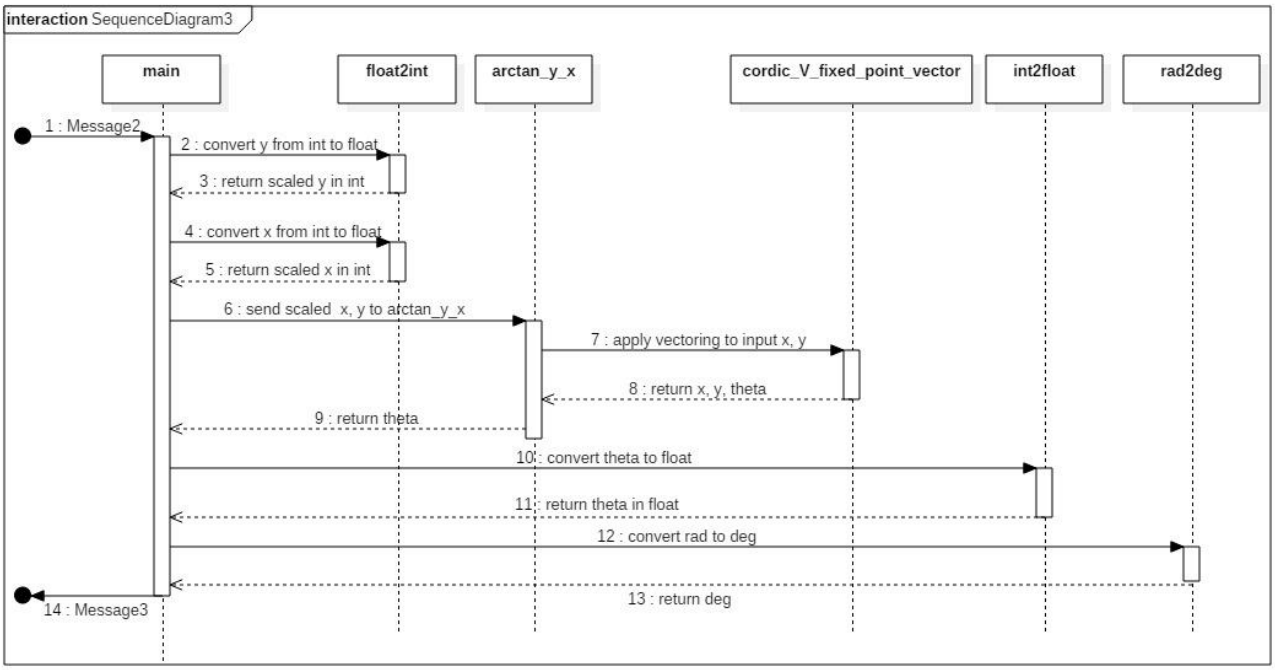

Figure 3: Sequence diagram for arctan(x,y) evaluation

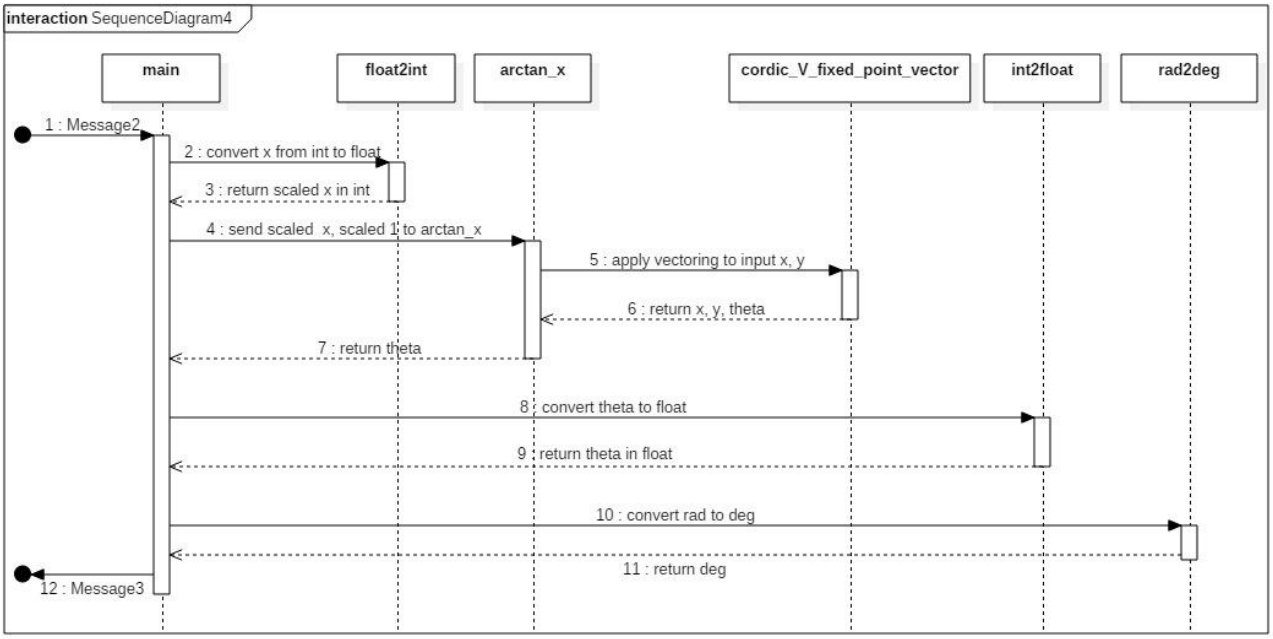

Figure 4: Sequence diagram for arctan(x) evaluation

5 PERFORMANCE EVALUATION

Varies of optimization method have been applied to the CORDIC algorithm. The benefits for these optimization methods are shown in the table below.

Optimization method	Benefit
Loop unrolling	In the final optimized assembly code, the loop has been unrolled entirely, which eliminates the loop overhead such as end of loop test.
Loop counter initialization	The loop is initialized by setting i=i-i instead of i=0. In the case where instruction move immediate is not available, this optimization can save 2-4 cycles. However, the loop has been fully unrolled in the final assembly code which makes loop counter optimization meaningless.
Use local variable instead of global variable	The z-table has been moved into the CORDIC function as a local variable, which reduce the chance of memory access.
Use register specifier	Frequently used variable x, y, z in the CORDIC functions is stored in registers instead of memory. Each memory access takes about 5 cycles, while register access is instant.
Combine two 16bits angles into a 32 bit word	Arm supports load a 32 bit word from memory. Combining two angles into one word will reduce number of memory access for z-table to half. However additional shifting operations are needed to split the word back to two angles which takes 2 cycles. The total cycles of loading the entire z-table without combining tow angles into one word takes 5*14=70, while total cycles reduce to (5+2)*7=49 by combing the angles.
