# 개선된 역수 알고리즘을 사용한 정수 나눗셈기

송홍복* · 박창수** · 조경연***

## The Integer Number Divider Using Improved Reciprocal Algorithm

Hong-Bok Song* · Chang-Soo Park** · Gyeong-Yeon Cho***

이 논문은 2007년도 동의대학교 연구년 교수지원에 의해서 연구되었습니다.


### 요 약

반도체 집적 기술의 발달과 컴퓨터에서 멀티미디어 기능의 사용이 많아지면서 보다 많은 기능들이 하드웨어로 구현되기를 원하는 요구가 증가되고 있다. 그래서 현재 사용되는 대부분의 32 비트 마이크로프로세서는 정수 곱셈기를 하드웨어로 구현하고 있다. 그러나 나눗셈기는 기존의 알고리즘인 SRT 알고리즘의 방식이 하드웨어 구현상의 복잡도와 느린 동작 속도로 인해 특정 마이크로프로세서에 한해서만 하드웨어로 구현되고 있다.

본 논문에서는 'w bit × w bit = 2w bit' 곱셈기를 사용하여 $\frac{N}{D}$ 정수 나눗셈을 수행하는 알고리즘을 제안한다. 즉, 제수 $D$ 의 역수를 구하고 이를 피제수 $N$ 에 곱해서 정수 나눗셈을 수행한다. 본 논문에서는 제수 $D$ 가 '$D=0.d\times 2^{L},\ 0.5<0.d<1.0$'일 때, '$0.d\times 1.g=1+e,\ e<2^{-m}$'가 되는 '$\frac{1}{D}$'의 근사 값 '$1.g\times 2^{-L}$'을 가칭 상역수라고 정의하고, 상역수를 구하는 알고리즘을 제안하고, 이렇게 구한 상역수 '$1.g\times 2^{-L}$'을 피제수 $N$에 곱하여 $\frac{N}{D}$ 정수 나눗셈을 수행한다. 제안한 알고리즘은 정확한 역수를 계산하기 때문에 추가적인 보정이 요구되지 않는다.

본 논문에서 제안하는 알고리즘은 곱셈기만을 사용하므로 마이크로프로세서를 구현할 때 나눗셈을 위한 추가적인 하드웨어가 필요 없다. 그리고 기존 알고리즘인 SRT 방식에 비해 빠른 동작속도를 가지며, 워드 단위로 연산을 수행하기 때문에 기존의 나눗셈 알고리즘보다 컴파일러 작성에도 적합하다. 따라서, 본 논문의 연구 결과는 마이크로프로세서 및 하드웨어 크기에 제한적인 SOC(System on Chip) 구현 등에 폭넓게 사용될 수 있다.

### ABSTRACT

With the development of semiconductor integrated technology and with the increasing use of multimedia functions in computer, more functions have been implemented as hardware. Nowadays, most microprocessors beyond 32 bits generally implement an integer multiplier as hardware. However, as for a divider, only specific microprocessor implements traditional SRT algorithm as hardware due to complexity of implementation and solw speed.

This paper suggested an algorithm that uses a multiplier, 'w bit × w bit = 2w bit', to process $\frac{N}{D}$ integer division. That is, the reciprocal number $D$ is first calculated, and then multiply dividend $N$ to process integer division. In this paper, when the divisor $D$ is



* 동의대학교 전자공학과

** 부경대학교 누리 지능형 지구환경재해 정보관리 전문인력 양성사업단

*** 부경대학교 전자컴퓨터정보통신공학부


접수일자 2008. 03. 04

'$D = 0.d \times 2^L$, $0.5 < 0.d < 1.0$', approximate value of '$\frac{1}{D}$', '$1.g \times 2^{-L}$', which satisfies '$0.d \times 1.g = 1 + e$, $e < 2^{-n}$', is defined as over reciprocal number and then an algorithm for over reciprocal number is suggested. This algorithm multiplies over reciprocal number '$1.g \times 2^{-L}$' by dividend $N$ to process $\frac{N}{D}$ integer division. The algorithm sugested in this paper doesn't require additional revision, 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 and performs operation by word unit, accordingly it is more suitable to make compiler than the existing division 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, 정수나눗셈, 역수알고리즘, integer divider, reciprocal algorithm

