공학석사 학위논문

가변시간 골드스미트 부동소수점 역수 알고리즘

지도교수 조 경 연

이 논문을 [illegible] 제출함

2005년 2월

부경대학교 대학원

컴퓨터공학과

한 경 헌

한경헌의 공학석사 학위논문을 인준함

2004년 12월 일

주 심 공학박사 서 경 룡 (인)

위 원 공학박사 김 종 남 (인)

위 원 공학박사 조 경 연 (인)

차 례

그 림 차 례

표 차 례

A Variable Latency Goldschmidt's Reciprocal Algorithm

Kyoung-Houn Han

Department of Computer Engineering

Graduate School

Pukyong National University

Abstract

Floating point expression is one of the computer number expression. Floating point expression is advantage that can display number of wide extent. Goldschmidt reciprocal algorithm that use much in floating point division repeats multiplication as much as fixed number of times and calculate reciprocal of divisor. This paper propose variable latency Goldschmidt floating point reciprocal algorithm that get reciprocal until enter in validity expression sphere of floating point that must gain changing Goldschmidt floating point reciprocal algorithm.

When floating point is known as F, '$\frac{1}{F} = \frac{T}{TF} = \frac{T}{B}$' becomes if multiply approximation value T of reciprocal $\frac{1}{F}$ of floating point denominator and numerator. In case B is bigger than 1, '$A = B - 1, \frac{1}{F} = T(1-A)(1+A^2)(1+A^4)\cdots$' is calculate until become '$A^i < 2^{\frac{-p}{2}}$'. p is validity cipher that consider cumulative error. If B is smaller than 1, '$A = 1 - B, \frac{1}{F} = T(1+A)(1+A^2)(1+A^4)\cdots$' is calculate until become '$A^i < 2^{\frac{-p}{2}}$'. Because A is integer of positive, circuit that decide '$A^i < 2^{\frac{-p}{2}}$' is circuit that '$\frac{p}{2}$' bit below point of A decide all '0' legal recognitions.

Existent Goldschmidt floating point reciprocal algorithm did arithmetic by fixing as maximum multiplication number of times that must yield maximum error range that must get and do arithmetic. Variable latency Goldschmidt floating point

reciprocal algorithm that propose in this paper can differ multiplication number of times because extent of error according to value that result arithmetic is different. Therefore, can reduce all arithmetic time than existent algorithm. Also, can compose approximation table of most suitable.

Research result of this paper can use comprehensively in field floating point processing as Digital Signal Processing, Computer Graphics, Multimedia Processing, scientific technique.

Ⅰ. 서 론

부동소수점 계산은 디지털 신호처리, 컴퓨터 그래픽스, 멀티미디어 처리, 과학 기술 연산 같은 것에서 폭 넓게 사용하고 있다. 최근에는 휴대용 통신 기기에서 화상 및 음성을 처리하게 되면서 SOC(System On Chip)에서도 하드웨어 부동소수점 계산기를 도입하고 있다. 부동소수점에서 나눗셈은 덧셈, 뺄셈 및 곱셈보다 출현 빈도가 낮지만, Oberman과 Flynn의 연구[1]는 나눗셈의 수행 시간이 덧셈이나 곱셈과 비슷하게 걸리는 것을 보이고 있다. 따라서 나눗셈기의 수행 속도를 높이는 연구가 필요하다.

부동소수점 나눗셈은 뺄셈을 반복하는 SRT[2~4] 알고리즘과 곱셈을 반복하는 뉴톤-랍손(Newton-Raphson) 알고리즘 및 골드스미트(Goldschmidt) 알고리즘[5~8]이 있다. SRT 알고리즘은 나머지를 동시에 구하는 정밀 연산이 가능하지만 수행 시간이 오래 걸리는 문제점이 있다. 한편, 곱셈을 반복하는 방식은 빠른 곱셈기와 근사 테이블을 사용하여 계산을 빨리 할 수 있으나 근사계산만이 가능한 단점이 있다. 그러나 소수의 정밀한 과학 기술 연산 분야를 제외하고 대부분 멀티미디어, 디지털 신호처리, 컴퓨터 그래픽스에서는 근사계산만으로도 실용상 문제가 없다.

곱셈을 반복하여 나눗셈을 하는 방식은 나누는 수의 역수를 구하고 이를 나뉘는 수에 곱해서 나눗셈을 수행한다. 60년대에 발표한 골드스미트 알고리즘[6]은 나누는 수와 나뉘는 수에 반복으로 곱셈하여 나눗셈을 수행하였다. 이것은 역수를 구해서 나뉘는 수에 곱하는 방식과 같다. 본 논문에서는 역수를 구해서 나뉘는 수에 곱하는 방식을 사용한다.

뉴톤-랍손과 골드스미트 역수 계산 알고리즘은 한 번 연산에 2회의 곱셈이 필요하다. 뉴톤-랍손은 2회의 연산이 서로 종속하지만 골드스미트는 서로 독립한다. 따라서 골드스미트 알고리즘에서는 곱셈기를 두 개 사용해서 연산 시간을 줄일 수 있으므로 IBM RS/6000, AMD K7 프로세서 같은 것에서 사용하고 있다[9].

본 논문에서는 골드스미트 부동소수점 역수 알고리즘을 변형하여, 오차가 정해진 값보다 작아지는 시점까지만 반복 연산을 수행하도록 한다. 종래 방식에서는 최대 오차를 계산하고, 항상 일정 회수만큼 반복 수행하였다. 이러한 종래 방식에서는 구하고자 하는 결과 값에 이르렀는데도 필요 이상의 연산을 수행하

여 연산 속도를 떨어뜨리는 단점이 있었다.

본 논문에서 제안하는 알고리즘은 이러한 종래의 단점을 개선하여 역수 계산기의 성능을 높일 수 있다. 또한 요구하는 역수 계산기 성능에 따른 최적의 근사 역수 테이블을 구성할 수 있다.

본 논문의 구성은 다음과 같다. 2장에서는 IEEE 754 부동소수점 형식과 뉴턴-랍손 알고리즘과 골드스미트 역수 알고리즘을 설명한다. 3장에서는 본 논문에서 제안하는 알고리즘을 구현하는데 필요한 알고리즘 변형식을 구하고 오차를 분석해서 연산 자릿수 및 반복 연산을 종료할 오차 한계를 계산한다. 4장에서는 제안한 알고리즘을 구현하는 회로와 상태 기계를 구성한다. 5장에서는 근사 테이블을 구성하고, 역수 계산에 소요되는 평균 곱셈 횟수를 계산한다. 그리고 그 결과를 종래 골드스미트 역수 알고리즘과 비교 분석한다. 6장에서 결론을 맺는다.

Ⅱ. 부동소수점 형식과 반복 연산 알고리즘

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

부동소수점 표현법으로 대표적인 것이 IEEE 754 표준이다[10-12]. IEEE 754 표준은 2진 부동 소수점 연산을 나타내는데 쓰는 표현법으로 1985년에 제정이 되었으며 현재 모든 부동소수점 연산은 이 표준에 따라 구현하고 있다. 이 표준은 데이터 형식, 라운딩, 산술, 형식 변환, 비교를 정의하고 있으며 무한대, NaN, '0' 같은 특수 값과 예외 발생과 이에 따른 처리 및 트랩을 정의하고 있다.

한편, 반복 연산을 하는 나눗셈 알고리즘은 이 형식을 바탕으로 유효범위의 값을 연산한다. 반복연산 나눗셈 알고리즘은 뉴턴-랍손 알고리즘과 골드스미트 역수 알고리즘이 있다. 뉴턴-랍손 알고리즘은 반복곱셈으로 제수의 역수의 근사치를 구해서 피젯수에 곱해서 몫을 구하는 것이고 골드스미트 알고리즘은 반복곱셈으로 제수를 1에 수렴하게 해서 곱셈의 결과 값이 몫에 가까워지게 해서 몫을 구하는 것이다. 이 두 알고리즘은 모두 제수의 역수를 구해서 피젯수에 곱하는 방식으로 같다.

2.1 IEEE 754 부동소수점 표준

IEEE 754 부동소수점 표준에서는 부동소수점 형식을 네 가지로 정의하고 있다. 단정도 형식과 배정도 형식의 기본 형식과 확장 형식으로 정의한다.

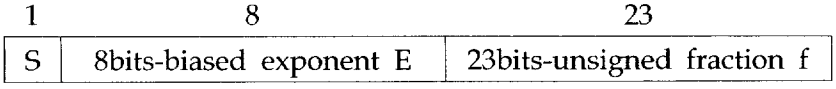


그림 1. 단정도 형식

(그림 1)은 IEEE 754 표준 부동소수점 기본 단정도 형식을 나타낸다. 이 형식

은 부호를 나타내는 1 비트 S , 2의 승수 즉 지수를 나타내는 8 비트 E , 소수점 이하를 나타내는 가수 23 비트 f로 모두 32 비트로 되어 있다.

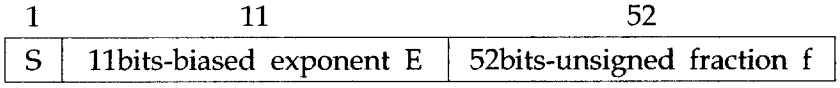


그림 2. 배정도 형식

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

확장형식의 단정도 형식은 지수비트를 11 비트로 늘리고 가수비트를 24 비트에서 32 비트 또는 그 이상으로 확장한 형식이다. 그래서 모두 44 비트 이상이 된다. 또한 확장형식의 배정도 형식은 지수비트를 11 비트에서 15 비트로 늘리고 가수비트를 53 비트에서 64 비트 또는 그 이상으로 확장한 형식이다. 모두 80 비트 이상이 된다. 이 형식들에서 실제로 많이 쓰는 것은 기본형식의 단정도 형식과 배정도 형식과 확장 형식의 배정도 형식이다.

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 single-precision format(단정도 형식)**
  - $E = 255$이고 $f \neq 0$이면 $F = NaN$
  - $E = 255$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 double-precision format(배정도 형식)**
  - $E = 2047$이고 $f \neq 0$이면 $F = NaN$
  - $E = 2047$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 결과 값이 표현 범위를 넘어서게 되는데 표현 범위 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 모두 네 가지가 있다. 다음이 그것이다.

