논문 99-02-02

고속나눗셈 연산기를 위한 영역변환상수 검색테이블의 설계 및 구현

Design and implementation of pre-scaling look-up table for very-high radix divider

李 炳 錫*, 李 禎 娥*

(Byeong-Seok Lee* and Jeong-A Lee*)

요 약

본 논문에서는 높은 자릿수를 이용하는 고속나눗셈 연산기의 성능을 향상시키는 한 방편으로, 나눗셈 연산시에 영역변환상수를 계산하지 않고 직접 검색테이블에 저장하는 방법을 제시하고자 한다. 그리고 영역변환상수 검색테이블의 크기를 줄이기 위하여 영역변환상수의 범위를 분석하여서 검색테이블의 크기를 일차적으로 줄였고, 범위를 분석한 영역변환상수를 두 개의 검색테이블로 구성하여서 이차적으로 크기를 줄였다. 제기된 방법론은 검색테이블의 크기를 줄이면서 나눗셈 연산기의 연산순환주기를 한 단계 낮출 수 있고, 연산순환주기를 감소하기 위한 기본 자릿수 선택시에 매우 유리하기 때문에 추후 다양한 응용이 기대된다.

Abstract

In this paper, we propose a new technique which allows to store the pre-scaling constants directly in a table thus eliminating the cycle for computing pre-scaling constants. Especially we analyzed the range of pre-scaling constants and rearranged them in a carry-save form using two look-up tables so that the size of the tables can be reduced significantly. The resulting scheme is compared with the previously developed method and shown to be effective with respect to area and time to implement the high-radix divider.

Keyword : Very-high radix, Divider, Look-up table, Pre-scaling, ROM-based

I. 서 론

나눗셈 알고리즘은 다른 연산 알고리즘인 덧셈이나 곱셈 알고리즘에 비해 복잡하고, 수행 시간이 길기 때문에 일반적으로 소프트웨어로 처리하는 경우가 대부분이다. 또한 하드웨어로 처리하는 경우에도, 덧셈은 2 ~ 4번, 곱셈은 2 ~ 8번의 연산순환주기(Cycles)에서 결과 값을 구할 수 있으나, 나눗셈 연산은 최하 8 ~ 60번 이상의 연산순환주기에서 결과 값이 발생한다. 그리고 사용 빈도수가 다른 알고리즘에 비해 적다는 이유로 그 동안 고속 나눗셈의 하드웨어의 연구가

* 朝鮮大學校 電子計算學科
(Dept. of Computer Science, Chosun Univ.)

※ 본 연구는 97년도 교육부 반도체분야 학술연구 조성비(ISRC 97-E-2035)에 의하여 연구되었음.

接受日: 1999年7月9日, 修正完了日: 1999年11月22日

활발하지는 않았다. 그러나 멀티미디어의 발달에 따라 고속의 FPU(Floating pointer unit)의 필요성이 증가하였고, 이에 따라서 나눗셈 연산이 차지하는 비중이 점점 더 커지게 되었다. 그리고 나눗셈 연산은 빈도수가 적지만 단위 연산 수행 시간이 길어서 전체 수행 시간의 관점에서는 성능에 영향을 미치는 중요한 연산이다[1][2].

나눗셈 연산은 높은 자릿수(High-Radix)를 기본 자릿수로 이용함으로써 매 연산에서 생성하는 몫 비트수를 증가할 수 있어서 전체연산에서 필요로 하는 연산순환주기를 줄일 수 있다. 높은 자릿수 고속나눗셈 연산기에서 전체적인 연산 속도는 수행시간이 가장 긴 구간의 시간(Critical path time)과 연산순환주기에 의하여 결정된다. 본 논문에서는 연산순환주기를 감소하는 방법으로서 영역변환상수를 연산기를 이용하여 구하는 방법 대신에 검색테이블로 구성하여 영역변환상수를 구하는 방법을 논하고자 한다. 영역변환상수를 직접 검색테이블에 저장하려면 검색테이블의 크기가 너무 커진다는 문제가 있기 때문에, 크기를 줄이는 방법으로 영역변환상수의 범위를 분석하여 일차적으로 크기를 줄이고, 두 개의 검색테이블을 이용하여서 이차적으로 크기를 줄이는 방법을 제시하였다[3]-[5].

본 논문의 구성은 2장에서 높은 자릿수 나눗셈 알고리즘을 소개하고, 3장에서는 영역변환상수의 범위를 분석하여 영역변환상수 검색테이블(Pre-scaling look-up table) 구성과 검색테이블의 크기를 줄이는 두 개의 검색테이블 설계에 관하여 설명하고, 4장에서는 검색테이블을 이용한 나눗셈 연산기의 성능평가를 하였다. 마지막으로 5장에서는 결론 및 향후 연구 방향에 대해서 논한다.

