공학석사 학위논문

곱셈기를 사용한 정확한 제곱근 알고리즘

2008년 2월

부경대학교 산업대학원

컴퓨터공학과

최 승 희

공학석사 학위논문

곱셈기를 사용한 정확한 제곱근 알고리즘

지도교수 조 경 연

이 논문을 공학석사 학위논문으로 제출함

2008년 2월

부경대학교 산업대학원

컴퓨터공학과

최 승 희

최승희의 공학석사 학위논문을 인준함

2008년 2월 10일

주 심 공학박사 서 경 룡 (인)

위 원 공학박사 권 오 흠 (인)

위 원 공학박사 조 경 연 (인)

목 차

표 차 례

그 림 차 례

Exact Square root Algorithm Using Multiplier

Seung-Hee Choi

Department of Computer Engineering
Graduate School of Industry
Pukyong National University

Abstract

Floating point division and square root operation have two kinds of method, one is a repetition of multiplication, Newton-Raphson algorithm and Goldschmidt's algorithm, the other is a repetition of subtraction, SRT algorithm. On this paper, square root algorithm is suggested, which use double format digit multiplier for satisfying the accuracy required on IEEE-754.

Using Newton-Raphson algorithm, approximate reversed square root is calculated, and one algorism for re-calculating square root reducing the error from above operation and the other algorithm for revising the calculated value are suggested.

The correctness of the suggested algorithm is verified by investigating the total number on floating point and by investigating the randomly subtracted billion number on double.

The suggested algorithm on this paper only use a double format digit multiplier, so an additional hardware is not required for floating point operation. This algorithm could be used requiring an accurate square root operation.

Ⅰ. 서 론

부동소수점에서 나눗셈 및 제곱근 연산은 덧셈(뺄셈) 및 곱셈보다 연산 시간이 오래 걸린다. 덧셈은 2~4 사이클, 곱셈은 2~5 사이클 만에 배정도실수 연산을 할 수 있지만, 나눗셈은 9사이클에서 60사이클 이상, 제곱근은 15사이클에서 70사이클 이상 소요되고 있다[1]. Oberman과 Flynn의 연구[2]는 나눗셈과 제곱근은 덧셈(뺄셈) 및 곱셈보다 출현 빈도는 낮지만, 시스템에서 나눗셈과 제곱근의 수행시간이 덧셈이나 곱셈과 비슷하게 소요됨을 보이고 있다. 또한, 제곱근 계산은 음성이나 3차원 그래픽 같은 멀티미디어 분야에서 출현 빈도가 높다. 따라서 제곱근 계산 속도를 높이는 연구가 요구되고 있다.

부동소수점 제곱근 계산은 뺄셈을 반복하는 SRT[3-4] 알고리즘과 곱셈을 반복하는 뉴턴-랍손(Newton-Raphson) 알고리즘 및 골드스미트(Goldschmidt) 알고리즘이 있다[5-9].

SRT 알고리즘은 덧셈(뺄셈), 부분곱셈, 몫 결정의 3단계 과정을 반복하는 알고리즘으로 매 반복마다 일정한 비트의 몫을 얻을 수 있다. 이러한 SRT 알고리즘은 IEEE-754에서 요구하는 정밀계산이 가능하지만, 연산속도가 느린 단점이 있다.

뉴턴-랍손 및 골드스미트 알고리즘은 반복식을 정의하고 근사값을 초기값으로 하여 반복 연산을 수행하여 오차를 줄여나간다. 반복할 때마다 오차가 자승에 비례해서 줄어들므로 연산 속도가 빠른 장점이 있으나 근사적인 값만을 구할 수 있다.

본 논문에서는 뉴턴-랍손 역제곱근 알고리즘을 이용하여 근사적인 역제곱근을 구하고, 이때의 오차의 자승에 비례하는 오차를 가지는 제곱근 알고리즘을

구하고, 계산된 제곱근을 보정하면서 스티키 비트(sticky bit)를 구하는 알고리즘을 제안한다.

본 논문에서 제안하는 알고리즘을 C 언어를 사용하여 구현하고, 동작을 검증했다. 단정도 실수에서는 전수 조사를 통해서, 배정도 실수에서는 10억개의 무작위 수에서의 계산을 통하여 모두 정확한 값을 얻어서 정확도를 검증했다.

본 논문의 구성은 다음과 같다. 2장에서는 부동소수점 형식과 반복 연산 알고리즘에 대한 관련 연구들을 소개하고, 3장에서 제곱근 알고리즘을 제안한다. 그리고, 4장에서 알고리즘을 구현하여 실험하고, 5장에서 SRT 제곱근 알고리즘과 비교한다. 마지막으로 6장에서 결론을 맺는다.

Ⅱ. 관련 연구

컴퓨터에서 쓰는 수 표현은 부호가 있는 정수와 부호가 없는 정수가 있다. 이와 함께 소수점을 포함한 실수도 필요하다. 실수는 부동소수점으로 표현한다.

부동소수점 표현법으로 대표적인 것이 IEEE-754 표준이다. IEEE-754 표준은 2진 부동소수점 연산을 나타내는데 쓰는 표현법으로 1985년에 제정되었으며, 현재 모든 부동소수점 연산은 이 표준에 따라 구현되고 있다. 이 표준은 데이터 형식, 라운딩, 산술, 형식변환, 비교 등을 정의하고 있으며, 무한대, NaN, '0' 같은 특수 값과 예외 발생 그리고 이에 따른 처리 및 트랩을 정의하고 있다.

한편, 반복 연산을 하는 제곱근 알고리즘은 이 형식을 바탕으로 유효범위의 값을 연산한다. 반복연산 제곱근 알고리즘은 뺄셈을 반복하는 SRT 알고리즘과 곱셈을 반복하는 뉴턴-랍손 알고리즘 및 골드스미트 알고리즘이 있다. SRT 알고리즘은 덧셈, 부분곱셈, 몫 결정의 3단계 과정을 반복하여, 매 반복마다 일정한 비트의 몫을 얻어나가는 것이고, 뉴턴-랍손 및 골드스미트 알고리즘은 반복식을 정의하고 근사값을 초기값으로 하여 반복연산을 수행하여 오차를 줄여나간다.

2.1 IEEE-754 부동소수점 표준

IEEE-754[10] 부동소수점 표준에서는 부동소수점 형식을 네 가지로 정의하고 있다. 즉, 단정도 및 배정도 형식의 기본 형식과 확장 형식으로 정의한다.