## Ⅰ. 서론

최근 컴퓨터에서 멀티미디어 기능이 보편화되면서 많은 기능들이 하드웨어로 구현되기를 원하는 요구가 증대되고 있다. 그런 기능들 가운데 하나가 정수 곱셈기이다. 얼마 전만 해도 곱셈기를 하드웨어로 구현하기 위해서는 많은 비용이 들었으므로, 컴파일러에서 시프트 연산을 사용해 소프트웨어로 구현하였다. 그러나 근래는 반도체 공정 기술의 발달로 반도체 집적 기술이 향상되어 대부분의 32비트 이상 마이크로프로세서는 정수 곱셈기를 하드웨어로 구현하여 내장하고 있고, 32 비트 곱하기 32 비트를 수행하여 64 비트의 결과를 얻는 곱셈 연산을 한 클럭에 수행할 수 있다.

정수 곱셈에 못지않게 정수 나눗셈도 비록 출현 빈도는 높지 않지만 시스템 성능에 영향을 미치는 연산이다. 실제 표 계산(spread sheet) 프로그램에서 정수 나눗셈의 출현 빈도는 약 10% 정도로 조사되고 있다[1]. 그런데 정수 나눗셈 연산을 컴파일러로 구현할 경우 계산 속도가 상당히 느려서 시스템 전체의 성능을 저하시키는 원인이 된다.

이러한 문제점을 해결하기 위해서 뺄셈을 반복하여 나눗셈을 수행하는 SRT[2] 방식을 하드웨어로 구현하는데, 이는 추가적인 하드웨어가 소요되고 또한 SRT의 연산 속도가 느려서 파이프라인 구조가 복잡하게 되는 단점을 가진다. 특히, 64 비트 정수 나눗셈에서는 연산 속도가 더욱 느려지는 문제점이있다.

최근에는 컴파일러에 작은 정수에 대한 역수를 미리 계산해 놓는 방식을 채택해서 사용하는데, 실제로 제수가 작은 정수인 나눗셈의 출현 빈도가 높으므로 상당히 효율적인 방법이다[3, 4, 5].

정수 나눗셈은 대부분의 32 비트 이상 마이크로프로세서에서 하드웨어로 구현되어 있는 정수 곱셈기를 사용하여 수행할 수도 있다. 이 방법은 곱셈을 반복하여 제수의 역수를 구하고 이를 피제수에 곱하는 방식을 이용해서 나눗셈을 수행하는 것이다. 곱셈을 반복하는 것은 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. 그런데 이 방법은 근사 값으로 계산을 하게 되므로 나눗셈의 결과를 보정하여 정확한 값을 구하는 추가적인 과정이 요구된다.

본 논문에서는 $\frac{N}{D}$ 정수 나눗셈에 있어서 제수 $D$가 '$D = 0.d \times 2^L$, $0.5 < 0.d < 1.0$'일 때, '$0.d \times 1.g = 1 + e$, $e < 2^{-n}$'가 되는 '$\frac{1}{D}$'의 근사 값 '$1.g \times 2^{-L}$'를 가칭 상역수라고 정의하고, 상역수를 구하는 알고리즘을 제안한다. 그리고 상역수'$1.g \times 2^{-L}$'를 피제수 $N$에 곱하여 $\frac{N}{D}$ 정수 나눗셈을 수행한다. 본 논문에서 제안하는 알고리즘은 정수 곱셈기만을 사용하여 나눗셈을 수행하므로 추가적인 하드웨어가 요구되지 않는다. 그리고 기존의 나눗셈 알고리즘인 SRT 나눗셈 방식과 비교했을 때 연산속도가 빠르다는 장점을 가진다.