II. 높은 자릿수 나눗셈

2.1 알고리즘

높은 자릿수(Very-high radix) 나눗셈 알고리즘은 매 자릿수 순환마다 여러 개의 몫 비트를 생성한다. 이 방법은 전체의 연산순환주기가 감소하기 때문에 빠른 연산 속도를 얻을 수 있으나, 몫 선택 함수가 복잡해지는 문제가 발생하여, 이 문제가 성능 향상의 걸림돌이 된다. 이러한 문제를 해결하기 위하여 제수와 피제수를 영역 변환하여 몫 선택 함수를 단순하게 하는 방법을 사용한다[4],[6]-[9].

나눗셈 $q = x/d$에서 자릿수가 r일 때, 생성되는 몫의 비트 길이는 $b\,(\log_2 r = b)$이며, 제수 d, 피제수 x 그리고 몫 q의 범위는 식(1)과 같으며, 영역변환상수 M을 이용하여 제수와 피제수의 영역을 식(2)와 같이 변환한 후에 식(3)을 반복함으로서 몫을 생성한다.

$$\frac{1}{2} \le x < d < 1,\ \frac{1}{2} \le q < 1 \tag{1}$$

$$z = Md,\ w[0] = Mx \tag{2}$$

$$w[j+1] = rw[j] - q_{j+1}z,\ j = 0, 1, \cdots, \left\lceil \frac{n}{b} \right\rceil \tag{3}$$

여기서, n은 x의 입력비트 길이

식(3)에서 중간에 생성된 몫 q_{j+1}은 식(4)에 의하여 구하여지며, 몫 선택은 나머지 값에서 반올림한 값에서 몫을 얻는다.

$$q_{i+1} = \left\lfloor \hat{y} + \frac{1}{2} \right\rfloor \quad (-r+1 < q < r-1) \tag{4}$$

여기서, $\hat{y} = \{rw[j]\}_2$ (소수점 두 번째 비트까지만 남기고 나머지 비트는 버림)

식(4)에서 제시된 몫 선택 함수는 매우 간단한데, 이는 제수와 피제수의 영역을 변환하였기 때문에 가능하며, 이를 위한 영역변환상수 M이 필요하다[9]. 영역변환상수 M은 식(5)를 이용하여 구하며, 영역변환상수를 구하기 위한 γ_1, γ_2는 식(6), 식(7)를 이용하여 구한다. 여기서 필요한 비트의 길이는 M은 2비트의 정수와 $(b+4)$의 소수 비트(총 $(b+6)$비트)가 필요하며, γ_1은 $(b+6)$비트, γ_2는 $(b+5)$비트가 필요하다.

$$M = -\widehat{\gamma}_1 d_h + \widehat{\gamma}_2 \quad (5)$$

여기서, d_h는 입력된 제수 d에서 $h(=b+5)$비트 값

$$\gamma_1 = \frac{1}{d_\tau^2 + d_\tau 2^{-\tau} + 2^{-(b+4+2)}} \quad (6)$$

$$\gamma_2 = \frac{2d_\tau + 2^{-\tau}}{d_\tau^2 + d_\tau 2^{-\tau} + 2^{-(b+4+2)}} \quad (7)$$

여기서, $\tau = \lfloor b/2 \rfloor + 2$

높은 자릿수 나눗셈 연산기는 고정된 연산 시간을 위하여 캐리 저장형(Carry-Save Form)으로 값을 저장하며, 연산 또한 캐리 저장 가산기(Carry-Save Adder; CSA)를 사용한다. 또한 몫 q_j를 구하기 위해서는 곱셈 연산이 필요하며 이에 따라서 늘어나는 시간을 줄이기 위해서 q_j를 새로운 형태로 변환(Re-coding)하여 Radix-4로 표현된 자릿수를 이용한다[9]. 모든 연산을 마친 후에는 캐리 전달 가산기(Carry Propagate Adder; CPA)를 이용하여 캐리(Carry)와 합(Sum)을 더하고, 반올림과 On-The-Fly 알고리즘을 이용하여 최종적인 몫을 구한다[10],[11].