그림 1은 IEEE-754 표준 부동소수점 단정도 형식의 기본형식을 나타낸다. 이 형식은 부호를 나타내는 1비트 S, 2의 승수 즉 지수를 나타내는 8비트 E, 소수점 이하를 나타내는 가수 23비트 f로, 모두 32비트로 되어 있다.

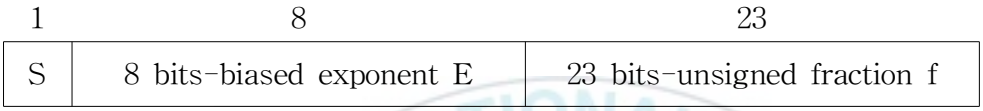

그림 1. 단정도 형식

그림 2는 IEEE-754 표준 부동소수점 배정도 형식의 기본형식을 나타낸다. 이 형식은 부호를 나타내는 1비트 S, 2의 승수 즉 지수를 나타내는 11비트 E, 소수점 이하를 나타내는 가수 52비트 f로, 모두 64비트로 되어 있다.

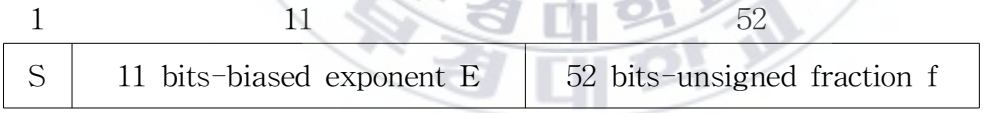

그림 2. 배정도 형식

확장형식의 단정도 형식은 지수비트를 11비트로 늘리고, 가수비트를 24비트에서 32비트 또는 그 이상으로 확장한 형식이다. 그래서 모두 44비트 이상이 된다. 또한, 확장형식의 배정도 형식은 지수비트를 11비트에서 15비트로 늘리

고, 가수비트를 53비트에서 64비트 또는 그 이상으로 확장한 형식이다. 모두 80비트 이상이 된다.

흔히 지수에 바이어스를 가해서 쓰는데 이것은 연산을 편리하게 하고, 부동소수점 형식의 인코딩 수의 절대 크기에 따라 차례대로 나타낼 수 있도록 하려는 것이다. 단정도 형식은 +127, 배정도 형식은 +1023의 바이어스를 쓴다. 이에 따라 표현할 수 있는 수를 나타내면 다음과 같다.

- 32 bit 단정도 형식
  - $E = 255$ 이고 $f \neq 0$ 이면 $F = \mathrm{NaN}$
  - $E = 255$ 이고 $f = 0$ 이면 $F = (-1)^S\infty$
  - $0 < E < 255$이면 $F = (-1)^S 2^{E-127}(1.f)$
  - $E = 0$ 이고 $f \neq 0$ 이면 $F = (-1)^S 2^{-126}(0.f)$
  - $E = 0$ 이고 $f = 0$ 이면 $F = (-1)^S 0$

- 64 bit 배정도 형식
  - $E = 2047$ 이고 $f \neq 0$ 이면 $F = \mathrm{NaN}$
  - $E = 2047$ 이고 $f = 0$ 이면 $F = (-1)^S\infty$
  - $0 < E < 2047$ 이면 $F = (-1)^S 2^{E-1023}(1.f)$
  - $E = 0$ 이고 $f \neq 0$ 이면 $F = (-1)^S 2^{-1022}(0.f)$
  - $E = 0$ 이고 $f = 0$ 이면 $F = (-1)^S 0$

지수가 0이고 가수가 0이 아닐 때에는 선두 비트가 0인 비정규화 수를 나타내는 것으로 정규화 수 형식으로 나타낼 수 없는 작은 값을 나타내는 데에 사용한다. 이러한 비정규화 수를 비정규화수(de-normalized number)라 한다.

그리고 부동소수점 수를 연산하면 결과값이 표현범위를 넘어서게 되는데 표

현범위 밖의 값은 잘라내야 한다. 이때 적절하게 잘라내는 것을 라운딩이라 한다. 라운딩은 무한한 정확도를 가지고 연산한 결과가 정해진 결과 형식에 맞추어 저장이 될 때 하게 되는데, 다음과 같이 모두 네 가지가 있다.

첫 번째, Round to Nearest Even (RNE)
두 번째, Round toward Zero (RZ)
세 번째, Round toward Minus Infinity (RNI)
네 번째, Round toward Plus Infinity (RPI)

RNE는 기본 라운딩 모드로 유효범위의 수 즉, 가수의 LSB에 따라서 한 비트 더해 주는 것으로 LSB가 1인 경우에는 '+1', LSB가 0인 경우에는 '+0', 정확하게 두 표현 가능한 수에서 같은 거리에 있으면 0이 되도록 결정하는 것이다. 이것은 라운딩을 해서 생기는 오차의 평균이 0이 되도록 하려는 것이다. 따라서, 위의 네 라운딩 모드에서 실제 값과 가장 가까운 표현 가능한 수로 결과가 결정이 된다. RPI는 표현값의 양의 무한대 쪽에 가깝게 라운딩하는 것이고, RNI는 반대로 음의 무한대 쪽에 가깝게 라운딩하는 것이다. 그리고, RZ는 표현값의 0에 가깝게 라운딩하는 것이다.

부동소수점 연산에는 '덧셈', '뺄셈', '곱셈', '나눗셈', '제곱근', '나머지' 같은 산술연산과 '부동소수점끼리 형식 변환', '부동소수점과 정수의 형식 변환', '비교' 등이 있다. '비교'를 할 때에는 'unordered'라 하여 NaN처럼 비교할 수 없는 수를 처리해야 하는 경우를 대비하고 있다.

특수값에는 무한대(Infinity), NaN(Not a Number), 부호화한 영(Signed Zero) 등이 있다.

또한, 예외처리를 규정하고 있는데 예외에는 크게 다섯가지가 있다. 첫 번째 무효연산, 두 번째 0으로 나누기, 세 번째 오버플로우, 네 번째 언더플로우 그

리고 다섯 번째로 부정확이 있다. 무효연산은 수행할 연산의 피연산자가 무효일 때 생기며, 0으로 나누기는 0 또는 무한대가 아닌 피제수를 0으로 나눌 때에 생기고, 오버플로우는 연산 결과가 정규화 수로 나타낼 수 있는 최대 절대값을 넘어설 때 생긴다. 또한 언더플로우는 연산 결과가 정규화 수로 나타낼 수 있는 최소 절대값 미만일 때에 생기고, 부정확은 라운딩으로 0이 아닌 가수 부분이 잘려서 없어졌을 때에 생긴다. 이때 각 예외 상황의 트랩 플래그가 활성화해 있으면 결과는 무시하고 트랩 루틴에 따라 처리하며, 활성화해 있지 않으면 기본 결과 값을 출력해야 한다.