첫 번째, Round to Nearest Even (RNE).
두 번째, Round toward 0 (RZ).
세 번째, Round toward Minus Inifinity (RNI).
네 번째, Roung 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 산술 연산과 '부동소수점 끼리 형식 변환', '부동소수점과 정수의 형식 변환',

'비교'가 있다. '비교'를 할 때에는 'unordered'라 하여 NaN처럼 비교할 수 없는 수를 처리해야 하는 경우를 대비하고 있다.

특수 값에는 무한대(Infinity), NaN(Not a Number), 부호화한 영(Signed Zero)이 있다.

또한 예외처리를 규정하고 있는데 예외에는 크게 다섯 가지가 있다. 첫 번째로 무효 연산, 두 번째로 0으로 나누기, 세 번째로 오버 플로우, 네 번째로 언더플로우, 다섯 번째로 부정확이 있다. 무효 연산은 수행할 연산의 피연산자가 무효할 때 생기며, 0으로 나누기는 0또는 무한대가 아닌 피제수를 0으로 나눌 때에 생기고 오버플로우는 연산 결과가 정규화 수로 나타낼 수 있는 최대 절대 값을 넘어설 때 생긴다. 또한 언더플로우는 연산 결과가 정규화 수로 나타낼 수 있는 최소 절대 값 미만일 때에 생기고 부정확은 라운딩으로 0이 아닌 가수 부분이 잘려서 없어졌을 때에 생긴다. 이 때 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 뉴턴-랍손 알고리즘

뉴턴-랍손 알고리즘은 피젯수에 젯수의 역수를 곱하여 최종 몫을 구하는 알고리즘이다. 나눗셈을 할 때에 몫을 Q , 나뉘는 수를 N , 나누는 수를 D라고 할 때에

$$Q = \frac{N}{D} \quad \text{또는} \quad Q = N \times \frac{1}{D} \qquad \text{--- (1)}$$

로 나타낼 수 있다. 이러한 연산을 수행하여 몫을 결정하려면 나누는 수 D의 역수를 잘 구하는 것이 중요하다. 뉴톤-랍손 알고리즘은 나누는 수의 역수를 구하는 기본 함수 $f(x)$를 정의하고 $f(x)$=0이 되는 해를 구하는 알고리즘이다. 나

누는 수의 역수를 구하는 함수 $f(x)$는 다음으로 정의한다.

$$f(x)=\frac{1}{x}-D=0 \qquad \text{---(2)}$$

x_0을 처음의 근사 값이라 하고 x_i는 i번째 근사 값이라 하면 다음 단계의 근사 값 x_{i+1}은 다음처럼 구할 수 있다.

$$x_{i+1}=x_i-\frac{f(x)}{f'(x)} \qquad \text{---(3)}$$

여기서 $f'(x)$는 $f(x)$의 x에 관한 미분이다. 즉, $f'(x)=-\frac{1}{x^2}$이다.

따라서,

$$x_{i+1}=x_i-\frac{f(x)}{f'(x)}=x_i-\frac{\frac{1}{x_i}-D}{-\frac{1}{x_i^2}}=x_i+(\frac{1}{x_i}-D)x_i^2 \qquad \text{---(4)}$$

$$=x_i+x_i-x_i^2D=2x_i-x_i^2D=x_i(2-x_iD)$$

i번째의 오차를 δ_i라 하면 '$x_i=\frac{1}{D}+\delta_i$'가 되며, 이것을 식-4에 대입해서 정리하면

$$x_{i+1}=\frac{1}{D}-D\delta_i^2=\frac{1}{D}+\delta_{i+1} \qquad \text{---(5)}$$

이 된다.

따라서 '$\delta_0{}^2$ < D'가 되도록 초기값 x_0를 선정하면 δ_n은 '0'으로 수렴한다. 따라서 '$x_n=\frac{1}{D}$'가 된다.

2.3 골드스미트 알고리즘

골드스미트 알고리즘도 뉴턴-랍손 알고리즘과 마찬가지로 곱셈반복방식을 쓰는 알고리즘이다. 나눗셈을 할 때에 몫을 Q , 나뉘는 수를 G_0 , 나누는 수를 F_0라 하면 $Q=\frac{G_0}{F_0}$는 다음같이 연산할 수 있다.

$$\begin{aligned} R_i &= 2 - F_i, \quad i \in \{0, 1, 2, \ldots\ldots, n-1\} \\ G_{i+1} &= G_i \times R_i \\ F_{i+1} &= F_i \times R_i \end{aligned} \qquad \text{---(6)}$$

(식6)에서

$$\begin{aligned} F_1 &= F_0 \times (2 - F_0) = F_0 \times (1 + A) = (1 - A) \times (1 + A) = 1 - A^2 \\ &\quad \text{단, } A = 1 - F_0 \\ F_2 &= F_1 \times (2 - F_1) = (1 - A^2) \times (1 + A^2) = 1 - A^4 \\ F_i &= 1 - A^2 \end{aligned} \qquad \text{---(7)}$$

가 된다. 그러므로 |A| < 1.0 이면 F_n은 1.0에 수렴한다.

따라서 (식6)에서 G_0가 '1.0'이라면 F_0의 역수는 다음같이 된다.

$$\begin{aligned} \frac{1}{F_0} &= R_0 \times R_1 \times R_2 \ldots\ldots \times R_n \\ &= (2 - F_0)(2 - F_1)(2 - F_2) \ldots\ldots (2 - F_n) \\ &= (1 + A)(1 + A^2)(1 + A^4) \ldots\ldots (1 + A^{2^n}) \end{aligned} \qquad \text{---(8)}$$

$$\text{단, } A = 1 - F_0$$

Ⅲ. 가변시간 골드스미트 역수 알고리즘

골드스미트 역수 알고리즘은 반복 곱셈 연산을 해서 나누는 수의 역수를 구해서 구하고자 하는 몫의 근사치를 구하는 알고리즘이다. 부동소수점 연산을 할 때에 연산은 결과 값이 유효범위에 있을 때에만 하면 된다. 기존의 알고리즘은 결과 값이 나타나는 최대 유효범위를 구해서 고정으로 반복 연산을 하였다. 본 논문에서는 결과 값의 크기에 따라 나타나는 유효범위가 다르므로 고정 연산을 하지 않고 유효범위에 맞추어 반복 연산하는 가변연산 알고리즘을 제안한다.

3.1 부동소수점 수 형식과 알고리즘 변형식

IEEE 754 로 규정하는 부동소수점은 $1.f_2 \times 2^{n+base}$이다. 1.f는 단정도 형식에서는 24 비트이고 배정도 형식에서는 53 비트이다. 역수의 지수부 연산은 본 논문에서는 생략한다.

부동소수점 수 F의 가수부 1.f는 다음 (식9)처럼 두 부분으로 나눌 수 있다.

$$1.f = 1.g + h \qquad \text{---(9)}$$

(식9)에서 g와 h의 길이를 각각 n_g, n_h비트로 정의한다. h는 $0 \le h < 2^{-n_g}$이며 h의 최대 값은 $h_{\max} = 2^{-n_g} - 2^{-n_g-n_h}$이다.

F의 역수를 구하는 (식8)의 수렴 속도를 빠르게 하려고 $\frac{1}{1.g}$를 근사계산하여 테이블 $T(g)$를 미리 작성해 놓는다. 근사 테이블은 ROM에 저장하거나 별도의 회로를 사용해서 산출하기도 한다. $T(g)$는 $\frac{1}{1.g}$의 근사계산이므로 $T(g) = \frac{1}{1.g} + e_t$이다. e_t는 근사에 따른 오차이다. $T(g)$를 (식8)의 분자와 분모에 곱하면 $\frac{1}{F} = \frac{T(g)}{T(g) \times F}$이다. 이 식에서 분모를 정리하면 다음 (식10)이

된다.

$$T(g)\times F=(\frac{1}{1.g}+e_t)(1.g+h)=1+e_t(1.g+h)+\frac{h}{1.g} \quad \text{---(10)}$$

(식10)에서 $B=T(g)\times F\geq 0$이면 역수 알고리즘은 다음 (식11)이 된다.

$$\begin{aligned} &A=B-1 \\ &\frac{1}{F}=T(g)\times(1-A)(1+A^2)(1+A^4)(1+A^8)..... \\ &\quad =T(g)\times R\times(1+A^2)(1+A^4)(1+A^8)..... \end{aligned} \quad \text{---(11)}$$

여기서 $R=1+A$이다.

한편, $B=T(g)\times F<0$이면 역수 알고리즘은 다음 (식12)가 된다.

$$\begin{aligned} &A=1-B \\ &\frac{1}{F}=T(g)\times(1+A)(1+A^2)(1+A^4)(1+A^8)..... \\ &\quad =T(g)\times R\times(1+A^2)(1+A^4)(1+A^8)..... \end{aligned} \quad \text{---(12)}$$

여기서 $R=1+A$이다.

(식11)과 (식12)는 본 논문에서 제안하는 가변 시간 골드스미트 역수 알고리즘을 쉽게 구현할 수 있도록 기존 알고리즘 변형한 것이다. 이들 식에서 A는 언제나 양의 수이다.

3.2 오차 분석

골드스미트 역수 알고리즘은 곱셈을 반복해서 하므로 오차가 누적이 된다. 그러므로 구하고자 하는 정밀도보다 긴 자리수의 연산을 해야 한다. 반복 연산의 자릿수 p라 할 때 p를 구하려면 오차를 분석해야 한다.

(식12)에서 $1-B$의 뺄셈은 케리 전달이 필요하다. 하드웨어 처리 속도를 빠르게 하려고 본 논문에서는 근사계산인 $1-2^{-p}-B$를 연산한다. A^i의 최대 오차는 다음 (식13)으로 구할 수 있다. $[X]$는 X의 오차가 없는 값을 나타내며 연산중의 곱셈 결과를 소수점 이하 p비트에서 잘라내면 절삭오차는 $e_r = 2^{-p}$ 보다 작다.

$$
\begin{aligned}
A &= [A] - e_r \\
A^2 &= ([A] - e_r)^2 - e_r = [A^2] - 2e_r \times [A] + e_r^2 - e_r \\
&\fallingdotseq [A^2] - e_r \qquad \because [A] \ll 1.0 \qquad \text{---(13)} \\
A^4 &\fallingdotseq [A^4] - e_r \\
A^8 &\fallingdotseq [A^8] - e_r
\end{aligned}
$$

(식11)에서 $2-R$ 또한 $2-2^{-p}-R$로 근사계산하면 (식11)과 (식12)의 최대 오차는 다음 (식14)가 된다.

$$
\begin{aligned}
X_1 &= T(g) \times R = T(g) \times ([R] - er) - e_r \fallingdotseq [X1] - 2e_r \\
X_2 &= T(g) \times R \times (1 + A^2) \\
&= ([T(g) \times R] - 2e_r)([1 + A^2] - e_r) - e_r \\
&\fallingdotseq [X_2] - 4e_r \qquad \text{---(14)} \\
X_3 &= T(g) \times R \times (1 + A^2)(1 + A^4) = [X_3] - 6e_r \\
X_4 &= T(g) \times R \times (1 + A^2)(1 + A^4)(1 + A^8) = [X_4] - 8e_r
\end{aligned}
$$

3.3 연산 자릿수

IEEE 754 단정도 형식에서 가수부의 유효자릿수는 24비트이다. 유효자릿수에 라운드 한 비트를 더하면 25 비트이므로 최대 오차는 2^{-25}보다 작아야 한다. (식14)에서 $8e_r = 8 \times 2^{-p} < 2^{-25}$이 되어야 한다. 따라서 IEEE 754 단정도 형식의 실수 역수 계산에 필요한 연산 자릿수는 p=28이다.

