곱셈기를 사용한 배정도 정수 나눗셈기

송홍복* · 조경연**

Double Precision Integer Divider Using Multiplier

Hong-Bok Song* · Gyeong-Yeon Cho**

요 약

본 논문에서는 'w bit × w bit = $2w$ bit' 곱셈기를 사용하여 $2w$ 비트 정수 N과 w 비트 정수 D의 $\frac{N}{D}$ 나눗셈을 수행하는 알고리즘을 제안한다. 본 연구에서 제안하는 알고리즘은 제수 D가 '$D=0.d\times 2^L$, $0.5<0.d<1.0$'일 때, '$0.d\times 1.g=1+e$, $e<2^{-w}$'가 되는 '$\frac{1}{D}$'의 근사 값 '$1.g\times 2^{-L}$'을 가칭 상역수로 정의하고, 피제수 N을 '$w-3$' 비트보다 작은 워드로 분할하고, 각 분할된 워드에 상역수를 곱해서 부분 몫을 계산하고, 부분 몫을 합산하여 배정도 정수 나눗셈의 몫을 구한다. 제안한 알고리즘은 정확한 몫을 산출하기 때문에 추가적인 보정이 요구되지 않는다. 본 논문에서 제안하는 알고리즘은 곱셈기만을 사용하므로 마이크로프로세서를 구현할 때 나눗셈을 위한 추가적인 하드웨어가 요구되지 않는다. 그리고 기존 알고리즘인 SRT 방식에 비해 동작속도가 빠르다. 따라서 본 논문의 연구 결과는 마이크로프로세서 및 하드웨어 크기에 제한적인 SOC(System on Chip) 구현 등에 폭넓게 사용될 수 있다.

ABSTRACT

This paper suggested an algorithm that uses a multiplier, 'w bit × w bit = $2w$ bit', to process $\frac{N}{D}$ integer division of $2w$ bit integer N and w bit integer D. An algorithm suggested of the research, when the divisor D is '$D=0.d\times 2^L$, $0.5<0.d<1.0$', approximate value of $\frac{1}{D}$, '$1.g\times 2^{-L}$', which satisfies '$0.d\times 1.g=1+e$, $e<2^{-w}$', is defined as over reciprocal number and the dividend N is segmented in small word more than '$w-3$' bit, and partial quotient is calculated by multiplying over reciprocal number in each segmented word, and quotient of double precision integer division is evaluated with sum of partial quotient. The algorithm suggested in this paper doesn't require additional correction, because it can calculate correct reciprocal number. In addition, this algorithm uses only multiplier, so additional hardware for division is not required to implement microprocessor. Also, it shows faster speed than the conventional SRT algorithm. In conclusion, results from this study could be used widely for implementation SOC(System on Chip) and etc. which has been restricted to microprocessor and size of the hardware.

키워드

나눗셈, 배정도 정수 나눗셈, 역수 알고리즘

Key word

integer divider, reciprocal algorithm

* 동의대학교 전자공학과 교수

** 부경대학교 IT융합응용공학과 교수

접수일자 : 2009. 10. 16

심사완료일자 : 2010. 01. 15

Ⅰ. 서 론

최근 대부분의 산업 분야에서는 컴퓨터를 주로 사용하고 있다. 그 중에서도 특히 멀티미디어 분야에 많이 보급되어 있는데, 이 분야에 관련된 많은 기능들이 하드웨어로 구현되기를 원하는 요구도 함께 증대되고 있는 실정이다. 이런 기능들 중 한 가지가 정수 곱셈기이다. 1980-90년대만 해도 곱셈기를 하드웨어로 구현하려면 많은 비용이 들어서 곱셈기를 컴파일러에서 시프트 연산을 사용하여 소프트웨어로 구현하였다. 그러나 근래에는 반도체 공정기술이 발달하면서 반도체 집적 기술도 함께 발전하였다. 그 결과 대부분의 32 비트 이상 마이크로프로세서는 정수 곱셈기를 하드웨어로 구현하여 내장하고 있으며, 32 비트 곱하기 32 비트를 수행하여 64 비트 결과를 얻는 곱셈 연산을 한 클럭에 수행할 수 있게 되었다.

정수 나눗셈 연산은 멀티미디어 처리를 비롯한 여러 분야에서 사용하는 빈도는 높지 않지만 정수 곱셈 연산 못지않게 시스템의 성능에 영향을 미친다. 실제 표 계산(Spread Sheet) 프로그램에서 정수 나눗셈 연산이 나타나는 빈도는 10% 정도로 조사되고 있다[1]. 정수 나눗셈 연산은 컴파일러로 구현할 경우 연산 속도가 매우 느려서 시스템 전체의 성능을 저하시키는 문제가 있다. 이 문제점을 해결하기 위해서는 정수 나눗셈도 하드웨어로 구현해야 하는데, 이 때 사용하는 방식이 뺄셈을 반복해서 나눗셈을 수행하는 SRT[2, 3, 4] 방식이다. 그런데 이 방법은 추가적인 하드웨어가 소요되고 또한 SRT 방식의 연산 속도가 느려서 파이프라인 구조가 복잡해지는 단점을 가진다. 그리고 이 방식을 사용한 64 비트 정수 나눗셈 연산의 경우에는 속도가 더욱 느려진다.

최근에는 컴파일러에 작은 정수에 대한 역수를 미리 계산해 놓고 나눗셈을 수행하는 방식을 채택하는 경우가 있다. 이 방식은 실제 제수가 작은 정수인 정수 나눗셈이 나타나는 빈도가 높으므로 상당히 효율적인 방법이다[5].

