곱셈기를 이용한 정확한 부동소수점 제곱근 계산기

조 경 연*

An exact floating point square root calculator using multiplier

Gyeong Yeon Cho*

요 약

부동소수점 제곱근 연산은 곱셈을 반복하여 근사값을 계산하는 뉴턴-랍손 알고리즘 및 골드스미트 알고리즘과 뺄셈을 반복하여 정확한 값을 계산하는 SRT 알고리즘이 있다. 본 논문에서는 곱셈기를 사용하여 정확한 값을 계산하는 제곱근 알고리즘을 제안한다. 본 논문에서는 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 10억 개의 무작위 수를 계산하여 모두 정확한 값을 얻었다. 본 논문에서 제안한 알고리즘은 곱셈기만을 사용하므로 별도의 하드웨어가 필요하지 않다. 따라서 실장제어용기기, 휴대용기기 등 정확한 제곱근 연산을 요구하는 분야에서 사용될 수 있다.

ABSTRACT

There are two major algorithms to find a square root of floating point number, one is the Newton_Raphson algorithm and GoldSchmidt algorithm which calculate it approximately by iterating multiplications and the other is SRT algorithm which calculates it exactly by iterating subtractions. This paper proposes an exact floating point square root algorithm using only multiplication. At first an approximate inverse square root is calculated by Newton_Raphson algorithm, and then an exact square root algorithm by reducing an error in it and a compensation algorithm of it are proposed. The proposed algorithm is verified to calculate all of numbers in a single precision floating point number and 1 billion random numbers in a double precision floating point number. The proposed algorithm requires only the multipliers without another hardware, so it can be widely used in an embedded system and mobile production which requires an exact square root of floating point number.

키워드

제곱근, 부동소수점, 곱셈기, 뉴턴-랍손 알고리즘

* 부경대학교 전자컴퓨터정보통신공학부 교수

접수일자 2009. 02. 25
심사완료일자 2009. 03. 20

Ⅰ. 서 론

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

부동소수점 제곱근 계산은 뺄셈을 반복하여 정확한 값을 계산하는 SRT[3-4] 알고리즘과 곱셈을 반복하여 근사값을 계산하는 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 정밀 계산이 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[10-11].

본 논문에서는 뉴톤-랍손 역제곱근 알고리즘을 이용하여 근사적인 역제곱근을 구하고, 이때의 오차의 자승에 비례하는 오차를 가지는 제곱근 알고리즘을 구하고, 계산된 제곱근을 보정하면서 스티키 비트(sticky bit)를 아울러 구하는 알고리즘을 제안한다.

본 논문에서 제안하는 알고리즘을 C 언어로 모델링하여 동작을 검증했다. 또한 단정도 실수에서는 전수 조사를 통해서, 배정도 실수에서는 10억 개의 무작위 수에서의 계산을 통하여 모두 정확한 값을 얻어서 정확도를 검증했다.

본 논문의 구성은 다음과 같다. 2장에서는 SRT 제곱근 알고리즘과 뉴톤-랍손 역제곱근 알고리즘을 소개하고, 3장에서 곱셈기를 사용한 정확한 제곱근 알고리즘을 제안한다. 4장에서 알고리즘을 C 언어로 모델링하여 실험하고, 5장에서 SRT 제곱근 알고리즘과 비교한다. 마지막으로 6장에서 결론을 맺는다.

Ⅱ. 관련 연구

2.1 SRT 제곱근 알고리즘[12]

SRT 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 방식이 있는데, 루트 비트를 수정하지 않고 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 PLA로 설계가 가능하다. Radicand가 식 (1)과 같이 주어졌을 때, 기본적인 radix-4 제곱근 연산 알고리즘의 순환방정식은 식 (2)와 같다.

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

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

여기서

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

이다.

단, P_j는 j번째 연산에서의 부분 나머지(partial remainder), Q_j는 j번째 연산에서의 몫, q_i는 i번째 연산에서의 몫에 해당하는 디지트이다. 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 초기화 시킨다.

SRT 제곱근 연산 알고리즘의 세부동작은 다음과 같다.

[0] 초기과정 - 입력오퍼랜드를 오른쪽으로 2비트 쉬프트하여 radicand(A)를 얻는다. 또한, P_0 및 Q_0를 초기화시킨다. $P_0 = A - 1$, $Q_0 = 1$

[1] 쉬프트된 나머지 $(4 \cdot P_j)$ 생성 → P_j를 왼쪽으로 2비트 쉬프트로 구현

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

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

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

최종결과는 1비트 왼쪽으로 시프트된 루트이다.

2.2 뉴턴-랍손 역제곱근 알고리즘

부동소수점 수 F의 역수 제곱근을 구하기 위해서 함수 f(x)를 식 (3)과 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 식 (4)가 성립한다.

