INVERSE MATRIX

2015722068 남윤창

Abstract

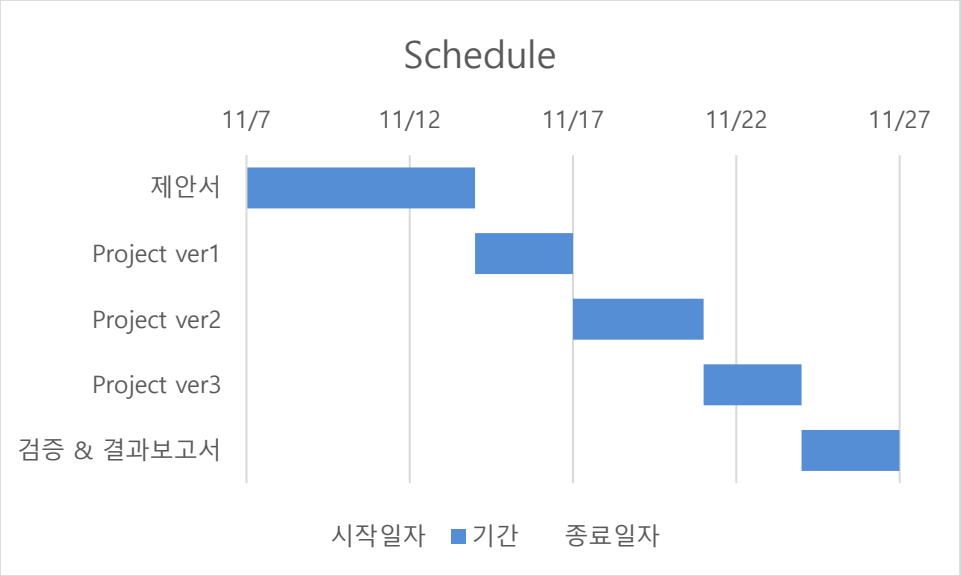
위 보고서는 어셈블리프로그램 설계 및 실습의 term project에 대한 결과보고서이다. Term project에서 부동소수점 값들로 이루어진 정방행렬에 대한 역행렬을 구하는 것에 대한 설계 과정, 알고리즘, 코드 및 결과화면을 나타낸다.

목차

Introduction …… 2
Project Specification …… 2
Algorithms ……. 3
Overview …… 4
Lower Triangle…… 4
Diagonal …….. 5
Upper Triangle…… 5
Row Function #1…….. 6
Row Function #2…….. 6
Row Function #3…….. 7
Adder & Subtractor…….. 7
Multiplication…..…….. 8
Division……..…….. 9
Performance & Result…….. 10
Conclusions…….…… 14
References…… 14

I. Introduction

본 프로젝트는 ARM Assembly 언어를 사용하여 임의의 부동소수점(Floating Point) 데이터로 이루어진 NxN(1<=N<=20) 정방행렬에 대한 Inverse Matrix를 구하는 코드를 작성하는 것이다. 이를 위해 가우스-조던 소거법(Gauss-Jordan Elimination)을 사용하였다. 역행렬을 구함에 있어 성능이 가장 좋은 코드를 구현하는 것이고, 성능의 기준은 Keil을 통한 state를 기준으로 값이 작을수록 성능이 좋다고 정의한다.



본 프로젝트의 일정은 다음과 같다. 제안서를 작성하며 코드작성 진행 순서를 결정하였고 설계도 및 Flowchart를 작성하였다. 위 스케쥴의 Project ver1에는 Inverse Matrix에 사용되는 각 부동소수점 사칙연산 모듈을 구현했다. 사칙연산으로는 Adder, Subtractor, Multiplier, Divider를 직접 구현했다. Project ver2에는 전체적인 Inverse Matrix를 구현했다. Ver1에서 구현한 사칙연산을 Top Module에 합치기 위해 Register Map을 작성하여 Register를 효율적으로 관리하도록 수정하였다. Project ver3에는 각 testset에 대한 내용을 모두 검증하며 오차범위를 구하였고, 불필요한 코드를 삭제하고 Register Map을 최적화하여 30%정도의 성능 개선을 하였다. 마지막 검증 & 결과보고서 기간에는 추가된 testset에 대한 코드 추가 및 검증과 오차범위 개선을 하였다. 기존에 버림으로 작성한 코드를 반올림하는 코드로 수정하였다. State는 조금 늘었지만 오차범위는 더욱 줄어드는 효과를 얻었다.

II. Project Specification

본 프로젝트는 1부터 20 사이의 크기를 갖는 정방행렬에 대한 역행렬을 구하는 코드를 작성하는 것이다. 모든 데이터는 부동소수점으로 이루어져 있다. 이를 연산하기 위한 부동소수점에 대한 사칙연산 코드를 작성하여 오차범위에 알맞은 역행렬 값을 구한다. 역행렬을 구하는 방법은 어느 방법을 사용해도 상관이 없다. 사칙연산은 기존에 제공하는 MUL, DIV연산자를 사용하지 않고 직접 구현해야 한다. 즉, 곱셈 같은 경우 Binary Multiplication 혹은 Booth Multiplication을 사용하여 직접 구현한다. 나눗셈도 직접 Mantissa의 Binary Division을 사용하여 구현한다. 부동소수점은 코드 작성 알고리즘에 따라 오차가 발생할 수밖에 없다. 따라서 버림, 올림, 반올림의 선택은 자유롭다. 하지만 전체적인 결과의 오차 범위가 상대오차 2^-10(~10^3)보다 작아야 한다. 상대오차를 E, Golden Value를 A, 연산을 통한 Result Value를 B라고 할 경우 E = |A-B|/A를 뜻한다.