정수 나눗셈은 정수 곱셈기를 사용해서 수행할 수도 있다. 이 방법은 곱셈을 반복하여 제수의 역수를 구하고, 이 역수를 피제수에 곱하는 방법을 이용해서 나눗셈을 수행하는 것이다. 곱셈을 반복해서 나눗셈을 수행하는 알고리즘은 뉴톤-랍손(Newton-Raphson) 알고리즘과 골드스미트(Goldschmidt) 알고리즘이 있다[6, 7, 8]. 뉴톤-랍손 알고리즘은 제수의 역수를 구해서 피제수에 곱하여 나눗셈을 수행하고, 골드스미트 알고리즘은 제수와 피제수에 반복적인 곱하기로 나눗셈을 수행한다. 이 방법은 근사 값으로 계산을 하기 때문에 결과를 보정하여 정확한 값을 계산하는 추가적인 과정이 필요하다.

본 논문에서는 제수의 역수를 구하고, 이를 피제수에 곱해서 배정도 정수 나눗셈을 수행하는 알고리즘을 제안한다. 본 논문에서 제안하는 알고리즘에서 제수의 역수는 송홍복 등[9]이 제안한 상역수를 구해서 사용한다. 그리고 피제수를 제수의 워드 길이보다 3 비트 이상 작은 워드로 분할하고 각 분할된 워드에 제수의 상역수를 곱해서 부분 몫을 계산하고, 부분 몫을 합산하여 배정도 정수 나눗셈의 몫을 구한다.

본 논문에서 제안하는 알고리즘은 정수 곱셈기만을 사용하므로 추가적인 하드웨어가 필요하지 않고, 기존의 나눗셈 알고리즘인 SRT 나눗셈 방식과 비교했을 때 연산속도가 빠르다는 장점을 가진다. 제안하는 정수 나눗셈 알고리즘은 C 언어를 사용하여 모델링하고, 동작을 확인한다.

본 논문의 구성은 다음과 같다. 2장에서는 근사 역수를 구하고 곱셈을 반복하여 나눗셈을 수행하는 알고리즘인 뉴톤-랍손 또는 골드스미트 알고리즘을 이용하여 상역수를 계산한다. 그리고 3장은 피제수를 분할하고, 분할된 피제수에 상역수를 곱하여 배정도 나눗셈을 수행하는 알고리즘을 제안한다. 4장에서는 제안한 알고리즘을 C 언어를 사용해서 모델링하여 동작을 확인하고, 기존의 SRT 알고리즘과 성능을 비교한다. 마지막으로 5장에서는 결론을 맺는다.

Ⅱ. 근사 역수 계산에 의한 상역수

1. 뉴톤-랍손 역수 알고리즘

뉴톤-랍손 역수 알고리즘은 역수의 근사 값을 초기 값으로 하고, 반복 연산으로 오차를 줄여 나간다. 오차는 반복할 때마다 자승으로 줄어든다. 그리고 1회의 반복 연산에 2회의 곱셈이 필요하다.

임의의 정수 D의 역수를 구하기 위하여 함수 '$f(X) = D - \frac{1}{X}$'을 정의한다. 뉴톤-랍손 알고리즘에

서 X_i를 X의 근사 값이라고 하면 X_{i+1}은 식 (1)과 같이 된다.

$$X_{i+1} = X_i - \frac{f(X_i)}{f'(X_i)} = X_i(2 - DX_i) \tag{1}$$

본 논문에서는 정수 곱셈기를 사용하므로 정수 D를 좌정렬 형식으로 표현하면 '$0.d \times 2^L$, $D < 2^L$' 이 된다. 여기에서 $0.d$를 가수부, 2^L을 지수부라고 표현한다. 예를 들어 32 비트 마이크로프로세서에서 정수 7을 위의 형식으로 표현하면 가수부는 0xE0000000, 지수부는 2^8 이다.

정수 D가 $D = 2^L$일 때 역수를 구하는 것은 자명함으로 본 논문에서는 $D = 2^L$인 경우는 제외한다. 따라서 정수의 가수부는 '$0 < 0.d < 1.0$'이 된다. 여기서 가수부 $0.d$는 식 (2)와 같이 두 부분으로 나눌 수 있다.

$$0.d = 0.u + v \tag{2}$$

식 (2)에서 u와 v의 길이를 각각 n_u, n_v 비트로 정의한다. 여기서 v는 '$0 \leq v < 2^{-n_u}$'이다. 식 (1)의 수렴 속도를 빠르게 하기 위해서 $\frac{1}{0.u}$을 근사 계산해서 테이블 $T(u)$를 미리 작성해 놓는다. 이 근사 테이블은 ROM에 미리 저장해 두거나 또는 별도의 회로를 사용해서 산출하여 사용하기도 한다. 근사 테이블 $T(u)$는 $\frac{1}{0.u}$의 근사 계산이므로 '$T(u) = \frac{1}{0.u} + e_u$'로 표현할 수 있다. 이 식에서 e_u는 근사에 따른 오차이다. 근사 테이블 $T(u)$를 X의 초기 근사값 X_0로 정의한다.

나눗셈 결과의 정확도를 위해서는 최적의 근사 역수를 구해야 한다. DasSarma의 연구 결과에 의하면 최적의 근사 역수는 식 (3)과 같이 주어진다[10].

$$T(u) = \frac{1}{0.u} \fallingdotseq RN\left(\frac{1}{0.u + 2^{-n_u - 1}}\right) \tag{3}$$

where RN is round to nearest

식 (3)에서 $T(u)$의 소수점 이하 길이를 t비트라고 하면 '$T(u) = (1.b_1b_2b_3 \cdots b_t)_2$, $1.0 < T(u) < 2.0$'이다.

따라서 근사 역수 테이블의 크기는 '$2^{n_u} \times t$'비트 이다. 한편 하드웨어를 간단하게 하기 위해서 근사 테이블을 만들지 않을 경우에는 역수의 초기값으로 '$X_0 = 0.d$ XOR $- 1$'을 사용한다.