IEEE 754 배정도 형식에서 가수부의 유효자릿수는 53 비트이다. 유효자릿수에 라운드 한 비트를 더하면 54 비트이므로 최대 오차는 2^{-54}보다 작아야 한다. (식14)에서 $8e_r = 8 \times 2^{-p} < 2^{-54}$가 되어야 한다. 따라서 IEEE 754 배정도 형식에서 실수의 역수 계산에 필요한 연산 자릿수는 p=57이다.

3.4 가변시간 골드스미트 역수 알고리즘

(식11)과 (식12)에서 $A^i < 2^{\frac{-p}{2}}$ 이면 $A^{2i} < 2^{-p}$이다. 그러므로 $A^i < 2^{\frac{-p}{2}}$ 가 되면 반복 연산을 종료한다. $A^i < 2^{\frac{-p}{2}}$ 의 판별은 A의 소수점 이하 $\frac{p}{2}$개의 비트가 모두 0인 것을 판별하는 것으로 이들 비트를 NOR하는 회로로 간단하게 구현할 수 있다. $\frac{p}{2}$개의 비트를 판별하므로 p는 반드시 짝수가 되어야 한다. 그러므로 배정도 형식에서 필요한 연산 자릿수는 p=58이 된다.

본 논문에서 제안하는 가변 시간 골드스미트 역수 알고리즘을 정리하면 다음 (표-1)과 같다.

표 1. 가변시간 골드스미트 역수 알고리즘

To compute, $\frac{1}{F}$ where F is (1.g+h).

$T(g)$ is pre-calculated approximate $\frac{1}{1.g}$.

p=28 if IEEE-754 single precision.

p=58 if IEEE-754 double precision.

1) $B = F \times T(g)$

2) if $B < 1$ then {

$A = 1 - 2^{-p} - B;$

$R = 1 + A;$

}

else {

$A = B - 1;$

$R = 2 - 2^{-p} - B;$

}

3) $R = R \times T(g);$

if $A < 2^{\frac{-p}{2}}$ then exit;

4) $A = A \times A;$

5) $R = R \times (1 + A);$

if $A \geq 2^{\frac{-p}{2}}$ then goto 4;

Ⅳ. 하드웨어구현 및 검증

(표 1)에서 보인 알고리즘대로 연산을 하는 하드웨어 계산기 블록도를 (그림 3)에 나타내었다. 또한 (표 2)에 상태 기계 흐름을 보인다.

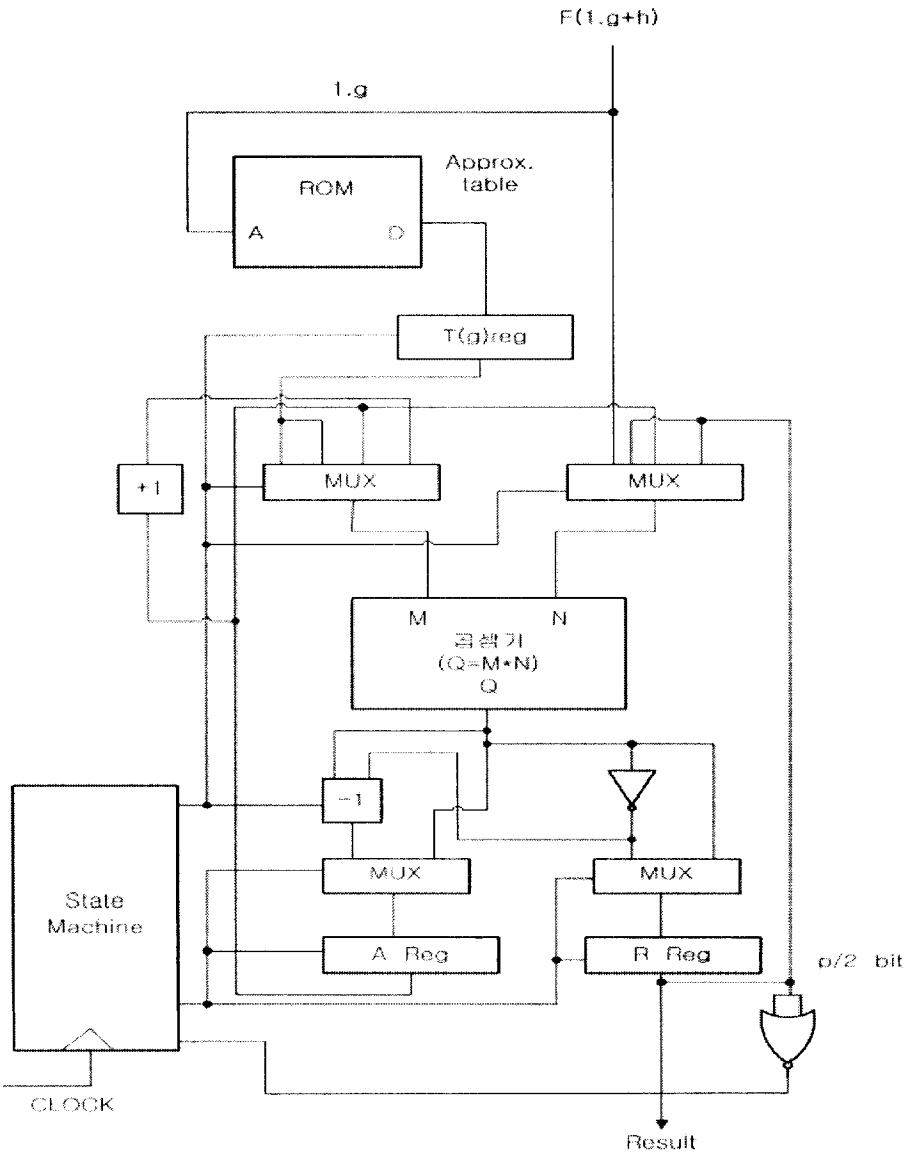


그림 3. 곱셈기 하나로 연산하는 골드스미트 나눗셈기 블록도

(그림 3) 블록도와 (표 2)의 상태 흐름도는 제안한 가변 시간 골드스미트 역수 알고리즘을 하드웨어로 구현한 것이다. (그림 3)은 롬 테이블 하나, 레지스터 셋, 먹스 넷, 곱셈기 하나, 1 덧셈기 하나, 1 뺄셈기 하나, 인버터, NOR게이트, 상태 머신으로 되어 있다.

그림의 맨 위쪽에 F값을 입력한다. F값의 상위 비트를 잘라서 1.g비트를 만든다. 1.g비트는 롬 테이블을 호출하는 데에 쓴다. 롬 테이블의 출력 값은 테이블 레지스터에 저장이 된다.

이 레지스터 값은 M 입력 쪽의 MUX가 첫 번째 단계에서 곱셈기의 입력 값으로 정하는데 N입력 쪽의 MUX가 곱셈기의 입력 값으로 정한 F와 함께 곱셈기의 입력 값으로 넣는다. 입력 쪽의 MUX는 4×1 MUX로 각 단계마다 입력 값을 차례로 하나씩 고르게 되어 있다.

곱셈기는 이렇게 입력한 값을 곱셈해서 A 레지스터와 R 레지스터에 저장하게 되는데 저장하기 전에 결과 값에 따라 다르게 변환해서 저장하게 된다.

변환 값은 곱셈기의 출력 부분에서 MUX가 다시 선택을 하게 한다. 출력 쪽의 MUX는 2×1 MUX로 곱셈결과 값에 따라서 MUX가 레지스터에 저장할 값을 선택하는 것이다.

유효비트 값 A는 A레지스터에 곱해주는 값 R은 R레지스터에 저장한다. 이 때의 R레지스터 값은 결과 값이 된다.

곱셈을 계속할지 그만 둘지 결정하는 것은 $\frac{p}{2}$값을 보고 판단하는데 이것은 그림의 오른쪽 아래쪽에 있는 NOR 게이트로 판단을 한다. $\frac{p}{2}$값이 모두 0이 되어서 더 이상 연산할 필요가 없어지면 NOR 게이트의 출력 값이 1로 되어서 상태 머신에게 알려주는데 상태 머신이 이 신호를 받고 연산을 종료한다. $\frac{p}{2}$값이 0이 되지 않으면 곱셈기의 M과 N에 다시 A 레지스터 값과 R 레지스터 값을 입력해서 연산을 해야 한다.

앞서 했던 연산과 마찬가지로 곱셈기의 출력 값은 경우에 따라서 변환 값을 MUX가 A 레지스터와 R 레지스터에 저장한다. 역시 이 때의 R 레지스터 값은 결과 값이 되고 NOR 게이트가 연산을 계속할지 그만 둘지 신호를 상태 머신에게 주게 되고 이 신호에 따라 연산을 계속하든지 그만 둔다.

이 과정은 곱셈기 하나를 이용한 알고리즘에서는 모두 다섯 단계를 거쳐서

처리를 한다. 이 처리과정 다섯 단계는 상태 머신이 관리한다. 모든 처리가 끝나면 그림의 아래쪽으로 결과 값이 나오게 된다.

단계별로 설명하면 상태-1에서 $\frac{1}{F}$의 근사 값을 테이블에서 읽어서 $T(g)$레지스터에 저장한다.

상태-2에서 (식10)처럼 $T(g)$레지스터를 분모 F에 곱해서 결과 B를 계산한다. 곱셈 결과 B를 '1'과 비교하여 (식11)과 (식12)처럼 A와 R을 생성하여 각각 A 레지스터와 R 레지스터에 저장한다.