Matrix_data Label에는 행렬의 가장 왼쪽 위 값부터 오른쪽으로, 행의 오른쪽 끝에 다다르면 그 아래행의 가장 왼쪽 값부터 순서대로 저장되어 있다. 이와 같이 제공되는 5개의 dataset을

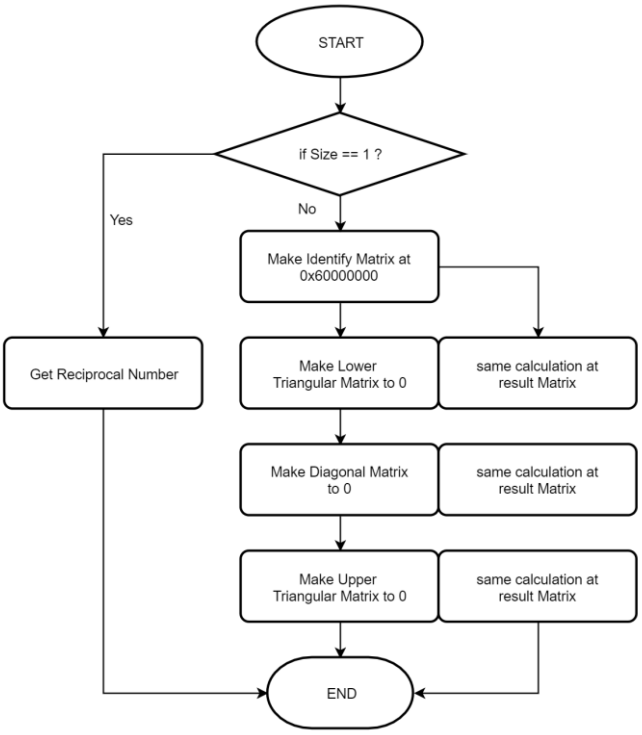
Result_data Label에 복사한 후, 연산을 각각 진행하여 0x60000000 번지에 결과를 1word 단위로 저장하여 Memory Image를 통해 연산이 제대로 진행되었음을 확인한다.

성능에 대한 내용은 Code Size는 배제하고 오로지 Speed, 즉 State수로만 판단한다. 따라서 Code Size에 따른 크기에 대한 Overhead는 고려하지 않는다.

III. Algorithms

1) Inverse Matrix

역행렬을 구하는 방법으로는 가우스-조던 소거법을 사용하였다. 우선 크기가 같은 항등행렬 I를 생성한다. 항등행렬이란 대각선의 값이 모두 1, 나머지 값은 모두 0으로 이루어지는 행렬이다. 이의 특징은 행렬 곱셈연산을 진행할 경우 기존 행렬이 그대로 나온다는 특징을 갖는다. 이는 크기에 따라 Loop를 돌며 직접 0x60000000번지에 생성하도록 하였다. 기존 Input Data Matrix를 A라고 하고 Result Matrix를 B라고 하겠다. A와 B를 동시에 같은 연산을 진행해 주어야 하기 때문에 병렬적으로 처리하였다.



<Flowchart – Program Overview>

[Initialize Result Matrix – Register Map]

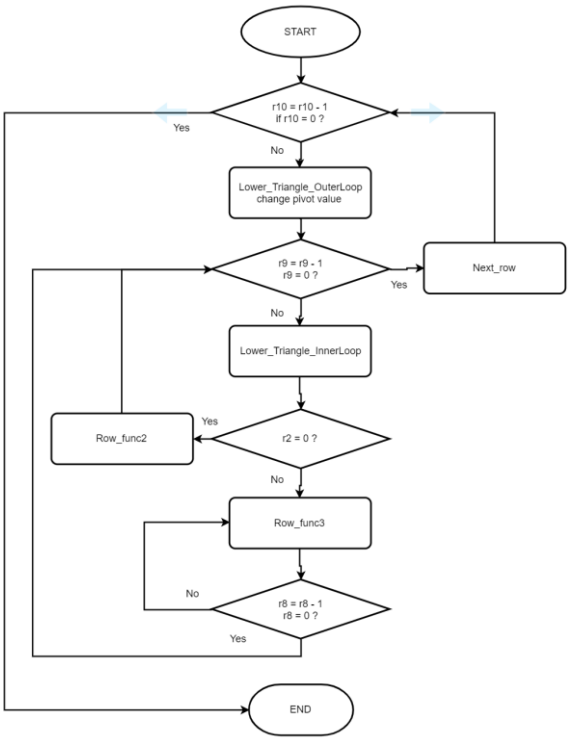
R0	Matrix Size N
R1	Constant 1: 0x3F800000
R2	Constant 0: 0
R9	Outer Loop Iterator
R10	Inner Loop Iterator
R11	Result Data Address
R12	Matrix Data Address

Result Matrix를 항등행렬로 초기화 할 때 사용한 Register들이다. 대각선일때만 constant 1을 대입하고 나머지의 경우 0을 대입하도록 설계하였다.