Merge mutually exclusive operations	There are 7 pairs mutually exclusive load operations and 14 pairs mutually exclusive move operations in CORDIC rotate and CORDIC vector functions. By merging these mutually exclusive operations, a total number of (7*5 +14*1) =49 cycles can be reduced.

Table 1: Benefit of various optimization methods

To investigate the performance gain by applying different optimization method, original C code, optimized C code, and optimized assembly code based on optimized C code are tested against same test case. The test case contains 300 times cosine evaluation, 300 times sine evaluation, 300 times arctan(x) evaluation, and 500 times arctan(x,y) evaluation. Each code has been run on an ARM machine 100 times, average running time and standard deviation is shown in the table below.

Code Version	Average Running Time (s)	Standard Deviation (s)	Improvement (%)
Original C code	0.219	0.048	NA
Optimized C code	0.187	0.035	14.6%
Optimized Assembly	0.137	0.062	37.5%

Table 2: Code running time

6 CONCLUSIONS

In this project, fixed point CORDIC algorithm has been implemented on an arm platform. The code has been manually optimized in both C level and assembly level along with GCC O3 level compiler automatic optimization. The performance of final optimized code improved 37.5% comparing with original code. This project shows even after the code has been optimized by compiler, manually applied optimization method can still improve the efficiency of the code. It also illustrate the assembly level optimization is necessary in additional to C level optimization.

7 BIBLIOGRAPHY

Mihai Sima (2016) Lesson 101 (Project) ‘CORDIC’ [Powerpoint slides] Retrieved from http://www.ece.uvic.ca/~msima/TEACHING/COURSES/SENG_440/PROTECTED/SLIDES/SENG_440_slides_Lesson_project_101.pdf