예외상황이 생겼을 때에는 각 예외상황을 트랩처리기(trap handler)를 활성화(enable), 비활성화(disable), 저장(save), 복원(restore)할 수 있어야 한다. 더구나 오버플로우(overflow)와 언더플로우(underflow) 경우의 트랩 처리가 활성화해 있을 때에는 지수 값에 다시 바이어스를 가해 정규화 한 수처럼 처리할 수 있도록 하고 있다.

2.2 SRT 제곱근 알고리즘

SRT[11] 알고리즘은 기존의 비복원 알고리즘의 수렴조건의 범위를 축소한 수렴조건을 채택하여 몫 결정에 redundancy를 도입한 알고리즘이다.

IEEE-754 표준안에 의한 부동소수점 데이터의 가수 범위는 (1, 2)이므로, 제곱근 연산에서의 입력 오퍼랜드인 radicand의 범위는 지수가 홀수인 경우를 고려하여, (1,4)의 범위를 갖게 된다. 이 경우에는 radix-4 SRT 나눗셈 연산 알고리즘에서의 수렴범위와 일치하지 않는다. 수렴범위의 일치를 위해 루트(root) 비트를 {-1, -1/2, 0, 1/2, 1}로 축소하여 수렴범위를 일치하게 하는 방

식이 있는데, 여기서는 루트비트를 수정하지 않고, 그 대신 radicand의 범위를 (1/4, 1)로 축소하여 나눗셈 연산과의 수렴범위를 같게 함으로써 하드웨어 공유에 있어 가장 중요한 부분을 차지하는 몫/루트 선택의 비교 동작을 하나의 PLA (Programmable Logic Array)로 설계 가능하게 하였다. 이를 위해 우선 입력오퍼랜드를 2비트 오른쪽으로 쉬프트 시킨 후 제곱근 연산을 수행하고, 그 결과를 1비트 왼쪽으로 쉬프트 시켜 최종 제곱근 연산결과를 얻는다.

radicand가 식 (1)과 같이 주어졌을 때, 기본적인 radix-4 제곱근 연산 알고리즘의 순환방정식은 식 (2)와 같다.

$$\frac{1}{4} \leq radicand(A) < 1 \tag{1}$$

$$P_{j+1} = 4 \cdot P_j - q_{j+1} \cdot (2\,Q_j + q_{j+1} \cdot 4^{-(j+1)}) \tag{2}$$

여기서

$$Q_j = \sum_{i=0}^{j} q_i \cdot 4^{-j} \quad , \quad q_i \in \{-2, -1, 0, 1, 2\}$$

단, P_j는 j번째 연산에서의 partial remainder, Q_j는 j번째 연산에서의 몫, q_i는 i번째 연산에서의 몫에 해당하는 digit이다.

여기서,

$$Q = q_0 + q_1 \cdot 4^{-1} + q_2 \cdot 4^{-2} + \ \ldots \leq q_0 + 2(4^{-1} + 4^{-2} + \ldots)$$

$$= q_0 + 2 \cdot \frac{4^{-1}}{1 + 4^{-1}} = q_0 + \frac{2}{3}$$

이므로, $Q \geq \frac{2}{3}$ 인 값을 나타내기 위해 $q_0 = 1$로 초기화시킨다.

SRT 제곱근 연산 알고리즘의 세부동작은 다음과 같다.

[0] 초기과정

입력오퍼랜드를 오른쪽으로 2비트 쉬프트 하여 radicand(A)를 얻는다. 또한, P_0 및 Q_0를 초기화시킨다.

$$P_0 = A - 1 \ , \quad Q_0 = 1$$

[1] 쉬프트 된 나머지 $(4 \bullet P_j)$ 생성 → P_j를 왼쪽으로 2비트 쉬프트로 구현

[2] 루트비트 (q_{j+1}) 결정 → 샘플된 Q_j와 $4 \bullet P_j$로부터 결정

[3] $F_j = -q_{j+1} \bullet (2Q_j + q_{j+1} \bullet 4^{-(j+1)})$

[4] $P_{j+1} = 4 \bullet P_j + F_j$

최종결과는 1비트 왼쪽으로 쉬프트 된 루트이다.

2.3 뉴턴-랍손 역제곱근 알고리즘

뉴턴-랍손 알고리즘은 피제수에 제수의 역수를 곱하여 최종 몫을 구하는 알고리즘으로, 나누는 수의 역수를 구하는 기본함수 f(x)를 정의하고, f(x) = 0 이 되는 해를 구하는 알고리즘이다. 함수 f(x)는 다음과 같이 정의한다.

$$f(x) = F - \frac{1}{\sqrt{x}} = 0 \tag{3}$$

x_0를 처음의 근사값이라 하고, x_i를 i번째 근사값이라 하면, $i+1$번째 근사값 x_{i+1}은 다음과 같이 구할 수 있다.

$$x_{i+1} = x_i - \frac{f(x)}{f'(x)}$$

따라서,

$$x_{i+1} = \frac{x_i(3 - Fx_i^2)}{2} \tag{4}$$

i번째 오차를 e_i라 하면, $x_i = \frac{1}{\sqrt{F}} + e_i$가 되며, 이것을 식 (4)에 대입하여 정리하면

$$x_{i+1} = \frac{1}{\sqrt{F}} - (\frac{3\sqrt{F}e_i^2}{2} + \frac{Fe_i^3}{2}) = \frac{1}{\sqrt{F}} - e_{i+1}$$

가 된다. 이때 e_{i+1}은 다음과 같다.

$$e_{i+1} < \frac{3\sqrt{F}e_i^2}{2} + \frac{Fe_i^3}{2}$$

여기서, $e_{i+1} \gg e_i^3$ 이므로, $e_{i+1} \propto e_i^2$ 이 된다. 즉, $e_{i+1} = \frac{3\sqrt{F}e_i^2}{2}$ 이다.

e_{i+1}이 충분히 작으면, 알고리즘의 반복을 종료한다.

2.4 오차 보정 방법

뉴턴-랍손 알고리즘은 피제수에 제수의 역수를 곱하는 연산을 반복 수행하여 최종 몫을 구하는 알고리즘이다. 따라서 반복 연산을 할 때 근사값으로 연

산을 하므로 연산 과정에 오차를 포함하게 되고 최종 결과에도 오차가 포함된다. 그래서 정확한 몫을 구하기 위해서는 오차 보정을 해 주어야 한다. 이 절에서는 Oberman이 [12]에서 설명한 오차를 보정하기 위해 구현된 방법을 소개한다.