본 논문에서 제안하는 정수 나눗셈 알고리즘은 C언어를 사용하여 모델링하여 동작을 확인하였다.

본 논문의 구성은 다음과 같다. 2장은 곱셈을 반복하여 나눗셈을 수행하는 알고리즘인 뉴톤-랍손과 골드스미트 알고리즘을 소개한다. 3장은 상역수를 구하는 알고리즘을 제안하고, 4장에서는 상역수를 피제수에 곱하는

정수 나눗셈 알고리즘을 제안한다. 그리고 5장에서는 제안한 알고리즘을 C 언어를 사용해서 모델링하여 동작을 확인하고, 기존의 SRT 알고리즘과 성능을 비교한다. 마지막 6장에서는 결론을 맺는다.

## Ⅱ. 근사역수

곱셈을 반복해서 역수를 구하는 방식은 뉴톤-랍손 역수 알고리즘과 골드스미트 나눗셈 알고리즘이 있다. 뉴톤-랍손 역수 알고리즘은 역수의 근사값을 초기 값으로 하고, 반복 연산으로 오차를 줄여 나가는데 반복할 때 마다 오차는 자승으로 줄어든다. 그리고 1회 반복 연산에 2회의 곱셈이 필요하다.

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. 골드스미트 나눗셈 알고리즘은 1회의 연산에 서로 독립적인 2회의 곱셈이 필요하다. 따라서 두 개의 곱셈기를 사용하면 연산 시간을 줄일 수 있어 IBM RS/6000, AMD K7 프로세서 등에서 사용되고 있다[9].

### 2.1 뉴톤-랍손 역수 알고리즘

임의의 정수 $D$ 의 역수를 구하기 위하여 함수 '$f(X)=D-\frac{1}{X}$'을 정의한다. 뉴톤-랍손 알고리즘에서 $X_i$를 $X$ 의 근사 값이라고 하면 $X_{i+1}$은 식 (1)과 같이 된다.

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

본 논문에서는 정수 곱셈기를 사용하므로 정수 $D$를 좌정렬 형식으로 표현하면 '$0.d\times 2^L,\ D<2^L$' 이 된다. 여기에서 $0.d$를 가수부, $2^L$을 지수부라고 표현한다. 예를 들어 32 비트 마이크로프로세서에서 정수 7을 위의 형식으로 표현하면 가수부는 0xE0000000, 지수부는 $2^3$이다.

정수 $D$가 $D=2^L$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.

$$0.d=0.u+v \qquad (2)$$

식 (2)에서 $u$와 $v$의 길이를 각각 $n_u$, $n_v$ 비트로 정의한다. 여기서 $v$는 '$0\le v<2^{-n_u}$'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 $e_u$는 근사에 따른 오차이다. 근사 테이블 $T(u)$를 $X$의 초기 근사값 $X_0$로 정의한다.

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

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

where RN is round to nearest

식 (3)에서 $T(u)$의 소수점 이하 길이를 $t$비트라고 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 $w$는 워드 길이이다. 본 논문에서 사용하는 뉴톤-랍손 역수 알고리즘은 식 (4)와 같다.

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

### 2.2 골드스미트 나눗셈 알고리즘

$\frac{X_0}{D_0}$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 $D_n$은 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 $X_n$이 나눗셈 결과이다. $X_0$가 1.0이면 식 (5)는 $D_0$의 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.

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

## Ⅲ. 상역수

가칭 w 비트 길이 정수 $D$의 상역수를 다음과 같이 정의한다.