전체적인 연산순환주기는 영역변환상수인 M을 구하는 1주기, 영역 변환된 제수와 피제수를 구하는 2주기, 누적 연산기에서 몫을 구하는 $\lceil n/b \rceil$ 주기 및 마지막 반올림과 On-the-fly를 수행하는 1주기가 필요하다. 다음 식(8)은 알고리즘 수행을 위한 전체적인 연산순환주기이며, 연산기의 블록도는 그림 1과 같다.

$$N_{cycles} = \left\lceil \frac{n}{b} \right\rceil + 4 \quad (8)$$

2.2 감마 검색테이블

영역변환상수를 계산하는데 필요한 γ_1과 γ_2는 일반적으로 연산기를 이용하여 계산하지 않고, 검색테이블(Look-up table)을 이용하여 직접 값을 구한다. 그리고 감마 테이블의 크기를 보면 식(9)와 같다. 여기서 제수의 범위는 식(1)처럼 0.5이상 1미만이고 제수는 항상 2^{-1} 값을 갖고 있기 때문에 입/출력 비트의 범위가 줄어든다.

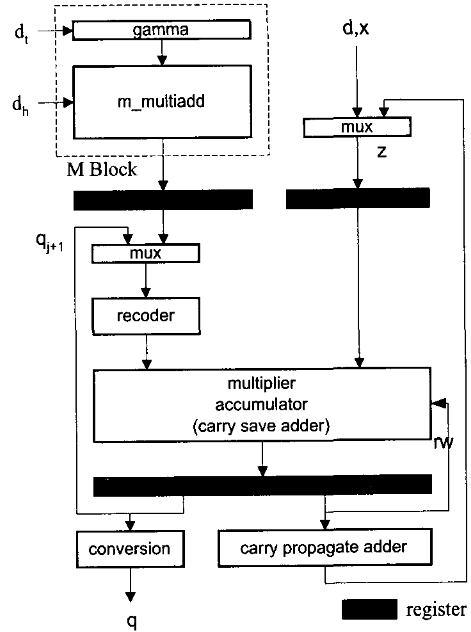


그림 1. 고속 나눗셈 연산기 블록도
Fig. 1. Block diagram of very-high radix divider.

$$\text{Gamma_Table} = 2^{\tau-1} \times (2b+11) \text{ (Bit)} \quad (9)$$

III. 영역변환상수 검색테이블

3.1 영역변환상수 범위

영역변환상수는 몫을 식(4)처럼 쉽게 구하기 위하여 제수와 피제수의 영역을 변환하는데 필요한 상수이다[7]. 여기서 영역변환상수의 입력비트 길이는 2b+5이고 출력은 두 자리의 정수와 (b + 4)의 실수로 출력된다[9]. 여기서 2-1은 항상 존재함으로 생략이 가능하며, 출력되는 M의 범위는 $1 \le M < 2$이고 정수를 표현하는 비트(정수자리 2비트)가 항상 동일하여 생략이 가능하다. 따라서 영역변환상수를 검색테이블로 구성하면 크기는 식(10)과 같다.

$$\text{M_Table} = 2^{b+4} \times (b+4) \text{ (Bit)} \quad (10)$$

그러나 입력비트의 길이가 길기 때문에 영역변환상수를 바로 검색테이블로 구현하게 되면 크기가 전체의 나눗셈 연산기 크기보다 크고, 합성(Synthesis)조차 불가능하게 된다. 따라서 크기가 작은 검색테이블을 구현하기 위한 방법으로 영역 변환된 제수의 범위를 분석하였다.

먼저 영역 변환된 제수의 범위를 보면 식(11)과 같으며, 프로그램을 이용하여 비트의 범위를 분석하면 식(12)와 같다. 또한 식(1)에서 제수의 입력 범위를 자세히 살펴보면 식(13)과 같다.

$$1-\frac{r-2}{4r(r-1)} < z < 1+\frac{r-2}{4r(r-1)} \tag{11}$$

$$\sum_{i=1}^{b+2} 2^{-i} < z < 1+2^{-(b+2)} \tag{12}$$

$$d_{\min} = 0.5,\ d_{\max} = 1-ulp \tag{13}$$

(Unit in the Last Place)

영역변환상수 M은 $z = Md$의 관계식으로 구해지므로, 식(12)와 식(13)을 이용하여 영역변환상수인 M의 범위를 구하면 식(14)와 같다. 또한, M의 정수 비트 범위 값은 {0 ~ 2}이기 때문에 정수를 표현하는데 2비트가 필요하나, M의 최소 값과 최대 값의 범위가 식(15)를 만족하기 때문에 M 범위를 식(16)과 같이 고정하여도 영역범위가 조정된 z의 값은 식(12)를 만족한다.