오차를 보정하여 정확한 값을 구하기 위해 구현된 중요한 기술은 세 가지가 있다.

첫째는 IBM 360/91 시스템에 구현된 골드스미트 알고리즘을 사용한 나눗셈 연산이다. 이 시스템은 계산된 몫의 정밀도를 위해서 10개의 여유 비트를 가지고 있다. 가드 비트인 최하위 비트에 1이 더해진다. 만약 10개의 가드 비트들이 모두 1이면 몫은 반올림되어 보정된다.

이 구현은 보정을 빨리한다는 이점이 있는데, 그 이유는 반복을 완료하는 전체 과정에서 어떤 추가적인 연산을 요구하지 않기 때문이다. 그런데 이것은 “round-to-nearest” 모드를 고려했을 때 결과들이 IEEE 규정에 맞는 것이 아니다. 이 구현은 정확한 반올림의 개념을 가지고 있지 않다.

두 번째는 IBM RS/6000 시스템의 나눗셈 구현에 사용된 방법이다. 이 방법은 데이터 경로 폭이 최종 결과 데이터 경로 폭의 두 배가 요구된다. 몫은 마지막 몫의 두 배 정밀도보다 조금 더 계산된다. 그리고 확장된 결과는 최종 정밀도로 반올림 되어 보정된다.

이 시스템 구현에는 반복 연산 전체에서 2n 비트 보다 더 큰 정밀도를 보증하는 연산 모두를 위한 곱셈-누산기를 융합해서 사용했다. 추가적인 반복을 완료하면 몫은 다음 식과 같이 된다.

$$Q' = estimate\ of\ Q = \frac{a}{b}\ accurate$$

나머지는 다음 식과 같이 계산된다.

$$R = a - b \times Q'$$

반올림되어 보정된 몫은 다음과 같이 계산된다.

$$Q'' = Q' + R \times b$$

여기서, 최종 곱셈-누산은 반올림 모드에서 요구된 정확히 반올림된 결과를 제공한 것을 가져온다.

이 방법의 단점은 추가적으로 한 번의 완전한 알고리즘의 반복을 요구하는 것과 반올림 하지 않은 결과가 요구하는 데이터 경로 폭보다 적어도 두 배 큰 데이터 경로 폭을 요구하는 것이다.

세 번째 방법은 TI8847 FPU(Floating Point Unit)에 사용된 것이다. 이 구조에서도 몫은 정밀도를 나타내는 여유 비트를 가지고 계산된다. 그러나 데이터 폭은 요구되는 최종 몫의 데이터 폭의 두 배보다는 적다. 스티키 비트를 결정하기 위해서 최종 나머지는 다음 식으로부터 직접 계산된다.

$$Q = \frac{a}{b} - R$$

$$R = a - b \times Q$$

이 계산은 나머지의 실제 크기를 계산하기 위해 필요한 것이 아니라 0이 되는 관계성을 필요로 한다.

최악의 경우인 전체 데이터 폭 뺄셈은 아마도 실제 나머지 R 형태로 사용

될 것이다. n 비트 입력 연산자들에 대해서 모든 중간 곱셈들이 적어도 2n 비트의 데이터 폭을 가지는 반복 연산들이 끝까지 충분한 정밀도를 가진다고 가정하면, 계산된 몫은 정확히 반올림된 결과를 향해 수렴한다. 그러나 수렴한 값은 정확한 값의 위 또는 아래 중에서 하나일 것이다.

이 경우에 나머지의 부호는 또한 실제 몫과 관련이 있는 추정된 몫의 위치 검출을 필요로 한다. 그래서 이 방법에서 정확한 반올림 사용을 지원하기 위해 알고리즘의 지연이 적어도 $Q \times b$형태를 위해 필요한 곱셈 지연과 0 검출 그리고 마지막 나머지에서 부호 검출 논리뿐만 아니라 가능한 전체 폭 뺄셈 지연에 의해서 증가한다.

TI8847에서 근사치로 계산된 몫과 실제 몫의 관련성은 명백한 나머지 R의 계산 없이 a와 $Q \times b$ 둘 다의 6 비트 조합 논리로 결정된다. Schwarz에 의하면 이 기술은 확실한 경우에 최종 나머지 계산을 피하기 위해 존재했다.

몫에서 정밀도를 나타내는 한개 여유 비트 계산에 의해서 보정을 하는 모든 경우들의 반이 완료될 수 있다. 그리고 단지 여유 가드 비트의 정밀 검사에 의해서 명백한 나머지 계산이 필요 없어진다.

Oberman은 근사치 계산된 몫에 있는 m개 가드 비트를 사용해서 후위 곱셈과 나머지 형태를 위한 뺄셈들이 모든 경우들 중에서 단지 2^{-m}만 필요로 한다는 것을 보였다.

또한 실제 나머지가 필요할 때, 전체 폭 뺄셈은 단지 피제수의 최하위 비트, 몫의 후위 곱셈 계산 그리고 곱셈기로부터 스티키 비트를 포함한 제수를 사용한 간단한 조합 논리에 의해서 대체될 수 있다. 이런 기술들의 조합은 평균적인 나눗셈의 성능을 향상시킨다.

추가적인 방법들은 뉴턴-랍손 구현들에서 부호 비트 곱셈기에 사용한 보정 방법을 위해서 제안된 것이다. 부호 비트는 뺄셈의 제거 혹은 반복 주기의 보수를 위한 표시를 허용한다. 이 구조에서 최종 곱셈과 메모리 접근을 포함해

서 9 사이클 내에 정확히 보정된 몫을 구하는 것이 가능하다.

곱셈기에서 부분 곱셈들의 중복된 이진 기록은 정확한 스티키 비트를 간단히 생성하기 위해서 허용된다. 곱셈기에서 스티키 비트와 특별한 기록회로의 사용은 단지 알고리즘에서 하나의 사이클만 추가해서 정확한 IEEE 보정이 가능하다.

Ⅲ. 제곱근 알고리즘

IEEE-754로 규정되는 부동소수점의 형식은 $1.f_2 \times 2^{n+bias}$ 이다. 제곱근 $\sqrt{F}$는 n이 짝수일 경우에는 $\sqrt{1.f_2} \times 2^{\frac{n}{2}+bias}$ 이고, n이 홀수일 경우에는 $\sqrt{1.f_2 \times 2} \times 2^{\frac{n-1}{2}+bias}$ 로 변환한다. 따라서, $\sqrt{F} = \sqrt{K} \times 2^e$ 이다.