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

뉴톤-랍손 또는 골드스미트 알고리즘을 사용하여 $\frac{1}{0.d}$의 근사 값 '$X_n = \frac{1}{0.d} + e_n$'을 계산한다. 여기서 $e_n$은 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 $e_n$에 의해서 근사 역수의 범위가 '$X_n < 1.0$'이 되는 경우가 발생한다. Matthew Frank[11]와 Robert Alverson[12]의 알고리즘에서는 근사 역수의 범위가 위와 같이 되었을 때 나눗셈의 결과가 틀린 값을 가지게 된다. 본 논문에서는 모든 수는 좌정렬 형식을 사용하므로 근사 역수 $X_n$의 최상위 비트가 '0'이면 근사 역수의 범위가 '$X_n < 1.0$'이다. 이 경우에는 근사 역수 $X_n$의 1의 보수를 취해서 $\frac{1}{0.d}$의 근사 값으로 한다.

2장에서 구한 $\frac{1}{0.d}$의 근사 값 $X_n$은 식 (7)과 같이 표현된다.

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

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

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

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

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

식(8)과 식 (9)로부터 $\beta$를 구하면 식 (10)이 된다.

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

따라서 $X_{n+1}$은 식 (11)과 같이 된다.

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

식(11)로부터 $X_{n+1}$의 오차는 $X_n$오차의 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 $X_n$을 왼쪽으로 한 비트 시프트 시키면 레지스터에 저장된 값은 식 (7)의 $\mathcal{L}$만을 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.

## Ⅳ. 정수 나눗셈

정수 나눗셈은 식 (17)로 표현할 수 있다.

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

식 (17)에서 제수 $D$ 를 '$D = 0.d \times 2^{-L}$, $0.5 < 0.d < 1.0$'로 표현하고, '$0.d \times 1.g = 1 + e$, $e < 2^{-w}$'가 되는 $0.d$의 상역수 '$1.g$'를 2장과 3장에서와 같이 계산한다. 그리고 피제수 $N$에 제수 $D$의 상역수 '$1.g \times 2^{-L}$'을 곱해서 정리하면 식 (18)이 된다.

$$N \times 1.g \times 2^{-L} = (DQ + R) \times 1.g \times 2^{-L} \quad (18)$$
$$= \frac{(DQ + R)(1 + e)}{0.d \times 2^{L}}$$
$$= Q + \frac{eN + R}{D}$$

또한, 식 (19)가 성립하므로 식 (18)로부터 정확한 $Q$ 값을 계산할 수 있다.

$$\left(\frac{eN + R}{D}\right)_{\max} = \frac{2^{-w}(2^{w} - 1) + D - 1}{D} \quad (19)$$
$$= \frac{D - 2^{-w}}{D} < 1.0$$

한편, '$1.g$'의 소수점 이하 자릿수는 워드 길이 $W$와 같다. 그리고 레지스터 또는 변수의 길이는 워드 길이와

동일하므로 식 (18)을 변형하여 식 (20)과 같이 계산한다.

$$N\times 1.g\times 2^{-L}=\frac{N\times 1.g}{2}\times 2^{-L+1}$$
$$\frac{N\times 1.g}{2}=\frac{N+N\times 0.g}{2}=\frac{N+T}{2} \qquad (20)$$
$$=T+\frac{N-T}{2},\quad T=N\times 0.g$$

## V. 구현 및 비교

본 논문에서 제안한 정수 나눗셈기의 상태 기계 흐름도를 표 1에 나타내었다. 표 1에서는 워드 길이가 64 비트이고 256×9 비트 근사 테이블을 적용하였으며, 근사 역수 계산은 뉴톤-랍손 방식을 사용한 경우이다. 그리고 정수 나눗셈기의 블록도를 그림 1에 나타내었다.