여기서 A와 R값은 B<1일 때에 A=1-B이고 R=1+A이다. 또 B≥1일 때에 A=B-1이고 R=2-B이다.

상태-3에서 R 레지스터 값과 $T(g)$ 레지스터 값을 곱해서 R 레지스터에 저장한다. 동시에 A의 소수점이하 비트 $\frac{p}{2}$개가 모두 0인가를 판정해서 상태 기계에 알린다. 판정이 맞으면 더 이상 계산할 필요가 없으므로 역수 계산을 마친다. $\frac{p}{2}$가 0이 아닐 때에는 상태-4로 넘어간다.

상태-4에서 A 레지스터 값의 자승을 계산한 A^{2i} 값을 A 레지스터에 저장한다.

골드스미트 역수 알고리즘은 한번에 곱셈을 두 번 해야 하기 때문에 상태-4의 곱셈과정이 끝나면 상태-5로 넘어간다.

상태-4와 상태-5가 필요한 곱셈 두 번을 나누어서 하는 것이다.

상태-5에서는 R 레지스터와 A^i에 1을 더한 것을 곱해서 그 결과를 R 레지스터에 저장한다. 그리고 A^i의 소수점이하 비트 $\frac{p}{2}$개 모두 0인가를 판정해서 상태 기계에 알린다. 판정이 맞으면 더 이상 계산할 필요가 없으므로 역수 계산을 마친다. 그렇지 않으면 상태-4로 가서 반복 연산을 한다.

종래 골드스미트 역수 계산기에 입력이 $\frac{p}{2}$개인 NOR 게이트를 추가하고 상태 흐름도를 일부 변경하는 것으로 가변 시간 골드스미트 역수 계산기를 구현할 수 있다.

골드스미트 역수 알고리즘은 두 번해야 하는 곱셈연산이 서로 독립하므로 곱셈기 두 개를 병렬로 사용할 수 있다. 곱셈기 두 개를 병렬로 사용하는 가변

시간 골드스미트 역수 계산기의 상태 흐름도는 (표 2)에서 상태-4와 상태-5를 하나로 묶어서 구현할 수 있다.

(표 3)에 곱셈기 두 개를 병렬로 사용하는 가변시간 골드스미트 역수 계산기의 상태 흐름도를 보인다. 그리고 블록도를 (그림 4)에 나타내었다.

표 2. 곱셈기 하나를 사용하는 가변 시간 골드스미트 역수 계산기 상태 흐름도

/* Result = $\frac{1}{F}$ */

(State-1)
Get approximate reciprocal table $T(g)$;

(State-2)
Multiplier in port M=$T(g)$;
Multiplier in port N=F;
Multiplier out port Q=B;
R=One's complement of B;
if B<1 then A=R-1;
else A=B-1;

(State-3)
Multiplier in port M=$T(g)$.
Multiplier in port N=R.
Multiplier out port Q=R.
if msb $\frac{p}{2}$ bit of A is all '0' then result is R;

(State-4)
Multiplier in port M=A;
Multiplier in port N=A;
Multiplier out port Q=A;

(State-5)
Multiplier in port M=1+A.
Multiplier in port N=R.
Multiplier out port Q=R.
if msb $\frac{p}{2}$ bit of A is all '0' then result is R.
else goto State-4.

표 3. 곱셈기 두 개를 사용하는 가변 시간 골드스미트 역수 계산기 상태 흐름도

```
/*  Result = 1/F  */
(State-1)
    Get approximate reciprocal table T(g);

(State-2)
    Multiplier 1 in port M=T(g);
    Multiplier 1 in port N=F;
    Multiplier 1 out port Q=B;
    if B<1 then {
                  R=One's complement of B;
                  A=R-1;
                }
          else  {
                  A=B-1;
                  R=One's complement of B;
                }

(State-3)
     Multiplier 2 in port M=T(g);
     Multiplier 2 in port N=R;
     Multiplier 2 out port Q=R;
     if msb p/2 bit of A is all '0' then result is R;

(State-4)
     Multiplier 1 in port M=A;
     Multiplier 1 in port N=A;
     Multiplier 1 out port Q=A;

     Multiplier 2 in port M=1+A;
     Multiplier 2 out port Q=R;
     if msb p/2 bit of A is all '0' then result is R;
                                  else goto State-4;
```

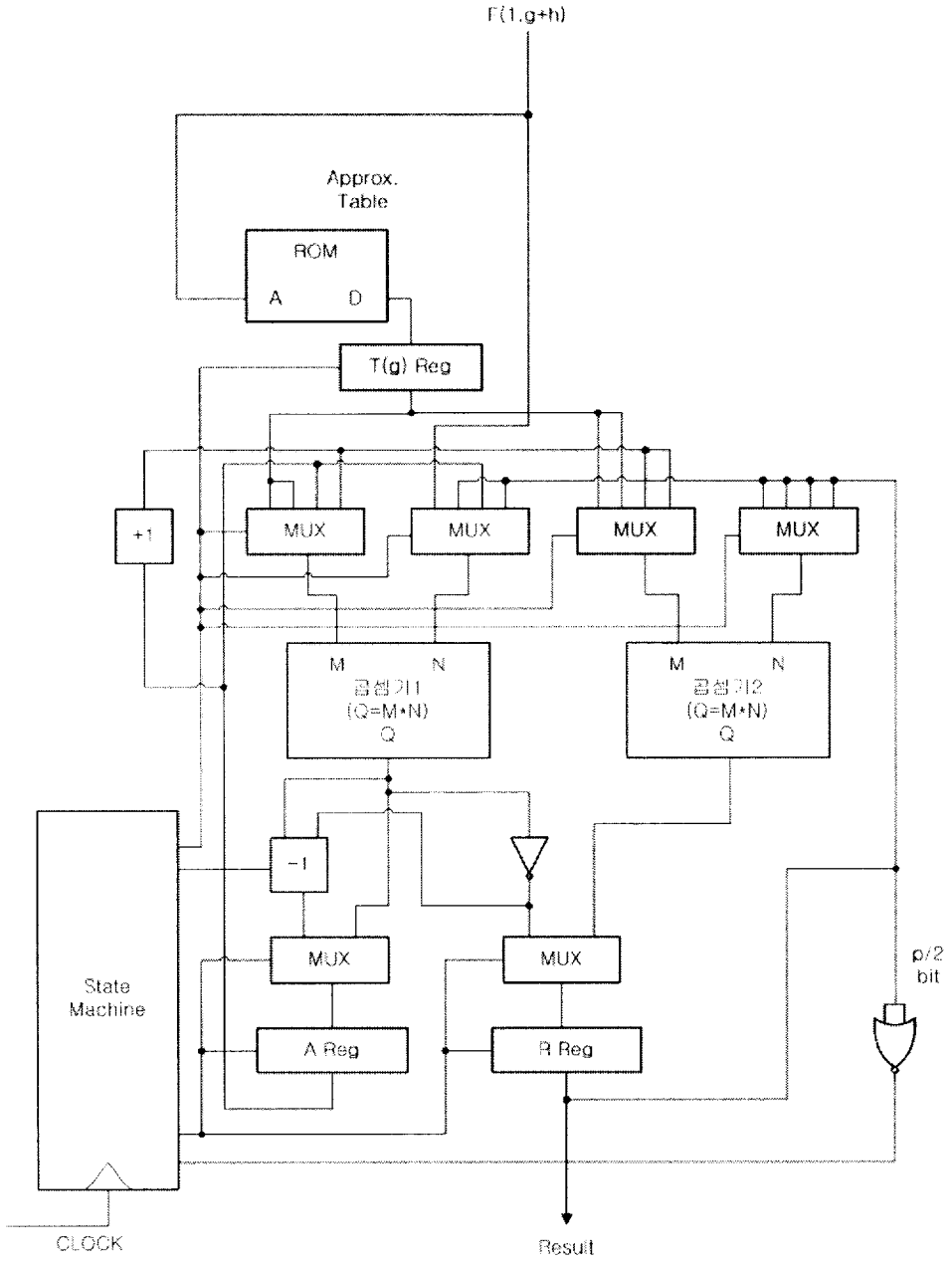


그림 4. 곱셈기 두 개로 연산하는 골드스미트 나눗셈기 블록도

설계한 가변 시간 골드스미트 역수 계산기는 Verilog HDL로 기술하고 시뮬레이션해서 동작을 확인하였다. 동작을 확인한 내용은 부록에 추가하였다.

V. 연구 결과 및 분석

(식10)에서 A는 다음 (식15)가 된다.

$$A = e_t(1.g + h) + \frac{h}{1.g} \qquad \text{---(15)}$$

(식15)에서 A는 h=0에서 '$1.g \times e_t$'가 되며 $0 \le h \le h_{\max}$에서 단조 증가함수이다. 한편 '$A = T(g) \times (1.g + h)$'이므로 h는 다음 (식16)이 된다.

$$h = \frac{1 + A}{T(g)} - 1.g \qquad \text{---(16)}$$

'$e_t > 0$' 인 경우에 h에 따른 A의 변화를 (그림 5)에 보인다.

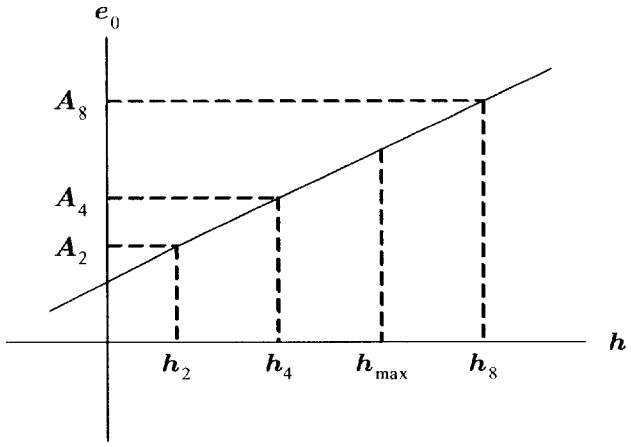


그림 5. $e_t > 0$인 경우에 h와 A의 그래프

(그림 5)에서 '$A_2 = 2^{\frac{-p}{2}}$'를 나타내며 h_2는 A_2인 h의 값으로 (식16)에서 구

한 것이다. 또한 '$A_4 = \sqrt{A_2}$'를 나타내며 h_4는 A_4인 h의 값이다. h_8은 '$A_8 = \sqrt{A_4}$'인 h값이다.