이때 실수부 K의 영역은 $1 < K < 4$ 이다.

가수부 $1.f_2$는 단정도 실수에서 24 비트, 배정도 실수에서는 53 비트이므로, 본 논문에서는 단정도 실수 연산에는 '32 × 32 = 64 비트' 곱셈기를, 배정도 실수 연산에는 '64 × 64 = 128 비트' 곱셈기를 사용한다. 이들 배정도 곱셈기는 정수 곱셈기와 동일한 사양이다. 또한 가수부 $1.f_2$는 좌정렬 형식을 사용한다. 즉, 가수부 $1.f_2$ or $1.f_2 \times 2$는 그림 3 및 그림 4와 같이 표현된다.

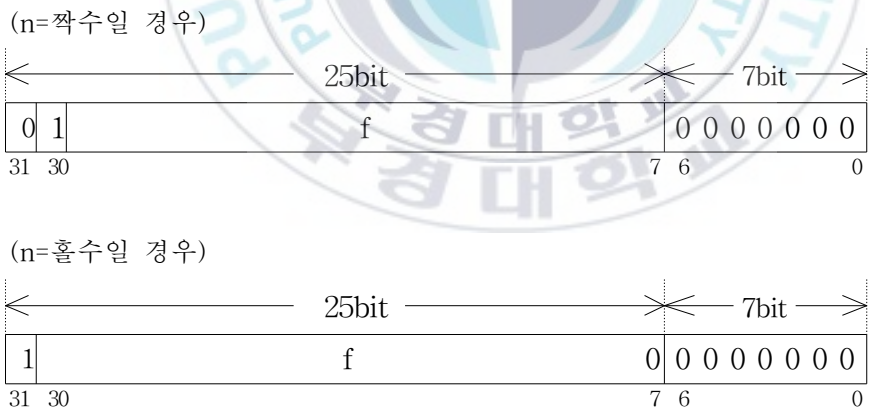

그림 3. 단정도 실수에서 표현되는 부동소수점 형식

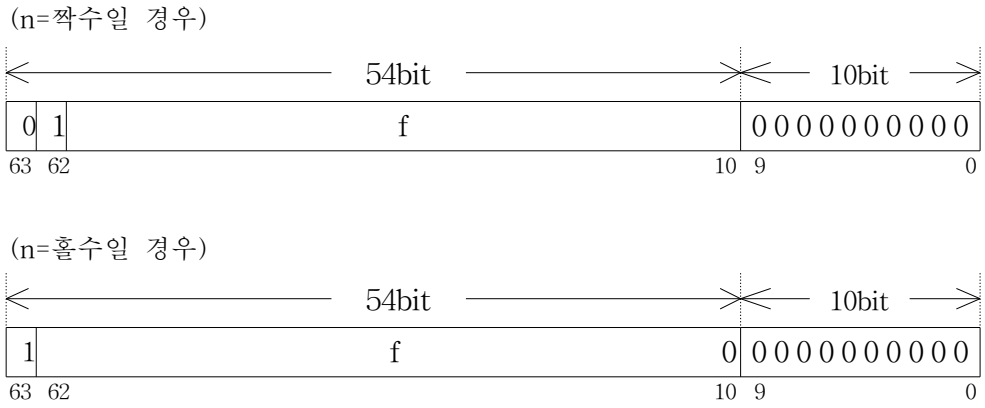

그림 4. 배정도 실수에서 표현되는 부동소수점 형식

가수부 $1.f_2$는 좌정렬(Left Justified) 시키고, 나머지 bit에는 '0'을 채워 넣는다.

부동소수점의 가수부 $1.f_2$는 식 (5)와 같이 두 부분으로 나눌 수 있다.

$$K = 1.f = 1.g + h \tag{5}$$

식 (5)에서 g와 h의 길이를 각각 n_g 및 n_h비트로 정의한다. h는 $0 \le h < 2^{-n_g}$이며, h의 최대값, $h_{\max} = 2^{-n_g} - 2^{-n_g - n_h}$이다.

뉴턴-랍손 알고리즘에서 실수 K의 역제곱근을 구하기 위해 함수 $F(X)$를 (3)과 같이 정의하고, (4)의 반복식을 구할 수 있다.

반복식, 식 (4)의 수렴속도를 빠르게 하기 위해서 $\frac{1}{\sqrt{1.g}}$을 근사계산하여 테이블 T(g)를 미리 작성해 놓는다. T(g)의 값은 $\frac{1}{\sqrt{1.g}}$의 근사계산이므로 오차

e_t를 포함하여 $T(g) = \frac{1}{\sqrt{K}} + e_t$가 된다. T(g)를 X의 초기 근사값 X_0로 정의한다.

초기 근사값 X_0를 식 (4)에 대입하여 반복 계산하면 K의 근사 역제곱근 $X_n \fallingdotseq \frac{1}{\sqrt{K}}$을 구할 수 있다. 근사 역제곱근 X_n에 가수부 K를 곱하면 식 (6)과 같이 근사 제곱근 Q_n이 된다.

$$Q_n = X_n \times K = \sqrt{K} - e \tag{6}$$

X_n이 근사값이므로 Q_n은 오차 e를 가지게 된다. 근사 제곱근 Q_n이 가지는 오차 e를 계산하면 다음과 같다.

$$\begin{aligned} Q_n^2 &= (\sqrt{K} - e)^2 \fallingdotseq K - 2e\sqrt{K} \\ K - Q_n^2 &\fallingdotseq 2e\sqrt{K} \\ e &= \frac{K - Q_n^2}{2\sqrt{K}} = \frac{Q_n(1 - X_n Q_n)}{2} \end{aligned} \tag{7}$$

식 (7)에서 구한 오차 e를 Q_n에 더하여 Q_{n+1}을 구하면 식 (8)이 된다.

$$Q_{n+1} = Q_n + e = Q_n \times \frac{3 - X_n Q_n}{2} = \sqrt{K} - \frac{e^2}{\sqrt{K}} \tag{8}$$

Q_{n+1}에 포함된 오차는 $\frac{e^2}{\sqrt{K}}$이 되어 Q_n에 포함된 오차의 자승에 비례하여

줄어든 것을 알 수 있다.

Q_{n+1}은 근사 계산값이므로 이를 보정하기 위한 상수를 표 1과 같이 정의한다.