표 1의 상태 기계 흐름도에서 상태-0은 제수 $D$를 좌정렬시키고 근사 역수 테이블을 읽어서 그 값을 좌정렬시켜 $G$ 레지스터에 저장한다. 상태-1과 상태-2는 뉴톤-랍손 반복식을 1회 연산한 것이다. 상태-3과 상태-4는 뉴톤-랍손 반복식을 1회 더 연산하고 그 결과 값이 음수이면 1의 보수를 취하여 근사 역수 $1.g$를 구한다. $1.g$에서 '1'은 항상 존재하므로 $1.g$를 왼쪽으로 1 비트 시프트 시켜서 '$g$'를 $G$ 레지스터에 저장한다. 상태-5부터 상태-8은 보다 정확한 역수를 구하기 위하여 식 (12)와 식 (13)을 구현한 것이다. 상태-9와 상태-10은 식 (16)을 구현한 것으로 상역수를 구하여 $G$ 레지스터에 저장한다. 상태-11과 상태-12는 식 (20)을 구현한 것으로 그 결과 값을 $G$레지스터에 저장한다. 상태-13에서 $2^{l}$을 곱해주면 정수 나눗셈 결과 값이 $Q$레지스터에 저장된다.

정확한 정수 나눗셈 결과를 얻기 위해서는 식 (21)을 만족시키는 상역수 $1.g$를 계산해야 한다.

$$0.d\times 1.g=1+e,\quad e<2^{-w} \qquad (21)$$

표 1. 64 비트 정수 나눗셈기 상태 기계 흐름도
Table 1. State machine flow of 64 bit Integer Number Divider

```
; N(64 bit) / D(64 bit) ==> Q(64 bit) integer divide
;   LOG : 6 bit register
;   N, D, Q, K, X, G : 64 bit register
;  mulhu(A, B) : A(64 bit) × B(64 bit) multiplier, result is MSB 64 bit
;   256×9 Table

state-0 : W ← (Leading zero of D)  1 ==> LOG ;
      Left justify D ==> K ;
      If (msb-1 to 0 bit of K are all '0')
        then { N >> LOG ==> Q ; exit ; }
      Left justify 256×9_TABLE(K) ==> G ;
state-1 : ~mulhu(G, K) ==> X ;
state-2 : mulhu(G, X) << 1 ==> G ;
state-3 : ~mulhu(G, K) ==> X ;
state-4 : mulhu(G, X) << 1 ==> X ;
      If (msb of X is 0) then ~X << 1 ==> G ;
        else X << 1 ==> G ;
state-5 : mulhu(G, K) ==> X ;
state-6 : K + X ==> X ;
state-7 : If (msb of X is 1) then (G << 1)  X ==> X ;
        else G − X ==> X ;
      mulhu(G, X) ==> G ;
state-8 : X − G ==> G ;
state-9 : mulhu(G, K) ==> X ;
      G − K ==> G ;
state-10 : G − X ==> G ;
state-11 : mulhu(G, N) ==> X ;
state-12 : (N − X) >> 1 ==> G
state-13 : (X + G) >> LOG ==> Q
```

본 논문에서는 32 비트 정수에 대해서는 전수 계산을 하여 식 (21)을 만족하는 것을 확인했으며, 64 비트 정수에 대하여는 RC5를 사용하여 구한 난수 $10^9$개에 대하여 식 (21)을 만족하는 것을 확인했다. 그러나 64 비트 정수에 대하여는 전수 계산을 하지 않았기 때문에 표 1의 상태 흐름도가 식 (21)을 만족하지 않을 가능성도 있다. 이를 보완하기 위해서 상태-9, 상태-10, 상태-11을 표 2에 나타낸 것과 같이 수정하면, 모든 64 비트 정수에서 정확한 계산이 되는 것을 보증할 수 있다.