식 (1)에서 '$2 - DX_i$' 뺄셈은 하드웨어 구현 시에 캐리 전달 지연 시간이 필요하다. 이러한 문제점을 해결하기 위하여 본 논문에서는 '$2 - 2^{-w} - DX_i$'을 계산한다. 이 식에서 w는 워드 길이이다. 본 논문에서 사용하는 뉴톤-랍손 역수 알고리즘은 식 (4)와 같다.

$$\begin{aligned} &For\ \ i = \{0, 1, 2, \cdots, n-1\} \\ &\quad X_{i+1} = X_i \times (2 - 2^{-w} - DX_i) \end{aligned} \tag{4}$$

2. 골드스미트 나눗셈 알고리즘

골드스미트 나눗셈 알고리즘은 제수와 피제수에 반복적으로 동일한 값을 곱하여 제수가 '1.0'에 수렴하면 피제수가 나눗셈의 결과가 되는 것이다. 이 때, 피제수를 1로 설정하면 제수의 역수를 구할 수 있다. 이 알고리즘은 1회의 연산에 서로 독립적인 2회의 곱셈이 필요하다. 따라서 2개의 곱셈기를 사용하면 연산시간을 줄일 수 있어 IBM RS/6000, AMD K7 프로세서 등에서 사용되고 있다[11].

$\frac{X_0}{D_0}$를 계산하는 골드스미트 나눗셈 알고리즘은 식 (5)와 같다.

$$\begin{aligned} &For\ \ i \in \{0, 1, 2, \cdots, n-1\} \\ &\quad R_i = 2 - D_i \\ &\quad X_{i+1} = X_i \times R_i \\ &\quad D_{i+1} = D_i \times R_i \end{aligned} \tag{5}$$

반복 연산을 수행하면 D_n은 1.0에 수렴하므로 다음 식이 성립한다.

$$\frac{X_n}{D_n} = \frac{X_0 R_0 R_1 R_2 \cdots R_n}{D_0 R_0 R_1 R_2 \cdots R_n} = X_n$$

따라서 X_n이 나눗셈 결과이다. X_0가 1.0이면 식 (5)는 D_0의 역수 값이 된다.

식(5)의 수렴 속도를 빠르게 하기 위하여 $\frac{1}{D}$의 근사 값을 미리 계산하여 근사 테이블 $T(u)$를 미리 작성해 놓으면 '$X=\frac{1}{D}$'는 식 (6)으로 구할 수 있다.

$$\begin{aligned} X &= \frac{1}{D} = \frac{T(u)}{T(u)\times D} = \frac{X_0}{D_0} \\ For\ & i \in \{0,1,2,\cdots,n-1\} \\ & R_i = 2 - 2^{-w} - D_i \\ & X_{i+1} = X_i \times R_i \\ & D_{i+1} = D_i \times R_i \end{aligned} \quad (6)$$

3. 상역수

뉴톤-랍손 또는 골드스미트 알고리즘을 사용하여 $\frac{1}{0.d}$의 근사 값 '$X_n = \frac{1}{0.d}+e_n$'을 계산한다. 여기서 e_n은 계산 오차로 '$|e_n| < 2^{-w/2-1}$'이 될 때까지 식 (4) 또는 식 (6)을 반복 연산한다. 이때 $\frac{1}{0.d}$의 범위는 '$1.0 < \frac{1}{0.d} < 2.0$'이 되는데, 계산 오차 e_n에 의해서 근사 역수의 범위가 '$X_n < 1.0$'이 되는 경우가 발생한다. Matthew Frank[12]와 Robert Alverson[13]의 알고리즘에서는 근사 역수의 범위가 위와 같이 되었을 때 나눗셈의 결과가 틀린 값을 가지게 된다. 본 논문에서는 보다 정확한 역수를 구하기 위해서 상역수[9]를 구해서 사용한다. w 비트 길이 정수 D의 상역수를 다음과 같이 정의한다.

정의-1)[9] 정수 D를 '$0.d\times 2^L$, $D<2^L$'이라 하면 '$0.d\times 1.g = 1+e$, $e<2^{-w}$'이 되는 $1.g\times 2^{-L}$을 D의 상역수라고 정의한다.

본 논문에서 모든 수는 좌정렬 형식을 사용하므로 근사 역수 X_n의 최상위 비트가 '0'이면 근사 역수의 범위가 '$X_n < 1.0$'이다. 이 경우에는 근사 역수 X_n의 1의 보수를 취해서 $\frac{1}{0.d}$의 근사 값으로 한다. $\frac{1}{0.d}$의 근사 값 X_n은 식 (7)과 같이 표현된다.

$$X_n = \frac{1}{0.d}+e_n = 1+g,\ 0<g<1.0 \quad (7)$$

'$w+1$'비트의 유효 자리수를 가지는 $\frac{1}{0.d}$의 근사 값을 구하기 위하여 X_n에 $0.d$를 곱하여 정리하면 식 (8)이 된다.

$$\begin{aligned} X_n \times 0.d &= \frac{X_n}{X_n-e_n} = 1+\frac{e_n^2+e_nX_n}{X_n^2-e_n^2} \\ &= 1+\alpha \end{aligned} \quad (8)$$

한편 어떤 값 β가 있어 '$(X_n-\beta)\times 0.d = 1$'이 된다면, 식 (9)가 성립해야 된다.

$$\begin{aligned} (X_n-\beta)\times 0.d &= \frac{X_n-\beta}{X_n-e_n} \\ &= 1+\frac{e_n^2+e_nX_n-\beta(X_n+e_n)}{X_n^2-e_n^2} \\ &= 1 \end{aligned} \quad (9)$$

식(8)과 식 (9)로부터 β를 구하면 식 (10)이 된다.

$$\begin{aligned} \alpha &= \frac{\beta(X_n+e_n)}{X_n^2-e_n^2} \\ \beta &= \alpha\times\frac{X_n^2-e_n^2}{X_n+e_n} = \alpha\times(X_n-e_n) \fallingdotseq \alpha X_n \end{aligned} \quad (10)$$

따라서 X_{n+1}은 식 (11)과 같이 된다.

$$\begin{aligned} X_{n+1} &= X_n - \alpha X_n \\ &= \frac{1}{F}+e_n - X_n\times\frac{e_n^2+e_nX_n}{X_n^2-e_n^2} \\ &\fallingdotseq \frac{1}{F}-\frac{e_n^2}{X_n} \end{aligned} \quad (11)$$

식(11)로부터 X_{n+1}의 오차는 X_n오차의 자승에 비례하는 것을 알 수 있다.