표 1. 제곱근 계산을 위한 보정 상수

```
if DOUBLE_PRECISION
    WS = 64  // word size
    FS  = 53 // floating point precision

if SINGLE_PRECISION
    WS = 32
    FS  = 24

MK = (-1) << (WS - FS - 3) ;
RD = 1 << (WS - FS - 4) ;
HW = ( 1 << (WS/2) ) - 1 ;
TMK = ( 1 << (WS - FS - 2) ) - 1 ;
TBIT = 1 << (WS - FS - 3) ;
```

Q_{n+1}을 반올림하고 유효 비트만을 남기기 위하여 보정 상수를 대입하여 식 (9)와 같이 제곱근 Q를 구한다.

$$Q = (Q_{n+1} + RD)\ AND\ MK \tag{9}$$

Q를 제곱하여 스티키 비트(sticky bit)를 다음과 같은 알고리즘에 의하여 구하고, 동시에 Q를 보정하면 $Q = \sqrt{K}$가 된다.

```
ST = 1 ;    /* sticky bit */
Y = MSB(Q * Q) ;
if ( NOT(Y AND TMK) )
    if ( (Q AND HW) == 0 ) ST = 0 ;
        else Q = Q - (RD<<1) ;
else if ( NOT(Y AND TBIT) ) Q = Q - (RD<<1) ;
```

Ⅳ. 알고리즘 구현 및 실험결과

본 논문에서 제안한 제곱근 알고리즘을 C 언어로 프로그램을 작성하였다.

식 (4)의 빨셈의 캐리 전파 지연 시간을 줄이기 위하여 식 (10)의 근사계산을 사용한다.

$$X_{i+1} = \frac{X_i(3-\epsilon-KX_i^2)}{2} \tag{10}$$

ϵ은 곱셈 워드의 LSB(least significant bit)이다. 즉, '3-ϵ'은 단정도 실수에서 0xBFFFFFFF이고, 배정도 실수에서는 0xBFFFFFFFFFFFFFFF이다.

근사 테이블 T(g)는 다음과 같이 작성했다.

$$\mathrm{T(g)} = \frac{1}{\sqrt{1.g}} \fallingdotseq \mathrm{RN}\left(\frac{1}{\sqrt{1.g + 2^{-n_g-1}}}\right)$$

RN = Round to nearest

근사 테이블 T(g)의 크기에 따라서 식 (4)의 반복 연산 회수는 표 2와 같다.

표 2. 근사 테이블 크기에 따른 반복 연산 회수

	Length of g	No. of iteration
Single precision	4 bit	2
	7 bit	1
Double precision	4 bit	3
	8 bit	2

본 연구에서는 단정도 실수와 배정도 실수에서 표 2에 보인 근사테이블 모두에 대하여 상태기계흐름도를 작성했고, 또한 이를 C로 모델링하여 프로그램으로 작성하여 동작을 확인했다.

표 3에 512×8비트 근사테이블을 사용한 배정도 실수에서의 제곱근 연산상태기계흐름을 보인다. 또한 배정도 실수 제곱근 계산기의 블록도를 그림 5에 나타내었다.

상태기계흐름을 보면, 우선 8비트짜리 근사테이블을 사용하였다. 입력은 가수부 $1.f$이고, 출력은 64비트 $1.f$의 제곱근값 Q와 1bit의 Sticky bit, ST, 그리고 1bit의 Guard bit, G이다.

배정도 실수 연산으로 64×64=128bit 곱셈기를 사용하여, 상위 64bit만을 곱셈결과로 사용한다. 각 단계별로 자세히 살펴보면,

우선 state-0에서 $1.f$를 왼쪽으로 정렬시키고, ROM 테이블에서 $\frac{1}{\sqrt{1.g}}$의 값을 읽어와 X레지스터에 저장한다.

state-1에서 state-3까지는 뉴턴-랍손 역제곱근 알고리즘의 반복식을 계산하는 것으로 $\frac{1}{\sqrt{K}}$의 근사값을 계산한다. 즉, X를 제곱하여 K를 곱하고, $3-\epsilon$에

표 3. 배정도 실수 제곱근 연산 상태흐름

```
      *** Table size is 512 X 8
   INPUT : 1.f
   OUTPUT : Q : 64 bit register -- SQRT of 1.f
            ST:  1 bit register -- Sticky bit
            G :  1 bit register -- Guard bit
   Internal : X, Y : 64 bit register
   Function : mulhu(A, B) = High 64 bit
                    of 64 X 64 = 128 bit unsigned multiply result

state-0 : Left justify 1.f ==> K
          TABLE() ==> X
state-1 : mulhu(X, X) ==> Y
state-2 : 0xBFFFFFFFFFFFFFFF - mulhu(K, Y) ==> Y
state-3 : mulhu(Y, X) << 1 ==> X
state-4 : mulhu(X, X) ==> Y
state-5 : 0xBFFFFFFFFFFFFFFF - mulhu(K, Y) ==> Y
state-6 : mulhu(Y, X) << 1 ==> X
state-7 : mulhu(X, K) ==> Y
state-8 : 0xBFFFFFFFFFFFFFFF - mulhu(X, Y) ==> X
state-9 : mulhu(X, Y) << 2 ==> X
state-10 : (X + 0x80) & 0xFFFFFFFFFFFFFF00 ==> X
state-11 : mulhu(X, X) ==> Y
           1 ==> ST
state-12 : IF ( (Y & 0x1FF) == 0  )
            THEN
               IF ( (X & 0xFFFFFFFF) == 0 )
                  THEN 0 ==> ST
                  ELSE X = X - 0x100
            ELSE  IF ( bit 8 of Y == 0 )
                     THEN X = X - 0x100
           bit 10 of X ==> G
           X >> 11 ==> Q
```