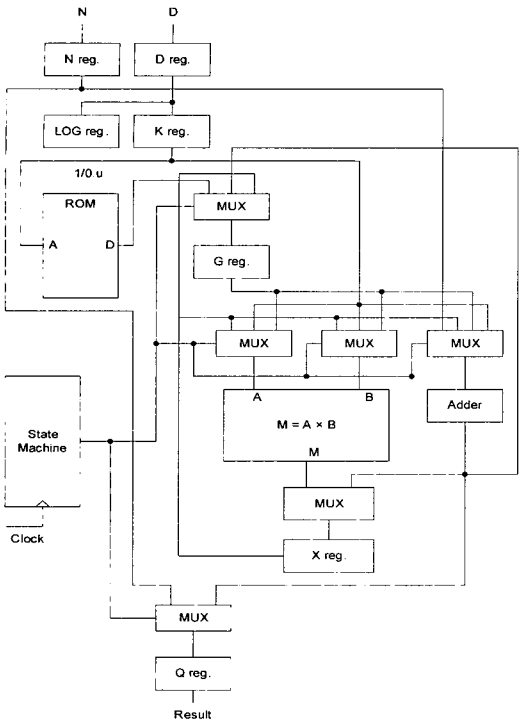


그림 1. 64 비트 정수 나눗셈기 블록도
Fig 1. Block diagram of 64 bit Integer Number Divider

본 논문에서 제안한 정수 나눗셈기는 C 언어를 사용해 모델링했다. 32 비트 정수 나눗셈은 전수 계산을 수행하였고, 64 비트 정수 나눗셈에 대해서는 RC5를 사용해서 구한 난수 $10^9$개에 대하여 계산을 수행하였다. 그 결과 32 비트와 64 비트 정수 나눗셈 모두 정확하게 계산되는 것을 확인하였다. 또한 뉴톤-랍손과 골드스미트 알고리즘 모두에서 동일한 결과를 얻을 수 있었다.

표 2. 수정된 64 비트 정수 나눗셈기 상태 기계의 일부
Table 2. A part of state machine flow of the modified 64 bit Integer Number Divider

```
state-9 : mulhu(G, K) ==> X ;
state-10 : X + K ==> X ;
state-11 : If (X = 0)
        then { mulhu(G, N) ==> X ;
            goto state-12 ; }
        else { G - X ==> G ;
            goto state-9 ; }
```

본 논문에서 제안한 알고리즘과 기존의 나눗셈 알고리즘인 SRT 알고리즘을 하드웨어로 구현했을 때의 동작속도를 비교하여 표 3에 보였다.

표 3. 단정도 정수 나눗셈 비교(no. of clock)
Table 3. Compare of single precision integer number divide(no. of clock)

| | SRT | | 제안 알고리즘 | | |
|---|---|---|---|---|---|
| | MOD-4 | MOD-8 | 테이블이 없을 때 | 16×5 테이블 | 256×9 테이블 |
| 32 비트 나눗셈 | 18 | 13 | 16 | 14 | 12 |
| 64 비트 나눗셈 | 34 | 24 | 18 | 16 | 14 |

표 3으로부터 본 논문에서 제안하는 알고리즘의 하드웨어 동작속도가 종래의 SRT 알고리즘과 비교했을 때 더 빠르다는 것을 알 수 있다.

## VI. 결 론

정수 나눗셈은 나타나는 빈도가 높지 않으나 응용분야에 따라서 시스템 성능에 영향을 미치는 연산이다. 이런 정수 나눗셈을 컴파일러로 구현했을 때는 연산 속도가 상당히 느려 전체적인 시스템의 성능을 저하시키는 원인이 된다. 그리고 이런 문제점을 해결하기 위해 뺄셈을 반복하는 SRT 알고리즘을 하드웨어로 구현하는 종래 방식도 추가적인 비용이 드는 등의 단점이 존재한다.

또 다른 방법으로는 대부분의 32 비트 마이크로프로세서에서 하드웨어로 구현한 정수 곱셈기를 사용하는 것이다. 이 방법은 곱셈을 반복해서 제수의 역수를 구하고 이를 피제수에 곱하여 나눗셈을 수행하는 것이다. 곱셈을 반복하는 대표적인 알고리즘으로 뉴톤-랍손 역수 알고리즘과 골드스미트 알고리즘이 있다. 그런데 이 방법은 근사값으로 계산을 수행하므로 나눗셈의 결과를 보정해야만 정확한 값을 구할 수 있다. 따라서 보정을 위한 추가적인 과정이 필요하다.