식(8)에서 '$X_n \times 0.d \geq 1.0$'이면 X_{n+1}은 식 (12)와 같이 된다.

$$X_n \times 0.d = 1 + \alpha$$
$$(X_n + \beta) \times 0.d = 1 + \alpha + \beta \times 0.d = 1$$
$$\beta = -\frac{\alpha}{0.d} \fallingdotseq -\alpha X_n = -(\alpha + g\alpha)$$

$$X_{n+1} = X_n + \beta = X_n - (\alpha + g\alpha) \quad (12)$$

한편 식 (8)에서 '$X_n \times 0.d < 1.0$'이면 X_{n+1}은 식 (13)과 같이 된다.

$$X_n \times 0.d = 1 - \alpha = t$$
$$(X_n + \beta) \times 0.d = t + \beta \times 0.d = 1$$
$$\beta = -\frac{t-1}{0.d}$$
$$\fallingdotseq -X_n \times (t-1) = -(t - g - gt - 1)$$

$$X_{n+1} = X_n + \beta \fallingdotseq 2g - t - tg \quad (13)$$

가능한 오차가 적은 정밀한 계산을 수행하기 위하여 식 (12) 또는 식 (13)을 계산할 때 X_n을 왼쪽으로 한 비트 시프트 시키면 레지스터에 저장된 값은 식 (7)의 g만을 가진다. 식 (11)로부터 X_{n+1}의 오차는 '$|e_{n+1}| < 2^{-w-2}$'가 되어야 하지만 실제에 있어서는 워드 길이에 따른 절삭 오차가 포함되어서 '$|e_{n+1}| < 2^{-w+1}$'이다. 다시 표현하면 식(14)가 된다.

$$X_{n+1} = \frac{1}{0.d} + e_{n+1}, \quad |e_{n+1}| < 2^{-w+1} \quad (14)$$

식 (14)의 양변에 $0.d$를 곱해서 정리하여 X_{n+2}를 구하면 식 (15)가 된다.

$$X_{n+1} \times 0.d = 1 + e_{n+1} \times 0.d$$

$$X_{n+2} = X_{n+1} - e_{n+1} \times 0.d \quad (15)$$

식 (15)의 양변에 $0.d$를 곱하여 정리하면 식 (16)이 된다.

$$X_{n+2} \times 0.d = 1 + e_{n+1} \times 0.d \times (1 - 0.d) \quad (16)$$

식 (16)에서 '$0 < (1 - 0.d) < 0.5$'이므로 X_{n+2}의 오차는 '$|e_{n+2}| < 2^{-w}$'이 된다. 따라서 식 (16)의 X_{n+2}가 $0.d$의 상역수 $1.g$이다.

III. 배정도 정수 나눗셈

정수 나눗셈은 식 (17)로 표현할 수 있다.

$$Q = \mathrm{INT}\left(\frac{N}{D}\right), \quad N = Q \times D + R \quad (17)$$

워드 길이가 w 비트일 때, 식 (17)에서 피제수 N이 $2w$ 비트, 제수 D가 w 비트인 정수 나눗셈이면 몫 Q는 $2w$ 비트, 나머지 R은 w 비트 길이이다. 여기에서 피제수 '$N = n_{2w-1}n_{2w-2} \cdots n_0$'를 식 (18)과 같이 두 부분으로 나누어 표현할 수 있다.

$$N = (n_{2w-1}n_{2w-2} \cdots n_{2w-m}) \times 2^{2w-m}$$
$$+ (n_{2w-m-1}n_{2w-m-2} \cdots n_0)$$
$$= N_a \times 2^{2w-m} + (n_{2w-m-1}n_{2w-m-2} \cdots n_0) \quad (18)$$

제수 D는 '$D = 0.d \times 2^{-L}$, $0.5 < 0.d < 1.0$'로 표현하고, '$0.d \times 1.g = 1 + e$, $e < 2^{-w}$'가 되는 $0.d$의 상역수 '$1.g$'를 2장에서와 같이 계산한다. 이것으로부터 식 (19)가 성립한다.

$$Q_a = \mathrm{INT}\left(\frac{2N_a}{0.d}\right) \quad (19)$$

식 (19)에서

$2N_a = Q_a \times 0.d + R_a$, $R_a = ra_{-1}ra_{-2} \cdots ra_{-w}$ 이다. 이 식에서 N_a를 식 (18)에 대입하면 다음과 같이 된다.

$$N = \frac{Q_a \times 0.d + R_a}{2} \times 2^{2w-m} + (n_{2w-m-1}n_{2w-m-2} \cdots n_0)$$

양 변을 $0.d$로 나누고 정리하면 다음과 같이 된다.

$$\begin{aligned}\frac{N}{0.d} &= Q_a \times 2^{2w-m-1} + \frac{R_a \times 2^{2w-m-1} + (n_{2w-m-1} \cdots n_0)}{0.d} \\ &= Q_a \times 2^{2w-m-1} + \frac{N_b \times 2^{2w-2m-1} + (na_{2w-2m-1} \cdots na_0)}{0.d}\end{aligned}$$

여기서

$$\begin{aligned}N_b &= na_{2w-m} \cdots na_{2w-2m} \\ &= (ra_{-1} \cdots ra_{-w}) + (n_{2w-m-1} \cdots n_{2w-2m}) \\ na_{2w-2m-1} &\cdots na_0 = n_{2w-2m-1} \cdots n_0\end{aligned}$$

이다.

식 (19)에서 $\frac{N_a}{0.d}$는 식 (20)과 같이 된다.

$$\begin{aligned}\frac{N_a}{0.d} &\doteqdot N_a \times 1.g = \frac{N_a}{0.d} + \frac{N_a \times e}{0.d} \\ &= Q.q + \frac{N_a \times e}{0.d} = M.m\end{aligned} \quad (20)$$

여기서

$$\begin{aligned}0 \le 0.q + \frac{N_a \times e}{0.d} < 2 \\ Q \le N_a \times 1.g < Q+2 \\ Q \le M < Q+1\end{aligned}$$

이다.