[Gauss Elimination – Register Map]

<상삼각행렬 Register Map>	
R0	Matrix Size N
R1	Parameter float 1
R2	Parameter float 2
R3	Matrix Data Pivot Below Address (to get const K)
R4	Matrix Data Pivot Address (to get const K)
R5	Return Value of each Calculation result
R6	Result Data Pivot Below
R7	Matirx Data Pivot Below
R8	Row Function Iterator
R9	Inner Loop Iterator
R10	Outer Loop Iterator & temporary space for R5(calculation result)
R11	Result Data Address (Result Data Pivot)
R12	Matrix Data Address (Matrix Data Pivot)
SP(R13)	Stack Pointer(sp)

총 13개의 User-defined Register를 사용하여 구현하였다. 6개의 Label을 사용하였다. 항등행렬을 생성하는 것 이외에 처음으로 시작하는 함수이므로 Main이라는 Label을 사용하였다. 초기 Register들을 지정해주었다. 그리고 하삼각행렬을 모두 0으로 만든다는 뜻을 위해서 'Lower_Triangle_OuterLoop'와 'Lower_Triangle_InnerLoop'라는 Label명을 사용하였다. 그리고 앞서 기술한 행 연산 2번과 3번에 대해 Row_func2, Row_func3이라는 Label을 선언하여 가독성을 높였다. InnerLoop의 실행이 모두 종료될 경우 OuterLoop을 다시 실행하기 위해 새로운 Pivot을 설정해 주는 Label은 'Next_row'로 선언하였다. 이를 사용한 상삼각행렬의 Flowchart는 다음과 같다.

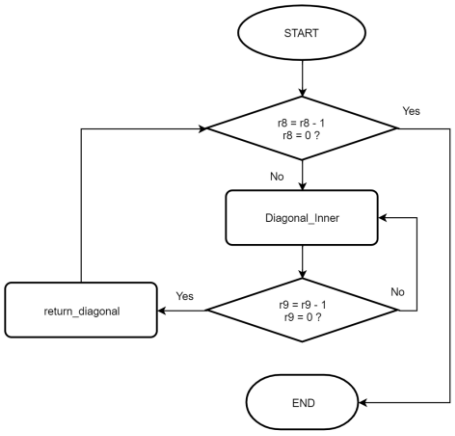


<Flowchart – Lower Triangle to 0>

[Diagonal – Register Map]

< 하삼각행렬 Registers >	R0	Matrix Size N
	R1	Parameter float 1
	R2	Parameter float 2
	R3	constant 1: 0x3F800000
	R4	Matrix Data Pivot Address (to get const K)
	R5	Return Value of each Calculation result
	R6	DIV value (temporary space of r5 calculation result)
	R8	Row Function Iterator
	R9	Inner Loop Iterator
	R11	Result Data Address (Result Data Pivot)
	R12	Matrix Data Address (Matrix Data Pivot)
	SP(R13)	Stack Pointer(sp)

하삼각행렬을 모두 0으로 만든 뒤에는 대각선을 1로 만들어줘야 한다. 행동이 다르므로 Register Map을 새로 재정의 해주었다. 대각선을 1로 만들어주는 Diagonal에서는 총 4개의 Label을 사용하였다. 하삼각행렬을 모두 0으로 만든 후 처음으로 Diagonal의 시작을 알리는 'Diagonal', 행의 개수만큼 반복하는 'Diagonal_Outer', 행 안의 Data수 만큼 반복하는 'Diagonal_Inner', 그리고 Inner Loop이 완료된 경우 Pivot의 재정의를 위해 선언한 'return_diagonal' Label이 있다. Flowchart는 다음과 같다.



<Flowchart - Diagonal>

대각선을 모두 1로 만들기 위해서는 나눗셈을 진행했다. Pivot을 정하고 그에 해당하는 숫자를 나누어 주어 1로 만드는 방식을 사용했다. 미리 구현해둔 Binary Division은 Diagonal_Inner Label에서 호출하여 [1]행 연산 1번의 내용과 동일하게 구현했다. 자세한 나눗셈 코드는 4) Division에서 설명한다.

[Upper Triangle – Register Map]

R0	Matrix Size N
R1	Parameter float 1
R2	Parameter float 2
R3	Matrix Data Pivot Up Address (to get const K)

R4	Matrix Data Pivot Address (to get const K)
R5	Return Value of each Calculation result
R6	Result Data Pivot Up
R7	Matirx Data Pivot Up
R8	Row Function Iterator
R9	Inner Loop Iterator
R10	Outer Loop Iterator & temporary space for R5(calculation result)
R11	Result Data Address (Result Data Pivot)
R12	Matrix Data Address (Matrix Data Pivot)
SP(R13)	Stack Pointer(sp)

상삼각행렬의 값을 모두 0으로 만들기 위해 가우스-조던 소거법을 사용했다. 이는 앞서 설명한 Gauss Elimination의 정 반대 모델이다. 따라서 Address를 관리하는 순서만 다를 뿐 정확히 Gauss Elimination과 연산 순서가 일치한다. 즉 주소를 관리할 때 ADD를 해야하는 순간 SUB를 실행하여 주소를 아래로 혹은 오른쪽으로 움직이던 것을 위로, 왼쪽으로 이동하며 연산하면 된다. 따라서 Flowchart는 생략한다. 다만 이 모든 연산이 끝난 경우 Diagonal이 아닌 Finish Label로 넘어가며 연산이 모두 종료된다.

위 Flowchart에서 사용한 가우스 조던 소거법은 3가지 행 연산을 사용한다. 3가지 행 연산은 다음과 같다. 3가지 행 연산 번호 순서는 임의로 배정하였다.