본 논문에서는 'w bit × w bit = 2w bit' 곱셈기를 사용하여 $\frac{N}{D}$ 정수 나눗셈을 수행하는 알고리즘을 제안했다. 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 구하는 알고리즘을 제안했고, 상역수 '$1.g\times 2^{-L}$'를 피제수 $N$에 곱하여 $\frac{N}{D}$ 정수 나눗셈을 수행하였다.

본 논문에서 제안한 정수 나눗셈을 C 언어를 사용하여 모델링하고 시뮬레이션을 수행하여 동작을 확인하였다. 32 비트 정수 나눗셈은 전수 계산, 64 비트 정수 나눗셈은 RC5를 사용하여 구한 난수 $10^9$개에 대하여 계산하여 모두 정확한 값을 계산하는 것을 확인하였다. 그리고 뉴톤-랍손 알고리즘과 골드스미트 알고리즘 모두에서 동일한 결과를 얻을 수 있었다.

본 논문에서 제안한 알고리즘은 'w bit × w bit = 2w bit' 곱셈기만을 사용하므로 마이크로프로세서를 제작할 때 나눗셈 구현을 위한 추가적인 하드웨어가 필요하지 않다. 그리고 종래의 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 구현할 때 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] E. Artzy, J. A. Hinds, and H. J. Saal, "A Fast Division Technique for Constant Divisors," Communications of the ACM, Vol. 19-2, pp. 98-101, Feb. 1976.

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

[5] Daniel J. Magenheimer, Liz Peters, Karl Pettis, and Dan Zuras, "Integer Multiplication and Division on the HP Precision Architecture," Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 90-99, Apr. 1987.

[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] 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] 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] Matthew Frank, "DESIGN OF AN INTEGER RECIPROCAL ALGORTIHM", www.cag.lcs.mit.edu/raw/memo/12/div.html, Aug. 1999.

[12] Robert Alverson, "Integer Division Using Reciprocals," Proceedings of the Tenth Symposium on Computer Arithmetic, Grenoble, France, pp 186-190, Jun. 1991.

## 저자소개


**송홍복(Hong-Bok Song)**

1983년 광운대학교 전자통신공학과 졸업
1985년 인하대학교 대학원 전자공학과 졸업(공학석사)
1985-1990년 동의공업대 전자통신과 조교수
1989-1990년 일본 구주공대 정보공학부 객원연구원
1990년 동아대학교 대학원 전자공학과 졸업(공학박사)
1994-1995년 일본 미야쟈키 대학교 전기.전자공학부 (POST-DOC)
1991년-현재 동의대학교 전자공학과 교수
※관심분야 : 다치논리 이론 및 시스템 설계, VLSI설계, 마이크로프로세서 응용

**박창수(Chang-Soo Park)**

1995년 인제대학교 전자공학과 졸업(공학사)
2001년 부경대학교 컴퓨터공학과 졸업(공학석사)
2007년 부경대학교 컴퓨터공학과 졸업(공학박사)
2008년 부경대학교 누리 지능형 지구환경재해 정보관리 전문인력 양성사업단 초빙계약교수
※관심분야 : 반도체 회로설계, 암호 알고리즘, 컴퓨터 구조

**조경연(Gyeong-Yeon Cho)**

1990 인하대학교 공과대학 전자공학과 공학박사
1983-1991 삼보컴퓨터 기술연구소 책임연구원
1991-2000 삼보컴퓨터 기술연구소 기술고문
1998-현재 에이디칩스 기술고문
1991-현재 부경대학교 공과대학 전자컴퓨터정보통신공학부 교수
※관심분야 : 전산기구조, 반도체 회로 설계, 암호알고리즘