식 (20)에서 '$M \times 0.d = T.t$'를 계산하면 다음과 같은 세 가지 경우가 발생한다.

① '$T = N$'이고 '$t = 0$'인 경우
$M = Q, q = 0$이고, '$Q_a = Q$, $R_a = t$'이다.

② '$T = N$'이고 '$t \ne 0$'인 경우
$M = Q+1$, $q \ne 0$이다. 즉, '$(Q+1) \times 0.d = (Q.q + 0.q') \times 0.d = N + 0.q' \times 0.d$, $0.q + 0.q' = 1$'이 된다. 그리고 '$Q_a = Q-1$, $R_a = t-d$'이다.

③ '$T = N-1$'인 경우
$M = Q, q \ne 0$이다. 즉, '$Q \times 0.d = (Q.q - 0.q) \times 0.d = N - 0.q \times 0.d$'이므로 '$T = N-1$'이 된다. 그리고 '$Q_a = Q$, $R_a = t$'이다.

본 논문에서 M은 우정렬, $0.d$는 좌정렬 수를 사용하므로 T는 우정렬, t는 좌정렬 수가 된다. 상기 세 가지 조건을 간단하게 판별하기 위해서 식 (19)에서 $2N_a$를 계산한다. 이 때, T가 짝수이고 t가 0이 아닌 경우이면 상기 조건 두 번째에 해당한다.

식 (19)의 $\frac{N_b}{0.d}$를 식 (20)과 동일한 방식으로 Q_b와 $R_b = rb_{-1}rb_{-2} \cdots rb_{-w}$를 각각 계산하면 식 (21)과 같이 된다.

$$\begin{aligned}\frac{N}{0.d} &= Q_a \times 2^{2w-m-1} + Q_b \times 2^{2w-2m-1} + \frac{R_b \times 2^{2w-m-1} + (na_{2w-2m-1} \cdots na_0)}{0.d} \\ &= Q_a \times 2^{2w-m-1} + Q_b \times 2^{2w-2m-1} + \frac{nb_{2w-2m} \cdots nb_0.nb_{-1} \cdots nb_{-(2m-w-1)}}{0.d}\end{aligned} \quad (21)$$

여기서

$$\begin{aligned}nb_{2w-2m} \cdots nb_0 &= (rb_{-1} \cdots rb_{-(2w-2m-1)}) + (na_{2w-2m-1} \cdots na_0) \\ nb_{-1} \cdots nb_{-(2m-w+1)} &= rb_{-(2w-2m)} \cdots rb_{-w}\end{aligned}$$

이다.

한편, 식 (19)에서 N_b는 '$m+1$' 비트 길이이다. 여기에 2를 곱하므로 식 (21)에서 피제수는 '$m+2$' 비트

길이가 된다. $1.g$에서 g는 w 비트 길이이므로 식 (20)의 Q는 '$m+3$' 비트 길이이다. 따라서 m의 최대 값은 '$w-3$'이 된다.

식 (21)의 마지막 항을 정리하면 식 (22)와 같다.

$$\frac{nb_{2w-2m}\cdots nb_0.nb_{-1}\cdots nb_{-(2m-w-1)}}{0.d} = \frac{(nc_{w-b+1}\cdots nc_0.nc_{-1}\cdots nc_{-b})\times 2^{2m-w-b+1}}{0.d\times 2^{2m-w-b+1}} = \left(\frac{nc_{w-b+1}\cdots nc_0}{0.d}+\frac{nc_{-1}\cdots nc_{-b}}{0.d}\right)\times 2^{-(2m-w-b+1)} \quad (22)$$

식 (22)에서 b는 소수점 아래에서 절삭할 비트의 수로 '$b=w-m$'이다. 즉, 소수점 아래 항 $0.nc_{-1}\cdots nc_{-b}$은 정수 계산 시에 절삭되는 절삭 오차 a로 '$0\le a\le(1-2^{-b})\times 2^{-(2m-w-b+1)}$'이다. 절삭 오차를 방지하기 위하여 $a_{\max}<c=2^{-(2m-w-b+1)}$를 식 (21)의 마지막 항의 피제수에 더하면 식 (21)은 식 (23)과 같이 된다.

$$\frac{N}{0.d}=Q_a\times 2^{2w-m-1}+Q_b\times 2^{2w-2m-1}+\frac{N_c+1}{0.d}\times 2^{-(2m-w-b+1)} \quad (23)$$

여기서 N_c는 다음과 같다.

$$N_c = nc_{w-b+1}\cdots nc_0$$

'$Q_c=(N_c+1)\times 1.g$'를 식 (23)에 대입하여 정리하면 식 (24)가 된다.

$$\frac{N}{D}=\frac{N}{0.d}\times 2^{-L} = (Q_a\times 2^{2w-m-1}+Q_b\times 2^{2w-2m-1}+Q_c\times 2^{-(2m-w-b+1)})\times 2^{-L} \quad (24)$$

한편, 식 (23)은 식 (25)와 같이 표현할 수 있다.

$$\frac{N}{D}=(Q_a\times 2^{2w-m-1}+Q_b\times 2^{2w-2m-1})\times 2^{-L}+\frac{N_d+c}{D} \quad (25)$$

$$N_d=(nc_{w-b+1}\cdots nc_0)\times 2^{-(2m-w-b+1)} < 2^{2w-2m+1}$$

'$Q_a=\mathrm{INT}\left(\frac{N_d}{D}\right)$, $N_d=Q_d\times D+R_d$'이라고 하면, 식 (26), 식 (27)이 성립한다.

$$(N_d+c)\times 1.g\times 2^{-L}=\frac{(DQ_d+R_d+c)(1+e)}{D} = Q_d+\frac{R_d+eN_d+(1+e)c}{D} \quad (26)$$

$$\left(\frac{R_d+eN_d+(1+e)c}{D}\right)_{\max} = \frac{D-1+2^{w-2m+1}+(1+2^{-w})}{D}\times 2^{-(2m-w-b+1)}<1 \quad (27)$$

식 (26)과 식 (27)로부터 식 (23)에서 절삭 오차 보정 항 c가 계산 결과에 영향을 미치지 않는다는 것을 알 수 있다. 따라서 식 (24)는 배정도 정수 나눗셈 $\frac{N}{D}$의 정확한 해를 제공한다.