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

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

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

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.

III. 정확한 제곱근 알고리즘

IEEE-754로 규정되는 부동소수점의 형식은 $1.f_2 \times 2^{n+bias}$이다.

제곱근 $\sqrt{F}$는 n이 짝수일 때는 $\sqrt{1.f_2} \times 2^{\frac{n}{2}+bias}$이고, n이 홀수일 때는 $\sqrt{1.f_2 \times 2} \times 2^{\frac{n-1}{2}+bias}$로 변환한다. 따라서, $\sqrt{F} = \sqrt{K} \times 2^e$ 이다. 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 X 32 = 64 비트' 곱셈기를, 배정도 실수 연산에는 '64 X 64 = 128 비트' 곱셈기를 사용한다. 또한 가수부 $1.f_2$는 좌정렬 형식을 사용한다. 즉, 가수부 $1.f_2$는 좌정렬(Left Justified)시키고, 나머지 비트에는 '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 식 (3)과 같이 정의하고, (4)의 반복식을 구할 수 있다.

반복식 (4)의 수렴속도를 빠르게 하기 위해서 $\frac{1}{\sqrt{1.g}}$를 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.

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

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

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

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
Table 1. Compensation constant of square root

```
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(sticky) 비트를 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

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

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

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

근사 테이블 T(g)는 다음과 같이 작성한다.

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

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
Table 2. Iteration by an approximate table

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

본 연구에서는 단정도실수와 배정도실수에서 표 2에 보인 근사테이블 모두에 대하여 상태기계 흐름도를 작성했고, 또한 이를 C로 모델링하여 프로그램으로 작성하여 동작을 확인했다. 표 3에 512×8비트 근사테이블을 사용한 배정도실수에서의 제곱근 연산상태기계흐름을 보인다. 또한 배정도실수 제곱근 계산기의 블록도를 그림 1에 보인다.

상태기계흐름을 보면, 입력은 가수부 $1.f$이고, 출력은 64비트 $1.f$의 제곱근 값 Q와 1 비트의 스티키 비트 ST, 그리고 1 비트의 가드(guard) 비트 G이다. 배정도실수연산이므로 64×64=128 비트 곱셈기를 사용하여, 상위 64 비트만을 곱셈결과로 사용한다.

표 3. 배정도실수 제곱근 연산 상태흐름
Table 3. State flow diagram of double precision square root