(그림 5)에서 $0 \le h \le h_2$구간에는 '$T(g) \times R$' 연산으로 역수를 구할 수 있으므로 곱셈을 두 번 수행한다.

$h_2 < h \le h_4$구간에서는 '$T(g) \times R \times (1 + A^2)$'연산을 수행하므로 역수 계산에 곱셈 네 번이 필요하다.

또한 $h_4 < h \le h_{\max}$구간에서는 '$T(g) \times R \times (1 + A^2)(1 + A^4)$'연산을 수행하므로 곱셈이 여섯 번 필요하다. e_t가 음수인 경우에도 비슷한 방법으로 h에 따른 곱셈 횟수를 산출할 수 있다.

DasSarma의 연구 결과 최적의 근사 역수는 다음 (식17)로 구할 수 있다[13].

$$T(g) = \frac{1}{1.g} \fallingdotseq RN\left(\frac{1}{\sqrt{1.g + 2^{-n_g - 1}}}\right). \quad \text{---(17)}$$

여기서, RN은 round to nearest이다.

$T(g)$의 비트 길이를 t라고 하면 근사 역수 테이블의 크기는 '$2^{n_g} \times t$' 비트가 된다. 테이블의 길이는 2^{n_g}이며 폭은 t비트이다.

본 논문에서 제안한 알고리즘에 따라 IEEE 754 표준 단정도 형식의 실수의 역수 계산에 필요한 곱셈 횟수를 (표 4)에 보이고 배정도 형식의 실수의 역수 계산에 필요한 곱셈 횟수를 (표 5)에 보인다.

표 4. IEEE 754 단정도 실수 역수 계산에 필요한 곱셈 횟수

Table size	Average No. of Multiply[1]	Average No. of Multiply[2]	No. of multiply[1]			
			2	4	6	8
$2^4 \times 4$	5.66	3.83	0.14%	16.8%	83.1%	0.0%
$2^5 \times 5$	5.32	3.66	0.27%	33.3%	66.5%	0.0%
$2^6 \times 6$	4.71	3.35	0.54%	63.5%	36.0%	0.0%
$2^7 \times 7$	4.00	3.00	1.08%	97.9%	1.03%	0.0%
$2^8 \times 8$	3.96	2.98	2.16%	97.8%	0.0%	0.0%
$2^6 \times 7$	4.19	3.10	0.94%	88.6%	10.5%	0.0%
$2^6 \times 8$	3.99	3.00	1.17%	98.2%	0.6%	0.0%
$2^6 \times 9$	3.98	2.99	1.17%	98.0%	0.05%	0.0%

여기서 Multiply[1]은 곱셈기를 하나 사용한 경우이고 Multiply[2]는 곱셈기를 두 개 사용한 경우이다.

표 5. IEEE 754 배정도 실수 역수 계산에 필요한 곱셈 횟수

Table size	Average No.of multiply[1]	Average No. of Multiply[2]	No. of multiply[1]			
			2	4	6	8
$2^7 \times 7$	6.11	4.06	0.0%	0.77%	92.9%	6.3%
$2^8 \times 8$	5.97	3.98	0.0%	1.53%	98.5%	0.0%
$2^9 \times 9$	5.94	3.97	0.0%	3.1%	96.9%	0.0%
$2^{10} \times 10$	5.88	3.94	0.0%	6.1%	93.9%	0.0%
$2^7 \times 8$	5.97	3.99	0.0%	1.3%	98.7%	0.0%
$2^6 \times 6$	6.90	4.45	0.0%	0.38%	54.3%	45.3%
$2^6 \times 7$	6.37	4.18	0.0%	0.67%	80.3%	19.1%
$2^6 \times 8$	6.09	4.04	0.0%	0.83%	97.0%	5.2%
$2^6 \times 9$	6.02	4.01	0.0%	0.83%	97.2%	2.02%

여기서 Multiply[1]은 곱셈기 한 개를 사용한 경우이고 Multiply[2]는 곱셈기를 두 개 사용한 경우이다.

종래 골드스미트 알고리즘에서는 최대 오차를 고려해서 반복 횟수를 정했다. 즉, 종래 알고리즘에 따른 단정도 실수 역수는 '$2^7 \times 7$' 테이블을 사용하면 곱셈을 6회 하였고 '$2^8 \times 7$' 테이블을 사용하면 곱셈을 4회 하였음을 표-4에서 알 수 있다. 그러나 본 논문에서 제안한 가변 시간 골드스미트 역수 알고리즘에서는 '$2^7 \times 7$' 테이블에서 곱셈을 평균 4.00회, '$2^8 \times 7$' 테이블에서 곱셈을 평균 3.96 회 하여서 역수를 계산할 수 있다. 또한 '$2^6 \times 8$' 테이블에서 곱셈을 평균 3.99회 하여서 역수를 계산할 수 있다. 따라서 곱셈을 평균 4회 해서 역수를 계산하려면 종래 알고리즘에서는 '$2^8 \times 8$' 테이블을 사용했지만 본 논문에서 제안하는 알고리즘을 사용하면 '$2^6 \times 8$' 테이블을 사용해도 곱셈을 4회 해서 역수를 계산할 수 있다. '$2^8 \times 8$' 테이블의 면적은 '$2^8 \times 8$' 테이블의 사분의 일에 지나지 않는다.

이러한 결과는 배정도 실수 연산에서도 동일하게 나타난다. 배정도 실수 역수 계산은 표-5에서 알 수 있는데 종래 알고리즘에서는 '$2^7 \times 7$' 테이블을 사용하면 곱셈을 8회, '$2^8 \times 8$' 테이블을 사용하면 곱셈을 6회 하였다. 그러나 본 논문에서 제안한 가변 시간 골드스미트 역수 알고리즘에서는 '$2^7 \times 7$' 테이블을 사용하면 평균 곱셈을 6.11회, '$2^8 \times 8$' 테이블과 '$2^7 \times 8$' 테이블을 사용하면 곱셈을 평균 5.97회 수행한다. '$2^7 \times 8$' 테이블의 크기는 '$2^8 \times 8$' 테이블의 반이다.

따라서 본 논문에서 제안한 알고리즘을 사용하면 종래 알고리즘에서 사용하던 역수 근사 테이블의 크기를 반이하로 줄일 수 있음을 알 수 있다.

역수 근사 테이블의 길이와 폭의 관계를 표-4와 표-5에서 알 수 있다. 표-4의 단정도 실수 나눗셈에서 '$2^6 \times 6$', '$2^6 \times 7$', '$2^6 \times 8$', '$2^6 \times 9$' 테이블은 길이가 '2^6'로 같으며 폭은 6비트에서 9비트까지로 다르다. 종래 알고리즘에서는 이들 모든 테이블에서 곱셈을 6회 하여서 역수를 계산하였지만 본 논문에서 제안하

는 알고리즘에서는 곱셈을 각각 4.71회, 4.19회, 3.99회, 3.98회 하여서 역수를 계산한다. 이들 결과에서 테이블의 폭을 1~2비트 늘리면 성능이 크게 개선이 되지만 3비트 이상을 늘리면 성능이 개선이 되지 않음을 알 수 있다.

테이블의 길이와 폭의 관계는 배정도실수에서도 동일하게 나타난다. 표-5에서 '$2^6 \times 6$', '$2^6 \times 7$', '$2^6 \times 8$', '$2^6 \times 9$' 테이블은 길이가 '2^6'로 같지만 폭은 6비트에서 9비트까지 다르다. 종래 알고리즘에서는 이들 모든 테이블에서 곱셈을 8회 하여서 역수를 계산하였지만 본 논문에서 제안하는 알고리즘에서는 곱셈을 각각 6.90회, 6.37회, 6.09회, 6.02회 하여서 나눗셈을 계산한다.

이들 결과에서 본 논문의 알고리즘에서는 역수 근사 테이블의 폭 t는 길이 L의 '$log_2 L$'보다 1~2 비트 큰 것이 가격대비 성능 면에서 좋음을 알 수 있다.

종래의 골드스미드 역수 알고리즘에서는 테이블 크기를 정하는 데에 제약이 많았다. 단정도 실수에서 곱셈을 6회 하여서 역수 계산을 하려면 '$2^4 \times 4$' 테이블을 사용했고 4회 하려면 '$2^8 \times 8$' 테이블을 사용하였다. 크기가 중간인 테이블은 의미가 없었다. 즉, '$2^5 \times 5$' 테이블과 '$2^6 \times 6$' 테이블과 '$2^7 \times 7$' 테이블을 사용하면 역수 계산에 곱셈이 6회만 필요하였다.

그러나 본 논문에서 제안한 알고리즘은 역수 계산기 성능에 따라 다양한 테이블을 선택할 수 있는 장점이 있다. 이것은 SOC처럼 크기가 제한이 된 실리콘에 CPU, 메모리, 입출력장치를 모두 집적하는 응용 분야에서 최적의 성능을 실현할 수 있는 방법을 제공해 준다.

Ⅵ. 결론

부동소수점 나눗셈은 뺄셈을 반복하는 SRT 알고리즘과 곱셈을 반복하는 뉴톤-랍손(Newton-Raphson) 알고리즘 및 골드스미트(Goldschmidt) 알고리즘이 있다. 골드스미트 알고리즘은 나누는 수와 나뉘는 수에 곱셈을 되풀이하여 나눗셈을 수행한다. 이것은 나누는 수의 역수를 나뉘는 수에 곱하는 방식과 같은 방식이다.

본 논문에서는 골드스미트 부동소수점 역수 알고리즘을 변형하여 오차가 정해진 값보다 작아지는 시점까지만 반복 연산을 하는 가변 시간 골드스미트 역수 알고리즘을 제안했다. 이를 구현하려고 골드스미트 역수 알고리즘을 변형하고 오차를 분석하여 연산 자릿수와 반복 연산을 종표할 오차 한계를 계산하였고 제안한 알고리즘을 구현하는 회로와 상태 기계를 구성하였다. 또한 근사 테이블을 구성하고 역수 계산에 소요가 되는 평균 곱셈 횟수를 계산하여 그 결과를 기존 골드스미트 역수 알고리즘과 비교 분석하였다.