Ⅳ. 구현 및 비교

본 논문에서 제안한 배정도 정수 나눗셈기의 상태 기계 흐름도를 표 1에 나타내었다. 표 1에서는 워드 길이가 32 비트이고, 256×9 비트 근사 테이블을 적용하였으며, 근사 역수 계산은 뉴톤-랍손 방식을 사용한 경우이다.

상태-0은 제수 D를 좌정렬시키고 근사 역수 테이블을 읽어서 그 값을 좌정렬시켜 G 레지스터에 저장한다. 상태-1과 상태-2는 뉴톤-랍손 반복식을 1회 연산하고 그 결과 값이 음수이면 1의 보수를 취하여 근사 역수 $1.g$를 구한다. $1.g$에서 '1'은 항상 존재하므로 $1.g$를 왼쪽으

로 1 비트 시프트 시켜서 'g'를 G 레지스터에 저장한다. 상태-3부터 상태-6은 보다 상역수를 구하기 위하여 식 (11)과 식 (12)를 구현한 것이다. 상태-7과 상태-8은 식 (15)를 구현한 것으로 상역수를 구하여 G 레지스터에 저장한다.

상태-9에서 식 (18)의 N_a을 구하여 $N1$ 레지스터에 저장하고, 또한 이후의 연산을 위하여 N 레지스터를 변경시킨다. 상태-10과 상태-11에서 '$N_a \times 1.g$'를 연산하여 식 (20)의 M을 계산하여 $Q1$ 레지스타에 저장한다. 상태-12에서 '$M \times 0.d = T.t$'를 계산하여 T를 R 레지스터에, t를 X 레지스터에 각각 저장한다. 상태-13에서 T와 t의 값을 참조하여 Q_a, R_a를 계산하여 각각 $Q1$, R 레지스터에 저장한다.

상태-14에서 식 (19)의 na_i를 연산하여 N 레지스터에 저장하고, $Q_a \times 2^{2w-m-1}$을 Q 레지스터에 저장한다. 상태-15에서 식 (20)의 N_b을 구하여 $N1$ 레지스터에 저장하고, 식 (22)의 nc_i를 구하기 위하여 N 레지스터를 시프트 시킨다. 상태-16과 상태-17에서 '$N_b \times 1.g$'를 연산하여 식 (20)의 M을 계산하여 $Q1$ 레지스터에 저장한다. 상태-18에서 '$M \times 0.d = T.t$'를 계산하여 T를 R 레지스터에, t를 X 레지스터에 각각 저장한다. 그리고 식 (23)의 절삭 오차를 줄이기 위하여 N_c에 1을 더하는데, N_c가 N 레지스터에 저장되어 있으므로, N 레지스터의 값을 1 증가시킨다. 상태-19에서 T와 t의 값을 참조하여 Q_b, R_b를 계산하여 각각 $Q1$, R 레지스터에 저장한다.

상태-20에서 식 (23)의 '$N_c + 1$'을 계산하여 $N1$ 레지스터에 저장한다. 상태-21에서 Q 레지스터의 값에 $Q_b \times 2^{2w-2m-1}$을 더한다. 상태-22와 상태-23에서 '$(N_c + 1) \times 1.g$'을 수행하여 Q_c를 계산한다. 상태-24에서 $Q_c \times 2^{-(2m-w-b+1)}$를 Q 레지스터 값에 더한다. 상태-25에서 Q 레지스터 값에 2^{-L}를 곱하면 Q 레지스터에 $\frac{N}{D}$ 정수 나눗셈 결과가 저장된다.

제안하는 정수 나눗셈기의 블록도를 그림 1에 나타내었다.

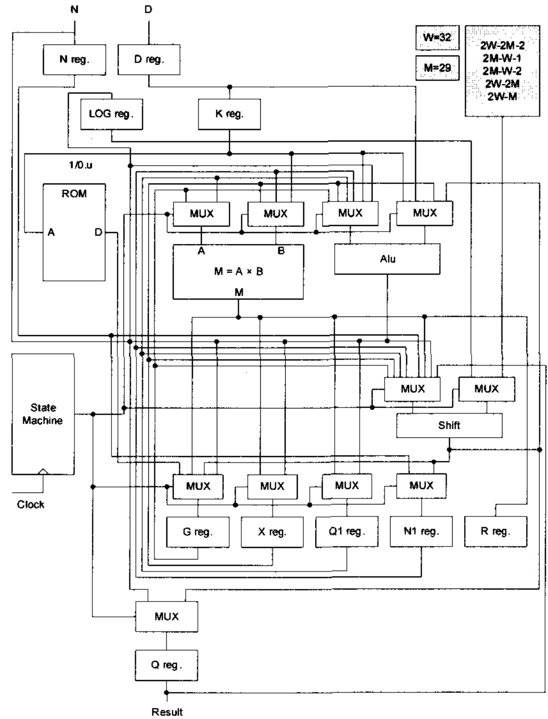


그림 1. 64 비트 배정도 정수 나눗셈기 블록도
Fig. 1. Block diagram of 64 bit Double Precision Integer Number Divider.

그림 1은 상태기계 흐름도를 기반으로 작성한 64비트 배정도 정수 나눗셈기의 블록도이다. 그림에 나타낸 것처럼 제안하는 알고리즘은 곱셈기를 이용하여 나눗셈을 수행한다. 따라서 나눗셈 연산을 하기위한 추가적인 하드웨어가 많이 필요하지 않다.