[1] Row Function 1. Multiply a row by a nonzero number (R1, k)
행 연산 1번은 특정 행에 상수 k배를 곱하는 것이다. 본 프로젝트에서는 특정행을 1로 만들기 위해서 사용한다. 따라서 직접 구현한 Divider를 사용해서 1로 만들어주는 방식을 채택하여 사용하였다. 행 연산 1번에 대한 내용의 자세한 알고리즘은 3) Division에 상세하게 설명한다.

[2] Row Function 2. Swapping two rows (R1, R2)
행 연산 2번은 Pivot이 0일 때 즉, 상수 k를 구할 수 없는 경우 아래 특정 행과 현재 기준행을 바꾸는 경우에 사용한다.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix}$$

다음과 같은 행렬 A, B가 있다고 하자. 이때 a11이 Pivot이라고 할 때, a11값이 0인경우 연산을 const k 값을 구할 수 없다. 따라서 a23과 a33중 0이 아닌 행과 바꾸어 준다. a21이 0이 아니라고 하자. 그렇다면 swap을 진행하면 다음과 같이 변한다.

$$\begin{pmatrix} a_{21} & a_{22} & a_{23} \\ a_{11} & a_{12} & a_{13} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} b_{21} & b_{22} & b_{23} \\ b_{11} & b_{12} & b_{13} \\ b_{31} & b_{32} & b_{33} \end{pmatrix}$$

이후 a21을 Pivot으로 재정의 한 후 a31과의 행연산을 계속 진행하게 된다.

[3] Row Function 3. Adding a Multiple of one row to another row (R1, k, R2)
행 연산 3번은 R1, R2 사이의 상수 k배를 구한 후 그 값을 R2에 더해주는 작업이다. 이는 하삼각행렬, 상삼각행렬의 값을 0으로 만드는 데에 사용되었다. 3번 행 연산에 대한 내용은 다음과 같다.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix}$$

다음과 같은 A, B, pivot이 a11, a21인 경우 a11과 a21의 배수 차이를 const k 라고 한다. 이를 a21이 포함된 행에 연산해주어 a21값을 0으로 만드는 연산이다.
우선 a21/a11을 미리 구현한 Division 연산을 사용하여 구한다. 이후 미리 구현한 Subtractor를 사용하여 a21값을 0으로 다음과 같이 만들어준다.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22} - a_{12} * k & a_{23} - a_{13} * k \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} - b_{12} * k & b_{23} - b_{13} * k \\ b_{31} & b_{32} & b_{33} \end{pmatrix}$$

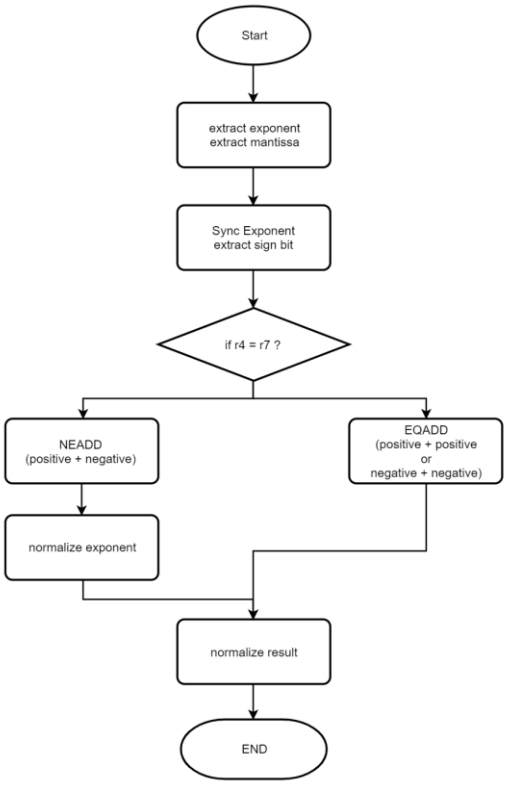
2) Adder & Subtractor

부동소수점의 덧셈과 뺄셈은 하나의 함수로 구현하였다. 부동소수점은 2의 보수가 아니라 Sign Magnitude 방식이라 32bit중 MSB(Most Significant Bit)가 0이면 양수 1이면 음수인 점을 활용하여 뺄셈의 경우 A + B에서 B의 Sign bit를 바꾸어 주어 A + (-B)의 형태로 만들어주었다. 자세한 알고리즘은 다음과 같다.

[Adder & Subtractor – Register Map]

R1	Parameter float 1
R2	Parameter float 2
R3	Exponent 1 & result Exponent
R4	Exponent 2 & Sign bit float 1
R5	Mantissa1 & Result return value
R6	Mantissa2
R7	Shift Amount & Sign bit float 2

가용 Register를 아끼기 위해 사용한 Register를 재사용 하며 구현했다. 또한 성능 개선을 위해 Barrel Shifter를 적극적으로 사용하였다. Mantissa와 Exponent, Sign bit를 분리하기 위해 많이 사용하였다.