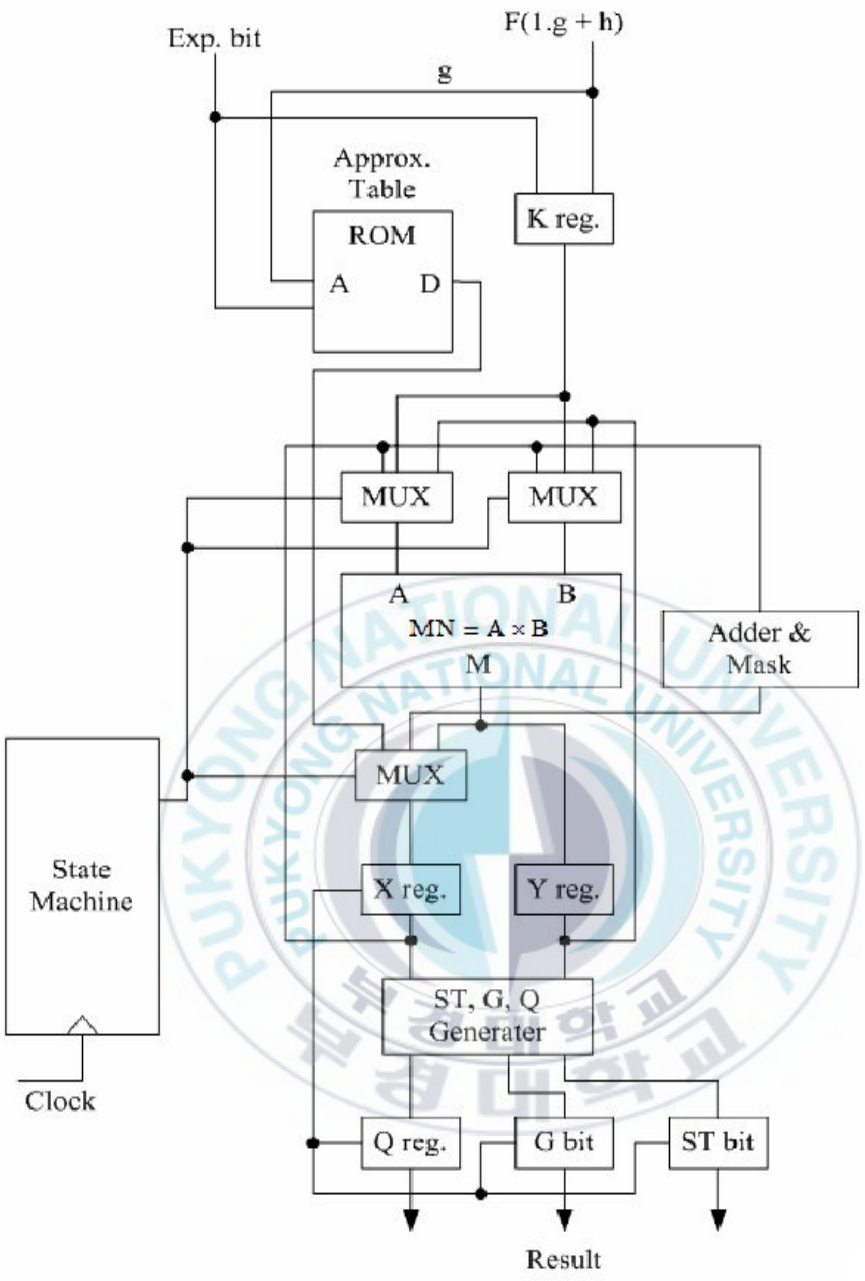

그림 5. 배정도 실수 제곱근 계산기의 블럭도

서 이 값을 빼고, 다시 여기에 X를 곱하고, 2로 나누기위해 왼쪽으로 1bit 시프트연산 시킨다.

state-4에서 state-6은 state-1에서 state-3까지의 반복이다. 배정도실수에서 8bit ROM 테이블을 사용할 때는 반복식을 2번 사용하면 $\frac{1}{\sqrt{K}}$의 근사값이 구해진다.

state-7에서 state-9까지는 Q_{n+1}을 구하는 과정이다. state-7에서 $\frac{1}{\sqrt{K}}$의 근사값 X_n에 K를 곱하여 Q_n을 구하고, state-8에서는 이 값에 다시 X_n을 곱한 후 $3-\epsilon$ 에서 이 값을 빼주고, state-9에서는 다시 Q_n을 곱하고, 왼쪽으로 2bit 시프트 시켜 Q_{n+1}을 구하게 된다.

state-10은 Q_{n+1}을 보정하기 위한 단계로, 반올림하고, 마지막 8bit를 0으로 만들어준다.

state-11과 state-12는 sticky bit와 Guard bit를 구하는 과정이다.

state-11은 state-10에서 보정하여 얻은 값 Q를 제곱하여 Y레지스터에 저장하고, stick bit, ST을 '1'로 만들어준다.

state-12에서는 Y와 'Ox1FF'를 AND연산하여 '0'인지 아닌지를 판단한다. '0'일 경우, Q와 'OxFFFFFFFF'를 AND연산하여, '0'일 경우 ST를 '0'으로 하고, '0'이 아닐 경우 Q에서 'Ox100'을 빼준다. 반면, Y와 'Ox1FF'를 AND 연산했을 때 '0'이 아닐 경우 Y의 8번 비트가 0이면 Q에서 'Ox100'을 빼준다. Q의 10번 비트는 Guard bit가 되고, Q를 11비트 오른쪽으로 시프트 하여 우리가 구하려는 값 Q를 구한다.

단정도 실수에서는 K의 전 영역의 수를 계산하여 모두 정확한 제곱근 값이 구해지는 것을 확인하였으며, 배정도 실수에서는 RC5 암호 알고리즘을 이용하여 10억 개의 무작위 수를 생성하여 계산한 결과 모두 정확한 제곱근 값이

구해지는 것을 확인했다.

Ⅴ. 비 교

본 연구의 결과를 SRT 제곱근 알고리즘과 비교하여 그 결과를 표 4에 나타내었다.

단정도 실수형일 경우, SRT 제곱근 알고리즘은 Mod-4는 Q=2bit짜리를 사용하므로, 24비트를 계산하기 위해 12clock을 계산하고, 초기화와 rounding을 합쳐 14clock이 소요되며, Mod-8에서는 10clock만에 계산된다. 본 연구에서는 g=4비트 테이블(ROM size = 32 × 4 bit)을 사용할 경우에는 13clock, g=7비트 테이블(ROM size = 256 × 7 bit)을 사용할 경우에는 10clock으로 SRT와 큰 차이를 보이지는 않았다.

반면, 배정도형식일 경우, SRT 알고리즘은 Mod-4와 Mod-8에서 각각 29clock, 20clock이 소요되는데 반해, 본 연구의 알고리즘에서는 g=4비트 테이블(ROM size = 32 × 4 bit)을 사용할 경우 16clock, g=8비트 테이블(ROM size = 512 × 8 bit)을 사용할 경우 13clock으로 SRT 보다 훨씬 빠른 속도로 계산해 내는 것으로 나타났다.