```
        *** 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
```

상태-0에서 $1.f$를 왼쪽으로 정렬시키고, ROM 테이블에서 $\frac{1}{\sqrt{1.g}}$의 값을 읽어와 X레지스터에 저장한다. 상태-1에서 상태-3까지는 뉴턴-랍손 역제곱근 알고리즘의 반복식을 계산하는 것으로 $\frac{1}{\sqrt{K}}$의 근사값을 계산한다. 즉, X를 제곱하여 K를 곱하고, $3-\epsilon$에서 이 값을 빼고, 다시 여기에 X를 곱하고, 2로 나누기 위해 왼쪽으로 1 비트 시프트시킨다. 상태-4에서 상태-6은 상태-1에서 상태-3까지의 반복이다. 배정도실수에서 8 비트 ROM 테이블을 사용할 때는 반복식을 2번 사용하면 $\frac{1}{\sqrt{K}}$의 근사값이 구해진다.

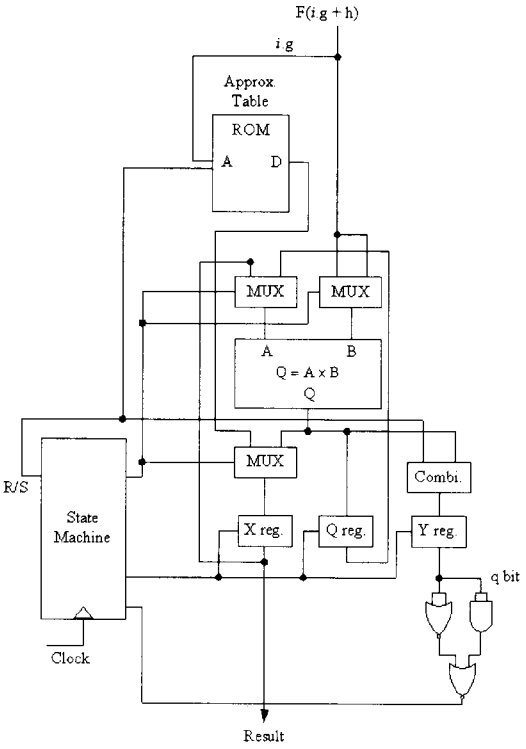

그림 1. 배정도실수 제곱근 계산기의 블록도
Fig. 1. Block diagram of double precision square root

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

상태-10은 Q_{n+1}을 보정하기 위한 단계로, 반올림하고, 마지막 8 비트를 0으로 만들어준다. 상태-11과 상태-12는 스티키 비트와 가드 비트를 구하는 과정이다. 상태-11은 상태-10에서 보정하여 얻은 값 Q를 제곱하여 Y 레지스터에 저장하고, 스티키 비트 ST을 '1'로 만들어준다. 상태-12에서 Y와 'Ox1FF'를 AND 연산하여 '0'인지 아닌지를 판단한다. '0'일 경우, Q와 'OxFFFFFFFF'를 AND 연산하여, '0'일 경우 ST를 '0'으로 하고, '0'이 아닐 경우 Q에서 'Ox100'을 빼준다. 반면, Y와 'Ox1FF'를 AND 연산했을 때에 '0'이 아닐 경우 Y의 8번 비트가 0이면 Q에서 'Ox100'을 빼준다. Q의 10번 비트는 가드 비트가 되고, Q를 11비트 오른쪽으로 시프트하여 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 10억개의 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 보인다.

단정도실수에서 SRT 제곱근 알고리즘은 Mod-4는 Q=2 비트를 사용하므로, 24 비트를 계산하기 위해 12개 클럭이 소요되며 초기화와 반올림을 합쳐 14 클럭이 소요되며, Mod-8에서는 10개 클럭이 소요된다. 본 연구에서는 g=4 비트 테이블(ROM size = 32 × 4 bit)을 사용할 경우에는 13개 클럭, g=7 비트 테이블(ROM size = 256 × 7 bit)을 사용할 경우에는 10개 클럭으로 SRT와 큰 차이를 보이지는 않았다.

표 4. 본 연구 결과와 SRT 제곱근 알고리즘의 연산회수 비교
Table 4. Comparison table of square root algorithm

(Unit: clock)

	SRT		This paper	
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

* () means ROM size.

반면, 배정도실수에서는 SRT 알고리즘은 Mod-4와 Mod-8에서 각각 29개 클럭, 20개 클럭이 소요되는데 반해, 본 연구의 알고리즘에서는 g=4 비트 테이블(ROM size = 32 × 4 bit)을 사용할 경우 16개 클럭, g=8 비트 테이블(ROM size = 512 × 8 bit)을 사용할 경우 13개 클럭으로 SRT보다 훨씬 빠른 속도로 계산할 수 있었다.

VI. 결 론

부동소수점 나눗셈 및 제곱을 계산하는 방법은 뺄셈을 반복하는 SRT 알고리즘과 곱셈을 반복하는 뉴턴-랍손 알고리즘 및 골드스미트 알고리즘이 있다. 뉴턴-랍손 알고리즘은 근사값을 초기 값으로 하고 반복 연산으로 오차를 줄여나가는데 반복연산을 수행할 때마다 오차가 자승에 비례하여 줄어든다.

본 논문에서는 곱셈을 반복하여 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 모델링하여 동작을 검증했다. 단정도 실수에서는 전수 계산을 했고, 배정도 실수에서는 10억 개의 무작위 수를 생성하여 계산한 결과 모두 정확한 제곱근이 계산되는 것을 확인했다. 또한, SRT 알고리즘과 비교했을 때에 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.

참고문헌

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

[2] C. V. Freiman, "Statistical Analysis of Certain Binary Division Algorithm," *IRE Proc.*, Vol. 49, pp. 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*, pp. 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, pp. 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, pp. 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*, pp. 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*, pp. 106-115, Apr., 1999.

[10] 김성기, 송홍복, 조경연, "가변시간 골드스미트 부동소수점 제곱근 계산기," *한국해양정도통신학회 논문지*, 제 9권 제 1호, pp. 188-198, Jan. 2005

[11] 조경연, "개선된 뉴톤-랍손 역수 및 역제곱근 알고리즘," *한국해양정도통신학회 논문지*, 제11권 제1호, pp. 46-55, Jan. 2007

[12] 이원, 권호경, 이용환, 이용석, "Radix-4 SRT 알고리즘을 사용한 나눗셈/제곱근 연산기에 관한 연구," *전자공학회논문지*, 제33권, A편, 제9호, 1996.

저자소개

조경연(Gyeong-Yeon Cho)

1990 인하대학교 대학원 전자공학과 (공학박사)
1983-1991 삼보컴퓨터 기술연구소 책임연구원
1991-2001 삼보컴퓨터 기술연구소 기술고문
1991-현재 부경대학교 공과대학 전자컴퓨터정보통신공학부 교수
1998-현재 에이디칩스(주) 비상임 기술고문
*관심분야 : 전산기구조, 반도체회로설계, 암호 알고리즘