표 1. 64/32 비트 배정도 정수 나눗셈기 상태 기계 흐름도
Table 1. State machine flow of 64/32 bit double precision Integer Number Divider

```
; N(64 bit) / D(32 bit) ==> Q(64 bit) integer divide
;   LOG : 5 bit register
;   N, Q : 64 bit register
;   N1, Q1, D, K, X, G, R : 32 bit register
;   mulhu(A, B) : A(32 bit) × B(32 bit) multiplier, result is MSB 32 bit
;   256 × 9 Table
;   Constant : W=32, M=29
```

```
상태-0 : W - (Leading zero of D) - 1 ==> LOG ;
              Left justify D ==> K ;
              If (msb-1 to 0 bit of K are all '0') then
                then { N >> LOG ==> Q ; exit ; }
              Left justify 256X9_TABLE(K) ==> G ;
상태-1 : ~mulhu(G, K) ==> X ;
상태-2 : mulhu(G, X) << 1 ==> X ;
     If (msb of X is 0) then ~X << 1 ==> G ;
                        else X << 1 ==> G ;
상태-3 : mulhu(G, K) ==> X ;
상태-4 : K + X ==> X ;
상태-5 : If(msb of X is 1) then (G << 1) - X ==> X ;
              else G - X ==> X ;
     mulhu(G, X) ==> G ;
상태-6 : X - G ==> G ;
상태-7 : mulhu(G, K) ==> X ;
              G - K ==> G ;
상태-8 : G - X ==> G ;
상태-9 : N >> (2W-M) ==> N1 ;
     N AND ( (1 << (2W - M)) - 1 ) ==> N ;
상태-10 : N1 << 1 ==> X ;
     mulhu(G, X) ==> Q1 ;
상태-11 : Q1 + X ==> Q1 ;
상태-12 : Q1 * K ==> X ;
     mulhu(Q1, K) ==> R ;
상태-13 : If( ( bit 0 of R is 0 ) AND (X is not 0) )
              then { Q1 - 1 ==> Q1 ; K - X ==> X ; }
                else -X ==> X ;
상태-14 : N + (X << (W-M-1)) ==> N ;
     Q1 << (2W-M-2) ==> Q ;
상태-15 : N >> (2W-2M) ==> N1 ;
     ( N AND ( (1 << (2W-M)) - 1 ) ) <<     (2M-W-2) ==> N ;
상태-16 : N1 << 1 ==> X ;
     mulhu(G, X) ==> Q1 ;
              N ==> N1 ;
상태-17 : Q1 + X ==> Q1 ;
상태-18 : Q1 * K ==> X ;
     mulhu(Q1, K) ==> R ;
              N1 + 1 ==> N1 ;
상태-19 : If ( ( bit 0 of R is 0 ) AND (X is not 0) )
              then { Q1 - 1 ==> Q1 ; K - X ==> X ; }
              else -X ==> X ;
상태-20 : N1 + (X >>3) ==> N1 ;
상태-21 : Q + ( Q1 << (2W-2M-2) ) ==> Q
상태-22 : mulhu(G, N1) ==> X ;
상태-23 : N1 + X ==> Q1 ;
상태-24 : Q + ( Q1 >> (2M - W - 1) ) ==> Q ;
상태-25 : Q >> LOG ==> Q ;
```

본 논문에서 제안한 배정도 정수 나눗셈은 C 언어를 사용하여 모델링하고 시뮬레이션을 수행하여 동작을 확인하였다. 제수의 상역수가 정확하게 구해지면 배정도 나눗셈이 정확하게 계산되므로 본 논문에서는 제수의 상역수를 구하는 것이 필수적이다. 제수가 32 비트 나눗셈인 경우에는 전수 계산, 64 비트 정수 나눗셈은 RC5를 사용하여 구한 난수 10^9개에 대하여 상역수를 계산하여 모두 정확한 값을 계산하는 것을 확인하였다. 또한 64/32 비트와 128/64 비트 나눗셈은 뉴톤-랍손 알고리즘과 골드스미트 알고리즘에서 RC5를 사용하여 구한 난수 10^9개에 대하여 수행하여 모두 정확한 계산 결과가 구해지는 것을 확인하였다.

본 논문의 알고리즘과 종래 SRT 알고리즘을 하드웨어로 구현했을 때의 배정도 정수 나눗셈의 동작 속도를 비교하여 표 2에 보인다.

표 2. 정수 나눗셈 비교 (no. of clock)
Table 2. Compare of integer number divide. (no. of clock)

	SRT		제안 알고리즘		
	MOD-4	MOD-8	no table	16×5 table	256×9 table
64/32 bit divide	34	24	30	28	26
128/64 bit divide	66	45	32	30	28

표 2로부터 본 연구에서 제안한 알고리즘의 하드웨어 동작 속도가 64/32 비트 배정도 정수 나눗셈에서는 종래 SRT MOD-4 알고리즘에 비하여 빠르고, SRT MOD-8 알고리즘보다는 다소 느리다. SRT MOD-8은 하드웨어가 상당히 복잡하지만 본 연구에서 제안한 알고리즘은 '32 × 32 = 64 비트' 곱셈기만을 사용하는 장점을 가진다.

128/64 비트 배정도 정수 나눗셈에서는 본 연구에서 제안한 알고리즘이 기존 SRT 알고리즘보다 상당히 속도가 빠른 것을 알 수 있다.

기존 SRT 알고리즘은 비트 단위로 연산을 수행한다. 즉, SRT MOD-4는 1회 연산에 2 비트씩, MOD-8은 1회 연산에 3 비트씩 몫이 구해진다. 반면에 제안한 알고리즘은 상역수를 계산한 이후에는 1회 연산에 'w-3' 비트씩 몫이 구해지기 때문에 워드 길이가 큰 경우에 기존 SRT 알고리즘보다 연산 속도가 빠르다.

V. 결 론

정수 나눗셈 연산은 멀티미디어 처리 등의 여러 분야에서 사용하는 빈도는 높지 않으나, 이용할 경우 응용분야에 따라서 시스템 성능에 영향을 미친다. 그래서 본 논문에서는 정수 곱셈기를 사용하여 배정도 정수 나눗셈을 수행하는 새로운 알고리즘을 제안한다.