표 4. 본 연구 결과와 SRT 제곱근 알고리즘의 연산회수 비교

(단위: clock)

구분	SRT		본 연구	
Single Precision	Mod-4	14	g = 4 bit (32 × 4 bit)*	13
	Mod-8	10	g = 7 bit (256 × 7 bit)*	10
Double Precision	Mod-4	29	g = 4 bit (32 × 4 bit)*	16
	Mod-8	20	g = 8 bit (512 × 8 bit)*	13

* ()는 ROM size를 나타냄.

Ⅵ. 결 론

부동소수점 나눗셈 및 제곱근은 뺄셈을 반복하는 SRT 알고리즘과 곱셈을 반복하는 뉴턴-랍손 알고리즘 및 골드스미트 알고리즘이 있다. 뉴턴-랍손 알고리즘은 근사값을 초기값으로 하고 반복 연산으로 오차를 줄여나가는데 반복 연산을 수행할 때마다 오차는 자승에 비례하여 줄어든다.

본 논문에서는 뉴턴-랍손 부동소수점 역제곱근 알고리즘에서 IEEE-754에서 요구하는 정확한 제곱근을 구하는 알고리즘을 제안했다.

제안한 알고리즘은 뉴턴-랍손 부동소수점 역제곱근 알고리즘을 이용하여 실수 K의 근사 역제곱근 X_n을 구하고, X_n에 K를 곱하여 근사 제곱근 Q_n을 구했다. 그리고 Q_n이 가지는 오차를 계산하여 이 오차의 자승에 비례하는 오차를 가지는 Q_{n+1}을 구했다. Q_{n+1}을 보정하고 스티키 비트를 구하기 위한 보정 상수를 정의하고, 이를 이용하여 Q_{n+1}을 보정하고 $\sqrt{K}$와 스티키 비트를 계산했다.

본 논문에서 제안한 제곱근 알고리즘을 C 언어로 프로그램을 작성해서, 단정도 실수에서는 전수 계산을 했고, 배정도 실수에서는 10억 개의 무작위 수를 생성하여 계산한 결과 모두 정확한 제곱근이 계산되는 것을 확인했다.

또한, SRT 알고리즘과 비교했을 때, 배정도형식에서 SRT보다 2배 가까이 빠르다는 것을 알 수 있었다.

본 논문에서 제안한 알고리즘은 곱셈 연산만을 사용하므로 실장제어용기기, 휴대용기기 등 정확한 제곱근 연산을 요구하는 분야에서 사용될 수 있다.

요 약

부동소수점 나눗셈 및 제곱근 연산은 곱셈을 반복하는 뉴턴-랍손 알고리즘과 골드스미트 알고리즘, 뺄셈을 반복하는 SRT 알고리즘이 있다. 본 논문에서는 IEEE-754에서 요구하는 정확한 제곱근을 배정도 정수 곱셈기를 사용하여 연산하는 제곱근 알고리즘을 제안한다.

즉, 뉴턴-랍손 알고리즘을 이용하여 근사 역제곱근을 구하고, 이의 오차를 줄이면서 제곱근을 구하는 알고리즘과 계산된 제곱근을 보정하는 알고리즘을 제안한다. 제안한 알고리즘은 단정도 실수에서는 전수 조사를 통해서, 배정도 실수에서는 10억개의 무작위 수에서의 계산을 통하여 모두 정확한 값을 얻어서 알고리즘을 검증했다.

본 논문에서 제안한 알고리즘은 배정도 정수 곱셈기만을 사용하므로 부동소수점 연산을 위한 별도의 하드웨어가 필요하지 않다. 따라서 실장제어용기기, 휴대용기기 등 정확한 제곱근 연산을 요구하는 분야에서 사용될 수 있다.

참 고 문 헌

[1] S. F. Oberman and M. J. Flynn, "Design Issues in Division and Other Floating Point Operations", IEEE Transactions on Computer, Vol. C-46, p. 154-161, 1997.

[2] C. V. Freiman, "Statistical Analysis of Certain Binary Division Algorithm", IRE Proc., Vol. 49, p. 91-103, 1961.

[3] S. F. McQuillan, J. V. McCanny, and R. Hamill, "New Algorithms and VLSI Architectures for SRT Division and Square Root", Proc. 11th IEEE Symp. Computer Arithmetic, IEEE, p. 80-86, 1993.

[4] D. L. Harris, S. F. Oberman, and M. A. Horowitz, "SRT Division Architectures and Implementations", Proc. 13th IEEE Symp. Computer Arithmetic, Jul., 1997.

[5] M. Flynn, "On Division by Functional Iteration", IEEE Transactions on Computers, Vol. C-19, No. 8, p. 702-706, Aug., 1970.

[6] R. Goldschmidt, Application of division by convergence, master's thesis, MIT, Jun., 1964.

[7] M. D. Ercegovac, et al., "Improving Goldschmidt Division, Square Root, and Square Root Reciprocal", IEEE Transactions on Computer, Vol. 49, No. 7, p. 759-763, Jul., 2000.

[8] D. L. Fowler and J. E. Smith, "An Accurate, High Speed Implementation of Division by Reciprocal Approximation", Proc. 9th IEEE symp. Computer Arithmetic, IEEE, p. 60-67, Sep., 1989.

[9] S. Oberman, "Floating Point Division and Square Root Algorithms and Implementation in the AMD-K7 Microprocessors", Proc. 14th IEEE Symp. Computer Arithmetic, p. 106-115, Apr., 1999.

[10] IEEE, IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard, Std. 754-1985.

[11] 이원, 권호경, 이용환, 이용석, “Radix-4 SRT 알고리즘을 사용한 나눗셈/제곱근 연산기에 관한 연구”, 전자공학회논문지, 제 33권, A편, 제 9호, 1996.

[12] Stuart F. Oberman, Michael J. Flynn, "Division Algorithms and Implementations", IEEE TRANSACTIONS ON COMPUTERS, VOL. 46, N0. 8, p. 833-854, AUGUST 1997.

감사의 글

본 연구를 수행하는 동안 끊임없는 지도를 하여주신 조경연교수님께 깊은 감사를 드리며, 아낌없는 지도를 하여주신 서경룡교수님과 권오흠교수님께 감사드립니다.

그리고, 언제든지 충고와 조언을 잊지 않으신 박창수선배님께 감사드리며, 마이크로프로세서연구실 선후배 여러분께도 감사드립니다.

끝으로, 오늘이 있기까지 헌신적인 사랑으로 지혜와 용기를 주신 부모님과 물심양면으로 밀어준 남편, 그리고 엄마 없이 밝게 잘 자라준 귀여운 딸 영선이에게 감사의 마음을 전합니다.