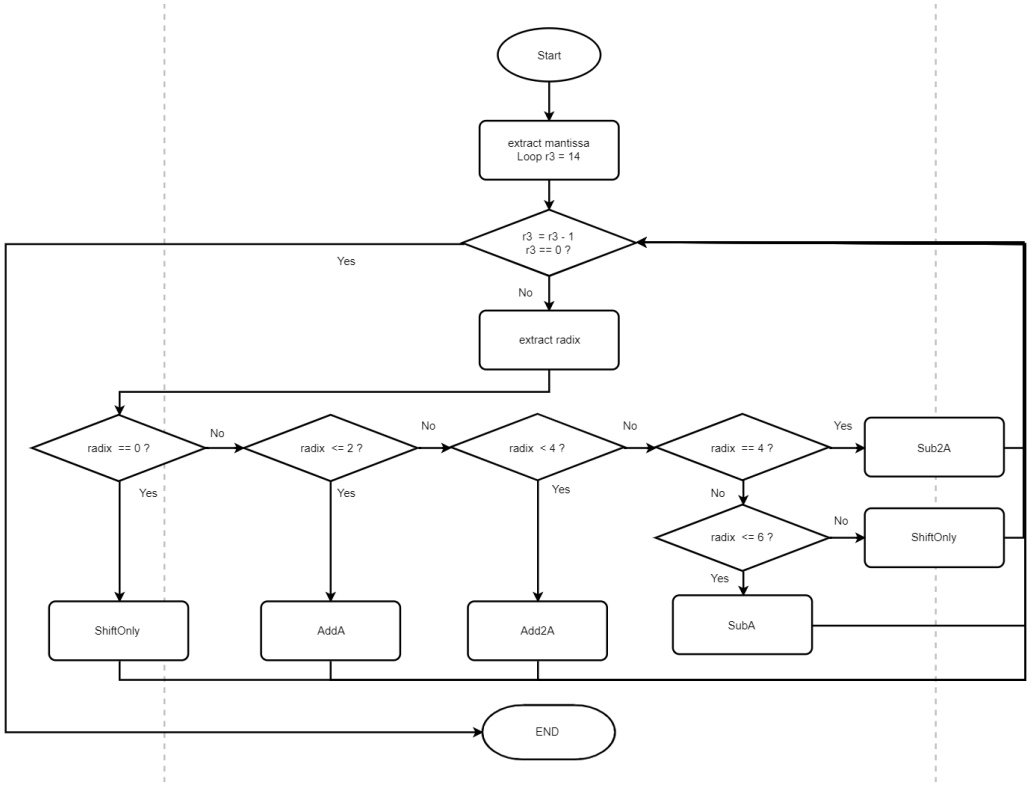
<Flowchart – Adder & Subtractor>

3) Multiplier

Multiplier는 Radix-4 Booth Multiplier를 구현하여 사용하였다. Booth Multiplier를 사용하게 된 이유는 Binary Multiplication보다 빠르다고 판단했기 때문이다. Binary Multiplier는 32bit의 경우 32cycle을 소모하게 된다. Booth Multiplier도 Radix-2라면 똑같이 32cycle을 소모한다. 따라서 cycle을 줄이기 위해 Radix-4로 하여 cycle을 절반으로 줄였다. 원래라면 16cycle을 소모하는 것이 맞다 하지만, Mantissa는 leading one을 포함하여 24bit이기 때문에 12cycle만에 종료되게 하였다. 반면 Radix-8을 사용한다면 8cycle만에 종료되게 할 수도 있지만 계산의 정확성과 많은 범위의 Branch를 줄이기 위해 Tradeoff를 고려하여 Radix-4로 정하게 되었다. 마지막으로 normalize 과정에서 반올림 코드를 추가하여 오차범위를 줄이는 과정을 추가했다.

[Register Map – Radix-4 Booth Multiplier]

R1	Parameter float 1
R2	Parameter float 2
R3	Loop Counter & Sign bit result
R4	Exponent 2 & Exponent result
R5	Mantissa1 (Multiplicand A) & Result return value
R6	Mantissa2 (Multiplier X)
R7	Radix
R8	Mantissa Result