종래 골드스미트 역수 알고리즘에 따른 단정도 실수 역수 계산은 '$2^7 \times 7$' 테이블을 사용하면 곱셈을 6회 하였고 '$2^8 \times 8$' 테이블을 사용하면 곱셈을 4회 하였다. 그러나 본 논문에서 제안한 가변 시간 골드스미트 역수 알고리즘에서는 '$2^7 \times 7$' 테이블에서 곱셈을 평균 4.00회 하였고 '$2^8 \times 8$' 테이블과 '$2^6 \times 8$' 테이블에서 곱셈을 평균 3.96회 하여서 역수 계산을 할 수 있었다. 본 논문에서 제안한 알고리즘을 사용하면 '$2^6 \times 8$' 테이블을 사용해도 곱셈을 평균 4회 하여서 역수를 계산할 수 있었다. '$2^6 \times 8$' 테이블의 크기는 '$2^8 \times 8$' 테이블 크기의 4분의 1에 지나지 않는다. 따라서 본 논문에서 제안한 알고리즘은 근사 역수 테이블의 크기를 줄일 수 있다.

본 논문에서는 근사 역수 테이블의 길이와 폭의 관계를 밝혔다. '$2^6 \times 6$', '$2^6 \times 7$', '$2^6 \times 8$', '$2^6 \times 9$' 테이블은 길이가 '2^6'로 같지만 폭은 6 비트에서 9 비트까지 다르다. 기존 알고리즘에서는 이들 모든 테이블에서 곱셈을 8회 하여서 역수를 계산했지만 본 논문에서 제안하는 알고리즘에서는 각각 곱셈 횟수가 6.90회, 6.37회, 6.09회, 6.02회로 나눗셈을 계산하였다.

이들 결과에서 본 논문의 알고리즘에서는 역수 근사 테이블의 폭 t는 길이 L

의 '$log_2 L$'보다 1~2 비트 큰 것이 가격대비 성능 면에서 좋음을 알 수 있었다.

본 논문에서 제안한 가변 시간 골드스미트 역수 알고리즘은 평균 곱셈 횟수가 중요한 디지털 신호처리 , 컴퓨터 그래픽스 , 멀티미디어 처리 , 과학 기술 연산에서 폭 넓게 사용할 수 있다. 또한 역수 계산기 성능에 따른 최적의 근사 역수 테이블을 구성할 수 있으므로 하드웨어 사양에 제한이 있는 SOC(System On Chip)에 유용하게 적용할 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] IEEE, *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Standard, Std. 754-1985

[11] Israel Koren , *Computer Arithmetic Algorithms/2ND EDITION* , Prentice-Hall , pp.53-84 and pp.181-221 , 2001

[12] 정재원 , "내장형 프로세서를 위한 IEEE-754 고성능 부동소수점 나눗

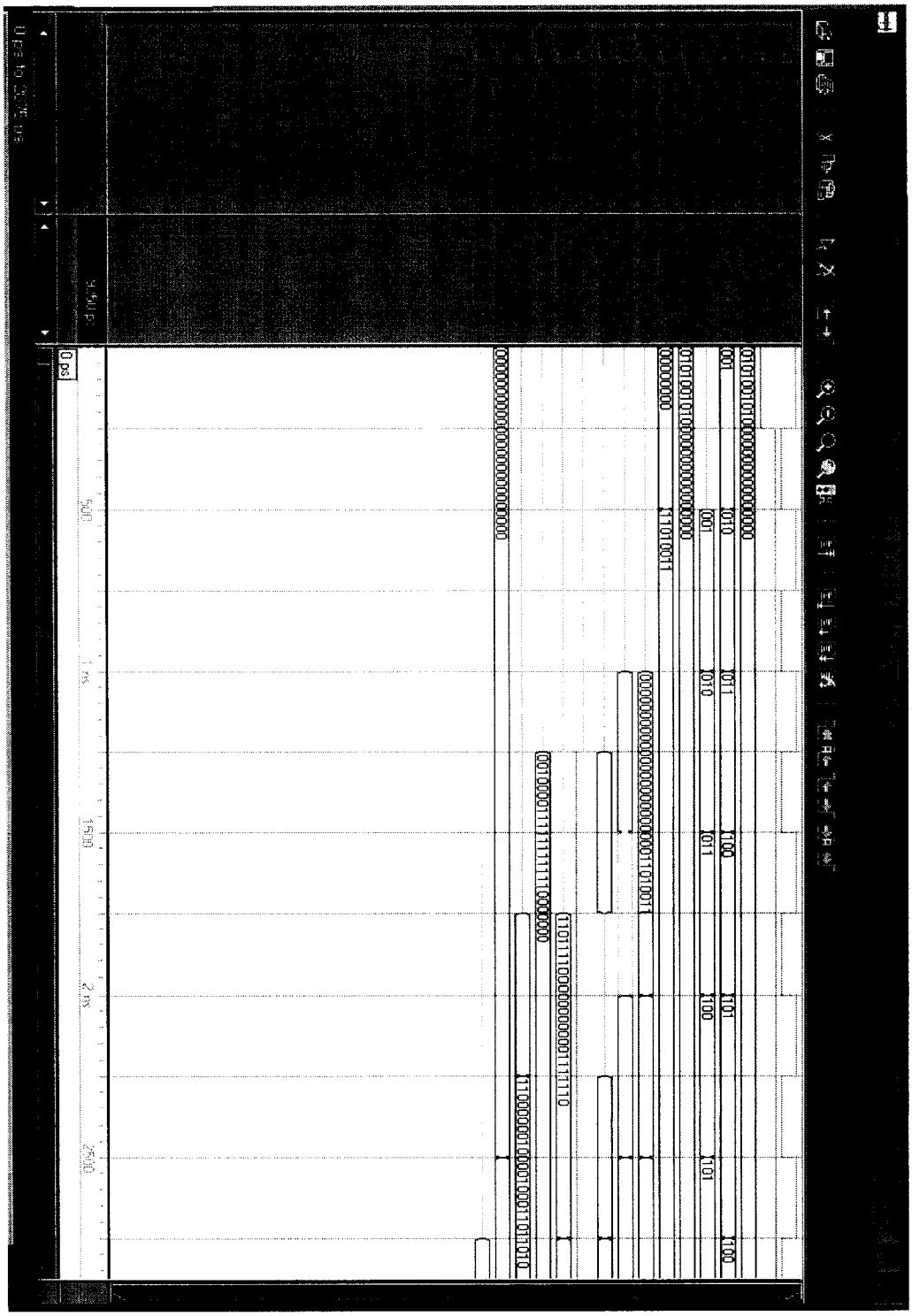
셈기의 설계" , 전자공학회논문지 , pp.1-49 , 2002
[13] D. DasSarma and D. Matula, "Measuring and Accuracy of ROM Reciprocal Tables," IEEE Transactions on Computer, Vol.43, No. 8, pp. 932-930, Aug. 1994