$$z_{\min}/d_{\max} \le M \le z_{\max}/d_{\min} \tag{14}$$

$$z_{\min}/d_{\max} \le 1\ ,\ \ 2 \le z_{\max}/d_{\min} \tag{15}$$

$$1 \le M < 2 \tag{16}$$

3.2 검색테이블의 크기를 줄이기 위한 영역 축소 방법

영역변환상수를 검색테이블을 구성하는 알고리즘의 기본적인 원리는 영역 변환된 제수 z가 식(12)의 조건을 만족하는 M을 구하는 것이다. 따라서 식(2)에서 영역 변환된 제수 z를 구하는 식을 이용하면 식(17)과 같이 구할 수 있다.

$$M \approx \widetilde{M} = \frac{\tilde{z}}{d} \tag{17}$$

다음은 d_{i-1}부터 d_i까지 식(12)를 만족하는 영역변환상수를 구한다. 이러한 방법으로 d_i에서 구한 영역변환상수의 최소 값을 d_{i-1}부터 d_i까지의 영역변환상수로 이용한다. 여기서 영역변환상수는 제수의 값에 따라 감소하고, d_{i-1}와 d_i사이의 제수가 쉽게 영역변환상수 값을 얻을 수 있으므로 d_{i-1}를 범위의 최소 값으로 사용한다. d_i에 대한 영역변환상수 $\widetilde{M}_i$는 식(18)과 같이 구한다.

$$\widetilde{M}_i = \frac{z_{\min}}{d_{i-1}} \tag{18}$$

제수의 입력비트 길이에 따라 영역변환상수의 값 역시 달라지게 된다. 또한 검색테이블의 크기 역시 입력비트의 길이에 따라 크기가 크게 달라지므로, 최소한의 입력비트 길이를 찾아야 크기가 작은 검색테이블을 만들 수 있다. 제수의 입력비트 길이를 f라 할 때, 영역변환상수는 식(19)와 같다.

$$\widetilde{M}_i = \frac{z_{\min}}{d_{i-1} - \sum_{i=f+1}^{n} \delta_i 2^{-i}} \tag{19}$$

$$\text{여기서}\ \delta_i \in \{0,1\}$$

식(19)를 이용하여 얻은 영역변환상수는 식(12)의 조건을 만족해야 한다. 또한 입력비트의 길이가 f비트인 제수를 이용하여 $\tilde{z}_i$를 구한다. 이 $\tilde{z}_i$는 식(20)으로 구하며, $\tilde{z}_i$는 $z_{\min}$과 $z_{\max}$ 사이에 존재해야 하기에 식(21)을 만족해야 한다.

$$\tilde{z}_i = \left(\frac{z_{\min}}{d_{i-1} - \sum_{i=f+1}^{n} \delta_i 2^{-i}} \right) \left(d_i - \sum_{i=f+1}^{n} \delta_i 2^{-i} \right) \quad (20)$$

$$z_{\min} < \tilde{z}_i < z_{\max} \quad (21)$$

f의 길이에 따라 $\tilde{z}_i$는 식(21)의 범위를 만족하거나, 벗어나게 된다. 즉, 제수의 입력비트 길이가 짧을 경우, 식(21)의 범위를 벗어나게 되나, f의 값을 점점 증가하여 입력비트의 길이를 길게 하면 식(21)의 범위를 만족하게 된다. 따라서 식(21)의 범위를 만족하는 최소의 입력비트 길이인 f를 찾는다. 그림 2는 입력비트의 길이에 따른 영역변환상수 $\widetilde{M}$과 영역 변환된 z의 관계를 나타낸다.

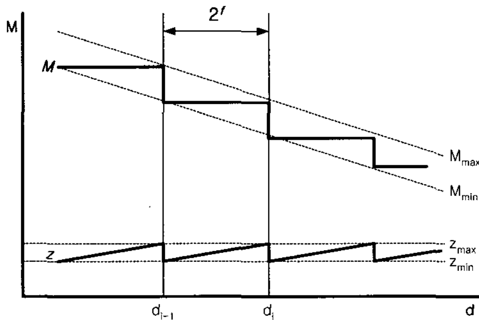


그림 2. 입력비트의 길이에 따른 M과 z의 관계
Fig. 2. Related of M and z with input bit width.