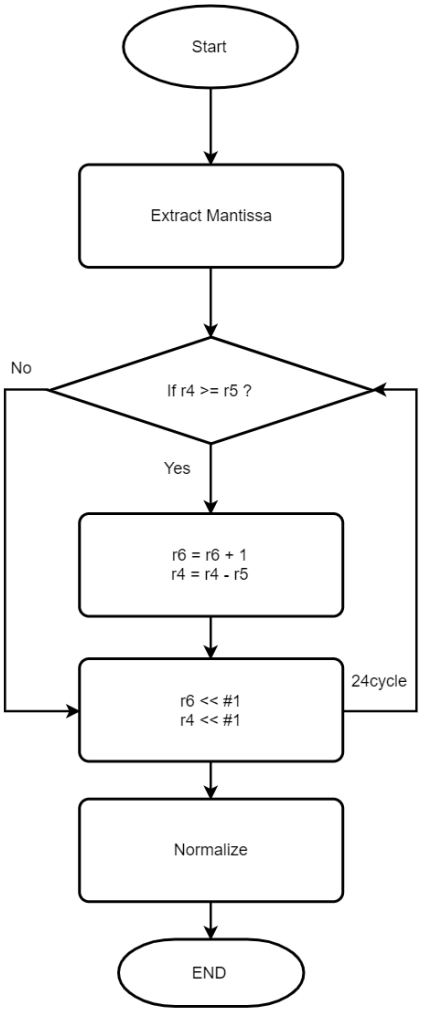
<Flowchart – Radix-4 Booth Multiplier>

4) Division

Division은 Binary Division을 사용하였다. 부동소수점은 Sign Magnitude 방식을 사용함에 따라 Mantissa는 무조건 양수라는 점을 착안했다. 따라서 나눗셈에서 부호를 고려할 필요 없이 양수/양수 라는 결론을 내렸다. 또한, leading one을 추가하여 계산하기 때문에 1.x/1.x라는 결과가 나오고 이는 Fixed Point 나눗셈과 동일하다. 따라서 결과가 0.1x 혹은 1.x, 둘 중 하나로 나오게 되는 점을 이용하여 Normalize하였다. 다만 기존 Binary Division은 몫과 나머지를 구한다. 허나 우리는 소수점 아래를 계산하여야 하므로 몫의 계산이 끝나도 계속 연산을 실행하도록 하였다. 따라서 leading one을 포함한 Mantissa는 24bit이므로 총 24번의 cycle을 통해 Subtraction과 shift를 사용하여 연산한다. State를 줄이기 위해 24번의 연산을 Branch를 사용하지 않고 Loop Unrolling 방식을 사용하여 Code Size는 늘었지만 성능을 개선하였다. 연산방법은 다음의 Register Map과 Flowchart를 통해 확인할 수 있다.

[Register Map – Binary Division]

R1	Parameter float 1
R2	Parameter float 2
R3	Loop Counter & Sign bit result
R4	Mantissa 1 & Result Exponent
R5	Mantissa 2 & calculation result
R6	Mantissa division result

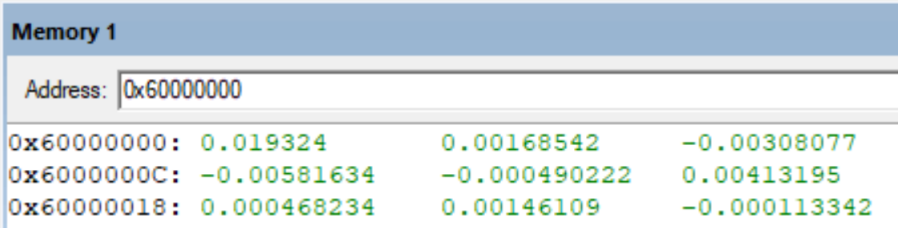


<Flowchart – Binary Division>

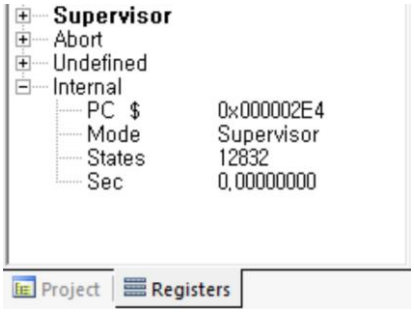
IV. Performance and Results

1) Test Dataset 1 (3x3 Matrix)

3x3 Matrix의 결과값을 도출하였고 a22의 값 -0.000490222의 값만 10^-5정도의 오차를 갖는 것을 확인하였다. Size가 작은 Matrix라 오차가 거의 나지 않는 것을 확인할 수 있다. State는 12832이다.



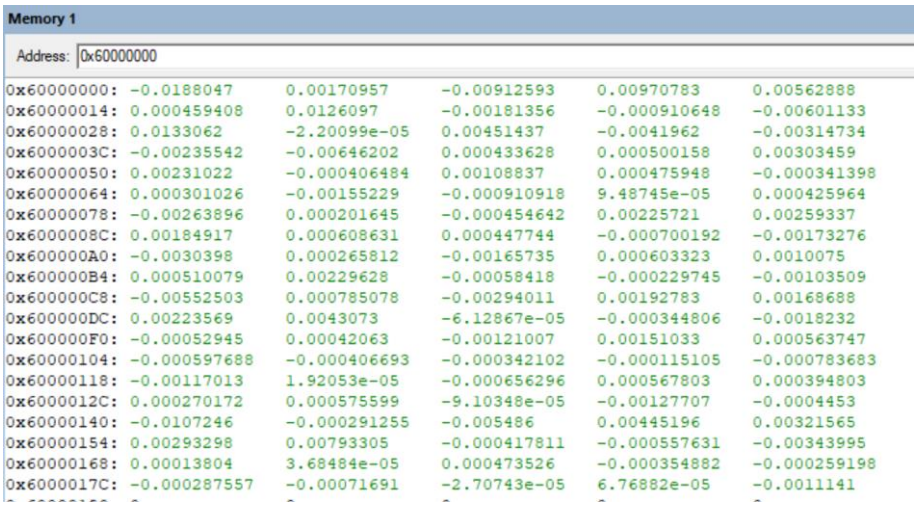
<Memory Image 0x60000000>



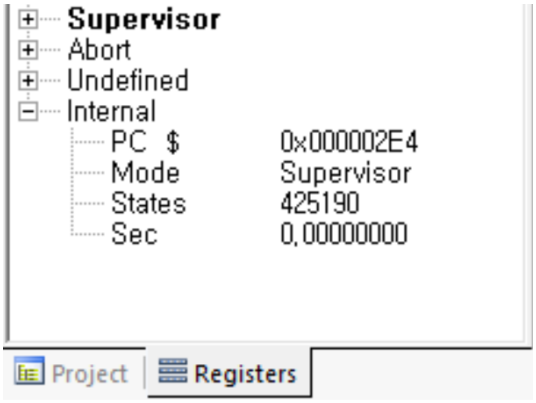
<State – 3x3: 12832>

2) Test Dataset 2 (10x10 Matrix)