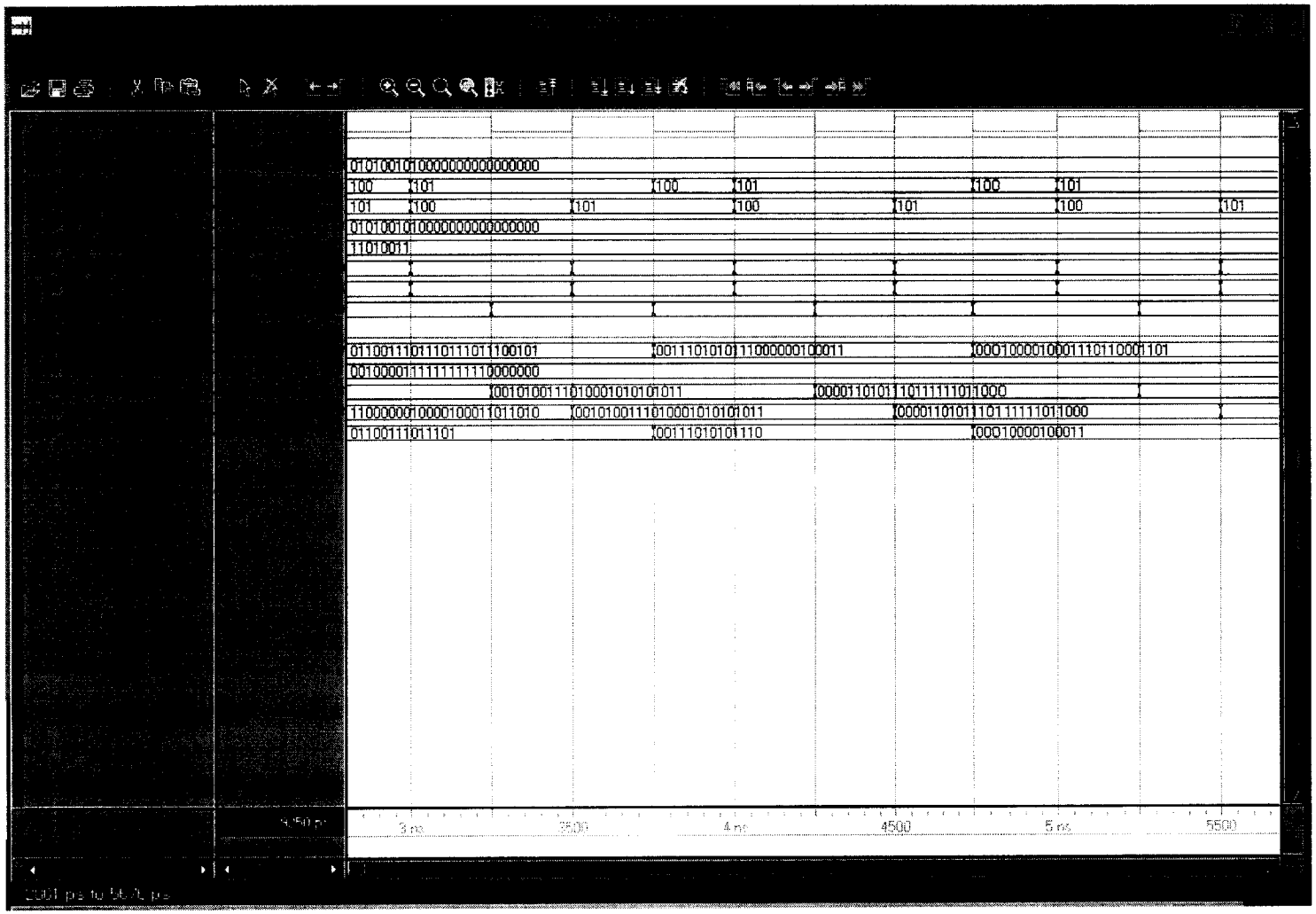
부 록

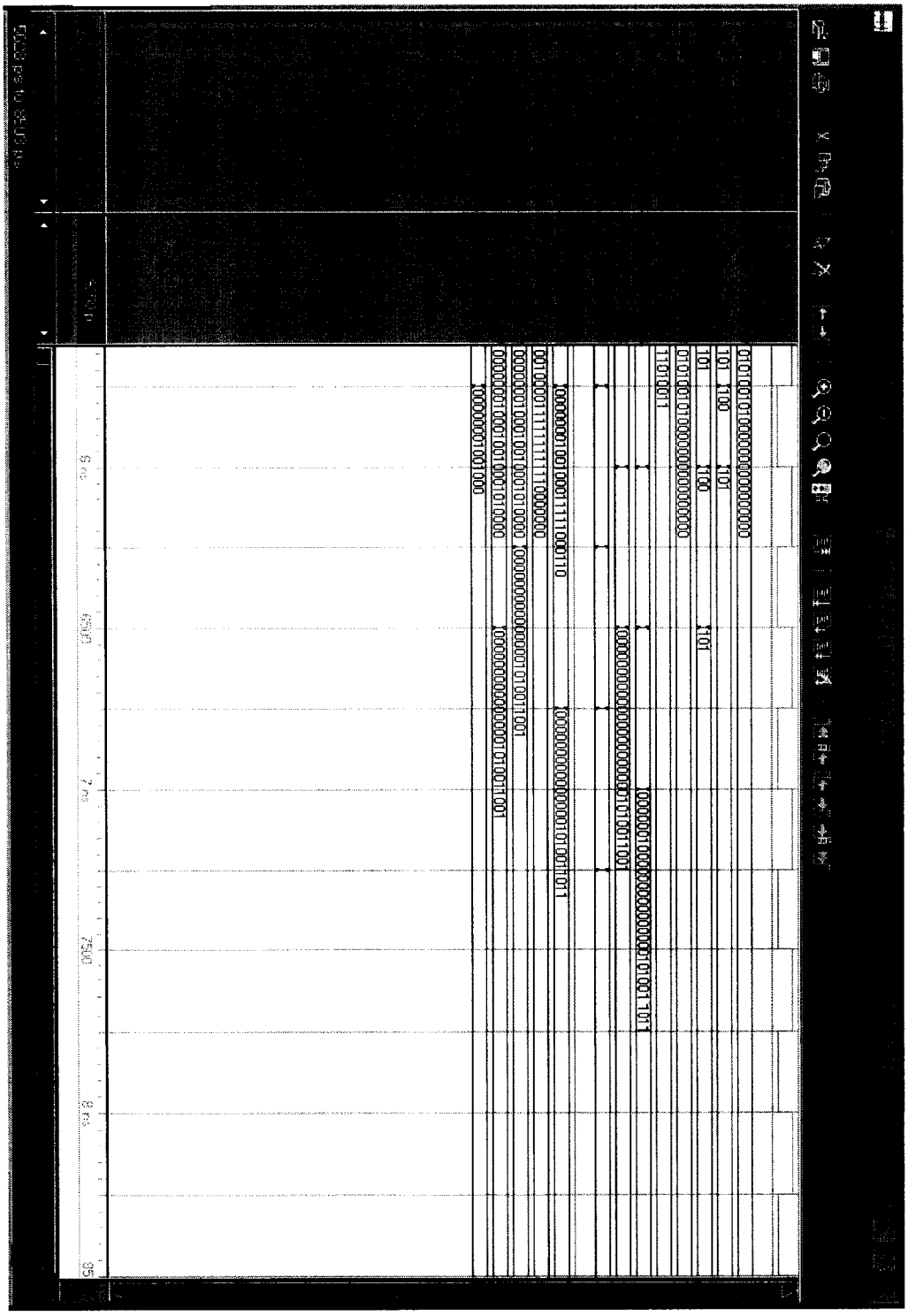
1. 나눗셈기의 시뮬레이션 동작 결과

첫 번째 줄은 클럭 주파수이다. 위에서 두 번째 줄은 reset 신호이고 위에서 다섯 번째 줄은 상태이고 맨 아랫줄이 $\frac{p}{2}$ 값이다. 그리고 맨 아래에서 두 번째 줄이 결과 값이다.

그림에서 맨 아랫줄 $\frac{p}{2}$값이 0이 되면서 상태가 멈추고 결과 값에 변화가 없어 연산이 끝나는 것을 볼 수 있다.