이 방법은 곱셈을 반복해서 제수의 역수를 구하고 이를 피제수에 곱하여 나눗셈을 수행하는 것이다. 그런데 이 방법은 근사 값으로 계산을 수행하므로 나눗셈의 결과를 보정해야만 정확한 값을 구할 수 있다. 따라서 보정을 위한 추가적인 과정이 필요하다. 그러나 제안하는 알고리즘은 송홍복 등이 제안한 상역수를 이용하여 근사 값 보정을 위한 추가적인 과정을 수행하지 않도록 하였다. 즉, 제수 D 가 '$D=0.d\times 2^{L}$, $0.5<0.d<1.0$'일 때, '$0.d\times 1.g=1+e$, $e<2^{-w}$'가 되는 '$\frac{1}{D}$'의 근사 값인 상역수 '$1.g\times 2^{-L}$'를 구한 후, 피제수 N 을 식 (18)과 같이 나누어 위에서 구한 상역수를 곱하여 부분 몫을 구하고, 구해진 부분 몫을 더하여 배정도 나눗셈 연산을 수행하였다.

본 논문에서 제안한 정수 나눗셈은 C 언어를 사용하여 모델링하고 시뮬레이션을 수행하여 동작을 확인하였다. 32 비트 정수 나눗셈은 전수 계산, 64 비트 정수 나눗셈은 RC5를 사용하여 구한 난수 10^9개에 대하여 계산한 결과 모두 정확한 값이 구해지는 것을 확인하였다. 그리고 뉴톤-랍손 알고리즘과 골드스미트 알고리즘 모두에서 동일한 결과를 얻을 수 있었다. 제안한 알고리즘은 C 모델링을 통한 시뮬레이션을 수행하였으며 하드웨어 구현을 위한 최적화 연구는 수행하지 않았다. 이에 관한 연구는 추후 연구 과제로 남겨둔다.

본 논문에서 제안한 알고리즘은 'w bit × w bit = $2w$ bit' 곱셈기만을 사용하므로 마이크로프로세서를 제작할 때 나눗셈 구현을 위한 추가적인 하드웨어가 필요하지 않다. 그리고 기존의 알고리즘인 SRT 알고리즘과 비교했을 때 연산을 수행하는 속도가 빠르다는 장점을 가진다. 또한, 워드 단위로 연산을 수행하기 때문에 비트 단위로 계산을 해야 하는 기존의 나눗셈 알고리즘에 비해 컴파일러 작성에도 더 좋은 적합성을 가진다.

따라서 본 논문에서 제안하는 정수 나눗셈 알고리즘은 나눗셈을 하드웨어로 구현하거나 컴파일러로 구현하는 경우 모두에서 기존의 나눗셈 알고리즘에 비해 효율성이 높다. 그러므로 마이크로프로세서 및 하드웨어 크기가 제한적인 SOC(System On Chip) 등의 구현에 폭넓게 사용될 수 있을 것이다.

참고문헌

[1] Thomas L. Adams and Richard E. Zimmerman, "An Analysis of 8086 Instruction Set Usage in MS-DOS program," Proceeding of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 152-160, Apr. 1989.

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

[3] Kadowaki Shunsuke, et al, "Integer Divider Using Absolute Value Computation of Redundant Binary Numbers," IEIC Technical Report, Vol. 104, No. 78, pp. 13-18, 2004.

[4] Chua-Chin Wang, et al, "Design of a cycle-efficinet 64-b/32-b integer divisor using a table-sharing algorithm," IEEE Transactions on VLSU systems, Vol. 11, Issue 4, pp. 733-740, Aug. 2003.

[5] S. Y. R. Li, "Fast Constant Division Routines, " IEEE Transactions on Computers, Vol. C34-9, pp. 866-869, Sep. 1985.

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

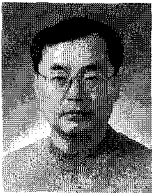
[7] R. Goldschmidt, Application of division by convergence, master's thesis, MIT, Jun. 1964.

[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] 송홍복, 박창수, 조경연, "개선된 역수 알고리즘을 사용한 정수 나눗셈기", 한국해양정보통신학회논문지, Vol 12, No. 7, pp. 1218-1226, Jul. 2008.

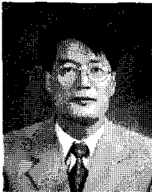
[10] D. DasSarma and D. Matula, "Measuring and Accuracy of ROM Reciprocal Tables", IEEE Transactions of Computer, Vol. 43, No. 8, pp. 932-930, Aug.. 1994.
[11] S. Oberman, "Floating Point Division and Square Root Algorithms and Implementation in the AMD-K7 Microprocessors", Proc. 14th IEEE Symp. Computer Arithmetic, pp. 106-115, Apr. 1999.
[12] Matthew Frank, "DESIGN OF AN INTEGER RECIPROCAL ALGORTIHM", www.cag.lcs.mit.edu/raw/memo/12/div.html, Aug. 1999.
[13] Robert Alverson, "Integer Division Using Reciprocals," Proceedings of the Tenth Symposium on Computer Arithmetic, Grenoble, France, pp 186-190, Jun. 1991.

저자소개



조경연(Gyeong-Yeon Cho)

1983~1990 : 삼보컴퓨터 기술연구소 부장
1990년 : 인하대학교 공학박사
1991년~현재 : 부경대학교 IT융합응용공학과 교수
1998년-현재 : (주)에이디칩스 기술 고문
※관심분야 : 전산기 구조, 반도체 회로 설계, 정보보호



송홍복(Hong-Bok Song)

1985~1990년 :동의과학대학 조교수
1990년 : 동아대학교 공학박사
1989~1990년 : 일본구주공대 객원연구원
1991년-현재 : 동의대학교 공과대학 전자공학과 교수
1994-1995년 : 일본 미야쟈키대학 전기. 전자 공학부 (POST-DOC)
※관심분야 : 다치논리이론 및 다치논리시스템 설계, VLSI설계, 마이크로프로세서 응용