10x10 Matrix의 결과값을 도출하였고 Worst case의 경우 대략 2x10^-5정도를 갖는 것을 확인하였다. 주어진 오차범위 2^-10보다 훨씬 작은 값을 갖는 것을 확인했다. State는 425190이다



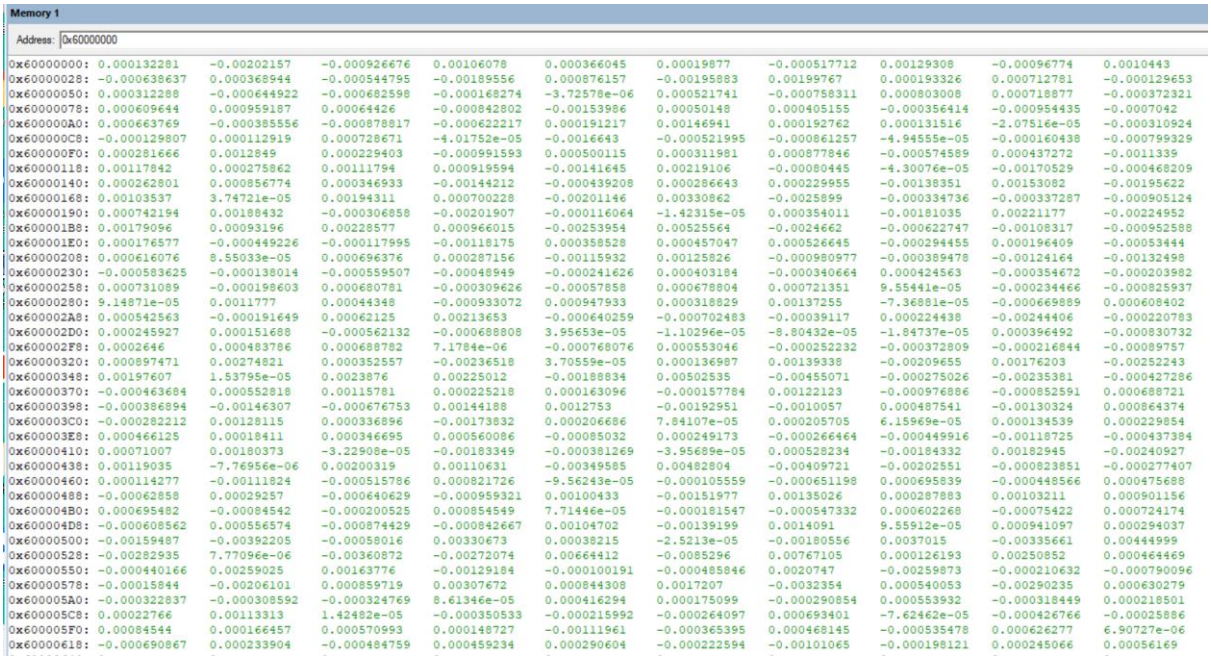
<Memory Image 0x60000000>



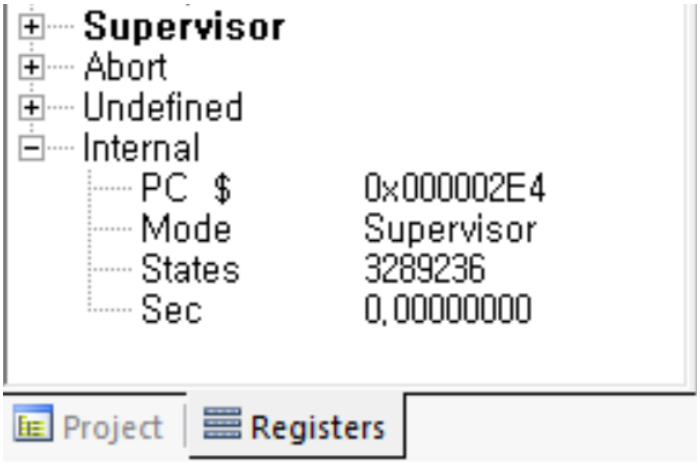
<state – 10x10: 425190>

3) Test Dataset 3 (20x20 Matrix)

20x20 Matrix Data를 돌려본 결과 오차 범위를 벗어나는 것을 확인하였고 반올림 코드를 추가하였다. 따라서 Worst Case의 오차범위가 4x10^-4정도인 것을 확인하였고 주어진 오차범위 2^-10보다 적게 나오는 것을 확인하였다. State는 3289236이다.



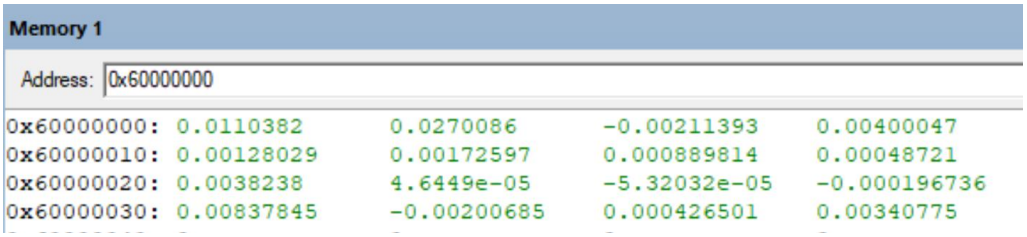
<Memory Image 0x60000000>



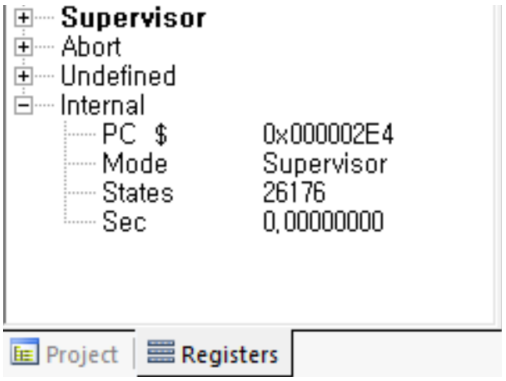
<state – 20x20: 3289236>

4) Test Dataset 4 (4x4 Matrix: Diagonal with 0)