3 ms
3500
4 ms
4500
5 ms
5500



2. Verilog HDL 소스코드

```
/*
#################################################################
#################################################################
#####       가변시간 골드슈미트 부동소수점 역수알고리즘 나눗셈기      ######
#################################################################
#################################################################
*/

/*
=========================
===      역수 롬테이블       ===
===      256 * 8           ===
=========================
*/
module ROM_table (in_value, out_value);
    input [7:0] in_value;
    output [7:0] out_value;
    reg[7:0] ROM [0:255];
    wire [7:0] out_value;
    wire [7:0] in_value;

    initial
    begin
        ROM[0]=8'h0;ROM[1]=8'hff;ROM[2]=8'hfe;
          ROM[3]=8'hfd;ROM[4]=8'hfc;ROM[5]=8'hfb;
          ROM[6]=8'hfa;ROM[7]=8'hf9;ROM[8]=8'hf8;
          ROM[9]=8'hf7;ROM[10]=8'hf6;ROM[11]=8'hf5;
          ROM[12]=8'hf4;ROM[13]=8'hf3;ROM[14]=8'hf2;
          ROM[15]=8'hf1;ROM[16]=8'hf0;ROM[17]=8'hf0;
          ROM[18]=8'hef;ROM[19]=8'hee;ROM[20]=8'hed;
          ROM[21]=8'hec;ROM[22]=8'heb;ROM[23]=8'hea;
          ROM[24]=8'he9;ROM[25]=8'he9;ROM[26]=8'he8;
```

```
ROM[27]=8'he7;ROM[28]=8'he6;ROM[29]=8'he6;
ROM[30]=8'he5;ROM[31]=8'he4;ROM[32]=8'he3;
ROM[33]=8'he2;ROM[34]=8'he2;ROM[35]=8'he1;
ROM[36]=8'he0;ROM[37]=8'hdf;ROM[38]=8'hdf;
ROM[39]=8'hde;ROM[40]=8'hdd;ROM[41]=8'hdc;
ROM[42]=8'hdc;ROM[43]=8'hdb;ROM[44]=8'hda;
ROM[45]=8'hd9;ROM[46]=8'hd9;ROM[47]=8'hd8;
ROM[48]=8'hd7;ROM[49]=8'hd7;ROM[50]=8'hd6;
ROM[51]=8'hd5;ROM[52]=8'hd4;ROM[53]=8'hd4;
ROM[54]=8'hd3;ROM[55]=8'hd2;ROM[56]=8'hd2;
ROM[57]=8'hd1;ROM[58]=8'hd0;ROM[59]=8'hd0;
ROM[60]=8'hcf;ROM[61]=8'hce;ROM[62]=8'hce;
ROM[63]=8'hcb;ROM[64]=8'hcc;ROM[65]=8'hcc;
ROM[66]=8'hcb;ROM[67]=8'hcb;ROM[68]=8'hca;
ROM[69]=8'hc9;ROM[70]=8'hc9;ROM[71]=8'hc8;
ROM[72]=8'hc8;ROM[73]=8'hc7;ROM[74]=8'hc6;
ROM[75]=8'hc6;ROM[76]=8'hc5;ROM[77]=8'hc5;
ROM[78]=8'hc4;ROM[79]=8'hc3;ROM[80]=8'hc3;
ROM[81]=8'hc2;ROM[82]=8'hc2;ROM[83]=8'hc1;
ROM[84]=8'hc0;ROM[85]=8'hc0;ROM[86]=8'hbf;
ROM[87]=8'hbf;ROM[88]=8'hbe;ROM[89]=8'hbe;
ROM[90]=8'hbd;ROM[91]=8'hbd;ROM[92]=8'hbc;
ROM[93]=8'hbc;ROM[94]=8'hbb;ROM[95]=8'hba;
ROM[96]=8'hba;ROM[97]=8'hb9;ROM[98]=8'hb9;
ROM[99]=8'hb8;ROM[100]=8'hb8;ROM[101]=8'hb7;
ROM[102]=8'hb7;ROM[103]=8'hb6;ROM[104]=8'hb6;
ROM[105]=8'hb5;ROM[106]=8'hb5;ROM[107]=8'hb4;
ROM[108]=8'hb4;ROM[109]=8'hb3;ROM[110]=8'hb3;
ROM[111]=8'hb2;ROM[112]=8'hb2;ROM[113]=8'hb1;
ROM[114]=8'hb1;ROM[115]=8'hb0;ROM[116]=8'hb0;
ROM[117]=8'haf;ROM[118]=8'haf;ROM[119]=8'haf;
ROM[120]=8'hae;ROM[121]=8'hae;ROM[122]=8'had;
ROM[123]=8'had;ROM[124]=8'hac;ROM[125]=8'hac;
ROM[126]=8'hab;ROM[127]=8'hab;ROM[128]=8'haa;
ROM[129]=8'haa;ROM[130]=8'haa;ROM[131]=8'ha9;
```

```
ROM[132]=8'ha9;ROM[133]=8'ha8;ROM[134]=8'ha8;
ROM[135]=8'ha7;ROM[136]=8'ha7;ROM[137]=8'ha7;
ROM[138]=8'ha6;ROM[139]=8'ha6;ROM[140]=8'ha5;
ROM[141]=8'ha5;ROM[142]=8'ha4;ROM[143]=8'ha4;
ROM[144]=8'ha4;ROM[145]=8'ha3;ROM[146]=8'ha3;
ROM[147]=8'ha2;ROM[148]=8'ha2;ROM[149]=8'ha2;
ROM[150]=8'ha1;ROM[151]=8'ha1;ROM[152]=8'ha0;
ROM[153]=8'ha0;ROM[154]=8'ha0;ROM[155]=8'h9f;
ROM[156]=8'h9f;ROM[157]=8'h9f;ROM[158]=8'hfe;
ROM[159]=8'h9e;ROM[160]=8'h9d;ROM[161]=8'h9d;
ROM[162]=8'h9d;ROM[163]=8'h9c;ROM[164]=8'h9c;
ROM[165]=8'h9b;ROM[166]=8'h9b;ROM[167]=8'h9b;
ROM[168]=8'h9a;ROM[169]=8'h9a;ROM[170]=8'h9a;
ROM[171]=8'h99;ROM[172]=8'h99;ROM[173]=8'h99;
ROM[174]=8'h98;ROM[175]=8'h98;ROM[176]=8'h98;
ROM[177]=8'h97;ROM[178]=8'h97;ROM[179]=8'h96;
ROM[180]=8'h96;ROM[181]=8'h96;ROM[182]=8'h95;
ROM[183]=8'h95;ROM[184]=8'h95;ROM[185]=8'h94;
ROM[186]=8'h94;ROM[187]=8'h94;ROM[188]=8'h93;
ROM[189]=8'h93;ROM[190]=8'h93;ROM[191]=8'h92;
ROM[192]=8'h92;ROM[193]=8'h92;ROM[194]=8'h91;
ROM[195]=8'h91;ROM[196]=8'h91;ROM[197]=8'h91;
ROM[198]=8'h90;ROM[199]=8'h90;ROM[200]=8'h90;
ROM[201]=8'h8f;ROM[202]=8'h8f;ROM[203]=8'h8f;
ROM[204]=8'h8e;ROM[205]=8'h8e;ROM[206]=8'h8e;
ROM[207]=8'h8d;ROM[208]=8'h8d;ROM[209]=8'h8d;
ROM[210]=8'h8c;ROM[211]=8'h8c;ROM[212]=8'h8c;
ROM[213]=8'h8c;ROM[214]=8'h8b;ROM[215]=8'h8b;
ROM[216]=8'h8b;ROM[217]=8'h8a;ROM[218]=8'h8a;
ROM[219]=8'h8a;ROM[220]=8'h8a;ROM[221]=8'h89;
ROM[222]=8'h89;ROM[223]=8'h89;ROM[224]=8'h88;
ROM[225]=8'h88;ROM[226]=8'h88;ROM[227]=8'h88;
ROM[228]=8'h87;ROM[229]=8'h87;ROM[230]=8'h87;
ROM[231]=8'h86;ROM[232]=8'h86;ROM[233]=8'h86;
ROM[234]=8'h86;ROM[235]=8'h85;ROM[236]=8'h85;
```

```
        ROM[237]=8'h85;ROM[238]=8'h85;ROM[239]=8'h84;
        ROM[240]=8'h84;ROM[241]=8'h84;ROM[242]=8'h83;
        ROM[243]=8'h83;ROM[244]=8'h83;ROM[245]=8'h83;
        ROM[246]=8'h82;ROM[247]=8'h82;ROM[248]=8'h82;
        ROM[249]=8'h82;ROM[250]=8'h81;ROM[251]=8'h81;
        ROM[252]=8'h81;ROM[253]=8'h81;ROM[254]=8'h80;
        ROM[255]=8'h80 ;

  end

  assign out_value=ROM[in_value];
endmodule//end of ROM_table(in_value, out_value)

/*
==============
=== 나눗셈   ===
==============
*/
module division
(   clock, reset, f_number, state, state_process, f_register,   table_register,
    m_input, n_input, multiplier_result, B_1, A, B, R,
    division_result, A_precision
);
   input clock, reset;
   input [24:0] f_number;
   output [2:0] state, state_process;
   output [24:0] f_register;
   output [7:0] table_register;
   output [31:0] m_input, n_input;
   output [63:0] multiplier_result;
   output [24:0] A, B, R;
   output B_1;
   output [24:0] division_result;
   output [13:0] A_precision;
```

```
reg [24:0] division_result;
reg [13:0] A_precision;

reg [24:0] table_number;
reg [7:0] table_temp;
reg [2:0] state,state_process;
parameter [2:0] state_signal1=3'b001,
                state_signal2=3'b010,
                state_signal3=3'b011,
                state_signal4=3'b100,
                state_signal5=3'b101,
                state_signal6=3'b110 ;
parameter [2:0] state_sequency1=3'b001,
                state_sequency2=3'b010,
                state_sequency3=3'b011,
                state_sequency4=3'b100,
                state_sequency5=3'b101,
                state_sequency6=3'b110 ;
parameter a_p_length=14;
parameter [2:0] mux_signal1=3'b001,
                mux_signal2=3'b010,
                mux_signal3=3'b011,
                mux_signal4=3'b100,
                mux_signal5=3'b101;
reg [2:0] case_signal;
reg [7:0] table_register;
wire [7:0] table_register_temp;
reg [63:0] multiplier_result;
reg [31:0] m_input, n_input;
reg [24:0] f_register;
reg [24:0] A, B, R;
reg [25:0] br_temp;
reg [1:0] out_mux;
parameter in_mux_signal=1'b0;
parameter [1:0] out_mux_signal_stop=2'b00,
```

```
                out_mux_signal_ar=2'b01,
                out_mux_signal_a=2'b10,
                out_mux_signal_r=2'b11 ;
reg B_1;

//=== reset signal ===
always @ (posedge clock or negedge reset)
begin
   if(!reset)
   begin
        table_register=8'b0;
        table_number=f_number;
        table_temp=table_number[25:18];
        state=state_signal1;
        f_register=25'b0;
        f_register=f_number;
        division_result=25'h0;
   end
   else
   begin
      state_process=state;
   end
end

ROM_table table_value(table_temp, table_register_temp);

//=== process ===
always @ (state_process)
begin
   if(reset)
   begin
        case(state_process)
            state_signal1:
                    begin
                        table_register=table_register_temp;
```

```
                state=state_signal2;
            end
        state_signal2:
            begin
                m_input=32'b0;
                n_input=32'b0;
                case_signal=mux_signal2;
                state=state_signal3;
            end
        state_signal3:
            begin
                m_input=32'b0;
                n_input=32'b0;
                case_signal=mux_signal3;
                state=state_signal4;
            end
        state_signal4:
            begin
                m_input=32'b0;
                n_input=32'b0;
                case_signal=mux_signal4;
                state=state_signal5;
            end
        state_signal5:
            begin
               case_signal=mux_signal5;
               division_result=R;
            end
    endcase
  end//end of if(reset)
end//end of always @ (state_process)

//=== multiplier ===
always @ (negedge clock)
begin
```

```
    if(reset)
    begin
        multiplier_result=m_input*n_input;
    end
end

//=== m_mux and n_mux ===
always @(case_signal or posedge clock)
begin
  case(case_signal)
      mux_signal1: m_input=31'b0;
      mux_signal2:
              begin
                  m_input=table_register;
                  n_input=f_register;
              end
      mux_signal3:
              begin
                  m_input=table_register;
                  n_input=R;
              end
      mux_signal4:
              begin
                   m_input=A;
                   n_input=A;
              end
      mux_signal5:
              begin
                   m_input={1'b1,A};
                   n_input=R;
              end
  endcase
end//end of always @(case_signal)

//=== process run ===
```

```
always @ (multiplier_result)
begin
   if(reset)
   begin
       case(case_signal)
           mux_signal2:
                begin
                   B=multiplier_result[33:9];
                   B_1=multiplier_result[34];
                   out_mux=out_mux_signal_ar;
                end
           mux_signal3:
                begin
                     out_mux=out_mux_signal_r;
                end
           mux_signal4:
                begin
                      out_mux=out_mux_signal_a;
                end
           mux_signal5:
                begin
                      out_mux=out_mux_signal_r;
                      if( A[24:11]!=14'b0)
                      begin
                         state=state_signal4;
                         A_precision=A[24:11];
                      end
                      else
                      begin
                         state=state_signal5;
                      end
                end
       endcase
   end
end
```

```
//=== a_mux and r_mux===
always @(negedge clock)
begin
  case(out_mux)
       out_mux_signal_ar:
            begin
                R=~B;
                if(B_1!=1'b1)
                begin
                    br_temp=R;
                end
                else
                begin
                    br_temp={B_1,B};
                end
                A=br_temp[24:0];

            end
       out_mux_signal_a:
                begin
                    A=multiplier_result[49:25];
                end
       out_mux_signal_r:
                begin
                   R=multiplier_result[49:25];
                end
   endcase

  end//end of always @(out_mux)

endmodule//end of division(clock, reset, f_number, division_result)
```

감사의 글

희망찬 마음으로 좀더 배우고자 대학원에 온지가 엊그제 같은데 어느덧 대학원 입학한지 벌써 2년이 지나버렸습니다. 제가 대학원에 와서 큰 행운이 있었다면 그것은 제 지도 교수님인 조경연 교수님을 만났다는 것입니다. 늘 하드웨어에 관심이 많던 제가 하드웨어를 알려고 해도 알기가 쉽지 않았습니다. 하드웨어란 무엇일까. 늘 그런 궁금증에 빠져 있었습니다. 그런데 그 궁금증을 풀 수 있도록 도와 주신분이 지금 지도교수님입니다. 하지만 늘 이해력이 모자라서 헤매기만 했는데 그래도 교수님께서는 그런 저를 붙잡아 주셨습니다. 대학생 시절 수업시간에 교수님께서 이런 질문을 하셨습니다. "시스템이란 무엇입니까" 저는 이렇게 대답했습니다. "시스템이란 어느 범위 안에 있는 개체가 이루어가는 질서입니다." 교수님께서는 모자란 것이 있다며 목적성이 빠졌다는 것이었습니다. 그래서 다시 교수님과 뜻을 맞추기를 "시스템이란 목적성이 있는 범위 안에 있는 개체들이 이루어 가는 질서"라고 하였습니다. 그 뒤로 저는 시스템의 실체를 하드웨어로 보았습니다. 소프트웨어는 반드시 하드웨어가 있어야지만 돌아가는 것이기에 하드웨어가 시스템의 바탕으로 생각하였습니다. 그 하드웨어를 알기까지 대학원에 입학한 뒤부터 2년 동안 많이 헤매었습니다. 그러는 동안 일부이지만 하드웨어가 무엇인지 깨닫게 되었습니다. 그동안 깨달음을 얻는데 도와주신 조경연 교수님과 논문심사를 맡아주신 서경룡 교수님 , 김종남 교수님과 그 외의 부경대학교 컴퓨터공학과 교수님들과 대학생시절 많은 깨달음을 주신 밀양대학교 컴퓨터공학과 교수님들과 여러 모로 공부하고 논문작성에 도와주신 같은 연구실의 박창수 선배님 그리고 Verilog HDL소스코드를 작성하는데 도와준 박근철씨에게 고마운 마음을 전하고 싶습니다. 그리고 저를 뒤에서 든든하게 밀어주신 부모님께도 고마운 마음을 전하고 싶습니다.

2005년 1월

한 경 현 올림