4x4 Matrix의 경우 대각선에 0이 포함되어 행 연산 2번을 수행해야 한다. 결과는 0.00128029는 7x10^-6결과의 오차가 발생하였고, 나머지의 경우는 오차가 발생하지 않았다. State는 26176이다.



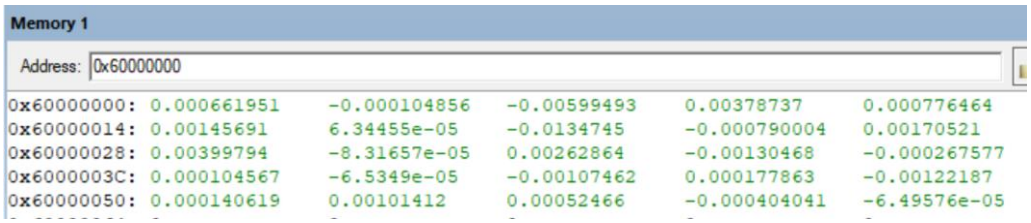
<Memory Image 0x60000000>



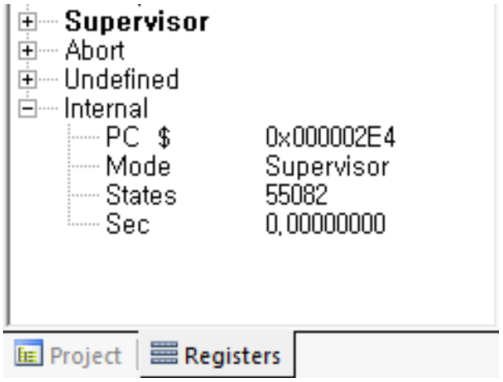
<state – 4x4: 26176>

5) Test Dataset 5 (5x5 Matrix: Diagonal with 0)

5x5 Matrix는 중간에 값이 0이 포함 되어있다. 따라서 0을 처리해주는 코드가 필요하다. 오차범위 1.2x10^-5정도를 갖는 올바른 결과값들을 얻은 것을 확인했다. State는 55082이다.



<Memory Image 0x60000000>



<state – 5x5: 55082>

V. Conclusion

본 프로젝트 코드를 작성하며 부동소수점의 오차에 대한 깊은 이해를 할 수 있다. 32bit Register를 사용하여 Single Precision, 23bit Mantissa를 이용하여 연산을 할 경우 코드 구현방법에 따라 오차가 크게 발생한다는 점을 직관적으로 이해할 수 있다.

작성 코드의 결과를 확인해본 결과 부동소수점의 연산 횟수가 늘어날수록 즉, Matrix Size가 클수록 오차의 범위가 1bit씩 누적되어 매우 큰 오차가 생기는 것을 확인했다. 돈에 관련된 업무를 할 경우 사용할 수 없다는 결론을 얻을 수 있다.

성능 개선을 위한 테크닉은 다음과 같다. 우선 성능 개선을 위해 20x20을 기준으로 비교하였다. 초안은 4370000가량 나왔다. 알고리즘을 크게 분석한다면 3중 Loop를 통한 행렬을 순회하며 1:1로 값을 연산하여 바꾸고 비교하는 방식을 사용하였다. 그 3중 Loop중 가장 안쪽 Loop에 사칙연산들이 모두 있기 때문에 이를 줄이는 것을 제 1 목표로 하였다. 따라서 Multiplier의 Code를 Barrel Shifter를 통해 정리하고, Booth Algorithm을 개선하는 방법을 사용하였다. Division은 Mantissa가 무조건 24bit이므로 24cycle을 branch가 아닌 Loop Unrolling을 사용하여 줄였다. 전체적으로 많이 사용한 기법은 MOV, ADD instruction을 ADD with Barrel Shifter를 사용하여 줄였다. CMP instruction을 사용하지 않기 위해 S를 통해 conditional flag를 바꾸는 기법을 사용하였다. Thumb of rule을 이해하고 활용하여 Conditional Execution과 Branch의 활용을 적절하게 사용하였다.

본 프로젝트 코드를 작성함에 따라 RISC Processor의 구동방식과 장, 단점에 대해 깊은 이해를 할 수 있다. Load/Store Architecture에 대한 이해와 Memory Access에 대한 Overhead를 확인할 수 있었다. 소프트웨어 코딩을 할 경우에도 Memory 동작을 이해하며 성능의 개선에 대해, 최적화에 대해 고민을 할 수 있는 능력을 기르는 좋은 기회가 되었다.

VI. Reference

[1] Booth Multiplication / http://people.ee.duke.edu/~jmorizio/ece261/F08/projects/MULT.pdf
[2] Binary Division / https://www.electrical4u.com/binary-division/
[3] ARM SWP Instruction /
http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dht0008a/CJHBGBBJ.html
[4] 광운대학교 어셈블리프로그래밍 설계 및 실습 강의자료/2019/이준환