영역변환상수 값의 비트 길이 역시 입력비트의 길이를 구하는 방법과 동일하다. 즉, 식(20)을 계산하면서 $\widetilde{M}_i$의 비트 길이에 따라 식(21)의 범위를 만족하거나, 벗어나는 현상이 발생한다. 다라서 식(21)의 범위를 만족하는 최소의 영역변환상수 값의 비트 길이인 g를 구한다. f와 g 값을 프로그램을 이용하여 구하면 식(22)와 같다.

$$f = b+2, \ g = b+3 \quad (22)$$

최종적인 영역변환상수는 식(23)으로 구하며, 식(16)의 조건에 따라 $\widetilde{M}_{\min}$은 식(24)가 되며, 전체적인 검색테이블의 크기는 식(25)와 같다. 여기서 입력비트는 항상 0.5(2-1)를 유지하고, 출력 비트는 정수를 표현하는 비트(정수자리 2비트)가 항상 동일하여 생략이 가능하다.

$$\widetilde{M} = \frac{z_{\min}}{d_{i-1} - \sum_{i=f+1}^{n} \delta_i 2^{-i}} - \sum_{i=g+1}^{n} \delta_i 2^{-i} \quad (23)$$

$$\widetilde{M}_{\min} = 1 \quad (24)$$

$$M'_Table = 2^{b+1} \times (b+2) \ (Bit) \quad (25)$$

3.3 캐리 저장형을 이용한 검색테이블

고속 나눗셈 연산기는 연산순환주기의 반복에서 연산기의 속도로 인한 시간의 지연을 방지하기 위하여 캐리 저장 연산기(Carry Save Adder)를 이용한다. 그래서 데이터 흐름 역시 캐리 저장형(Carry Save Form)으로 저장한다. 즉, 데이터 전송을 합(Sum)과 캐리(Carry)로 분리하여 저장하고 연산을 한다. 여기서 M'_Table 역시 캐리 저장형을 이용하여 2개의 검색테이블로 분리하여서 2개의 출력 값을 갖는 방법을 이용할 수 있다. 즉, 최소한의 입력비트로 $\widetilde{M}$의 근사값을 찾은 다음, 이에 대한 수정 값을 독립적으로 출력한 후, 나중에 서로의 값을 더하여서 본래의 $\widetilde{M}$을 구하는 방법이다. 이 방법에서는 최소한의 입력비트로 얻은 M_1과 $\widetilde{M}$과 M_1의 수정 값인 M_2를 각각의 검색테이블로 구성하여야 한다. 이렇게 한 이유는 입력비트의 길이가 짧을수록 작은 크기의 검색테이블을 만들 수 있기 때문이다.

식(23)처럼 $\widetilde{M}$은 제수의 입력비트 길이가 f일 때 발생한다. 여기서 입력비트의 길이를 더 짧게 하면 검색테이블의 크기는 작아지지만 식(12)를 만족하지 못하게 된다. 그래서 더 작은 크기를 갖는 검색테이블을 구하기 위해서는 입력비트의 길이가 더 짧으면서 식(12)를 만족하는 방법을 찾아야 한다.

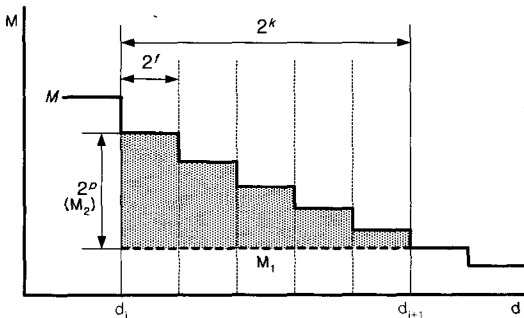


그림 4. 입력비트의 길이가 f와 k일 때의 M과 M1 관계
Fig. 4. Related of M and M1 with input bit width of f and k.

그림 4에서 입력비트의 길이가 $(f > k)$일 때, $\widehat{M}$은 입력비트의 길이가 f일 때의 영역변환상수이고, M_1은 입력비트의 길이가 k일 때의 영역변환상수이다. 여기서 입력비트의 길이가 k일 때의 값인 M_1을 구한 다음에, $\widehat{M}$과 M_1의 수정 값인 M_2를 구하여 각각의 검색테이블에 저장한다. 그리고 $\widehat{M}$을 구할 때에는 M_1과 M_2를 더하면 쉽게 구할 수 있다. 두 개의 검색테이블은 식(26)과 같이 구한다.

$$M_1 = \frac{z_{\min}}{d_{i+1}}, \quad M_2 = \widehat{M} - M_1 \tag{26}$$

여기서 M_2의 입력비트의 길이는 $\widehat{M}$에 대한 M_1의 수정 값이기 때문에 f만큼의 길이가 필요하다. 이 방법의 문제는 M_2의 출력 비트 길이를 p라 할 때, M_1을 구하기 위해서 입력비트의 길이를 k만큼 줄이면 $\widehat{M}$과 M_1의 수정 값의 비트 길이인 p가 늘어나게 되며, 수정 값을 저장한 검색테이블의 크기 또한 증가하게 된다. 따라서 두 개의 검색테이블 크기의 합이 최소인 k와 p를 찾아야 한다. M_1과 M_2의 크기는 식(27)과 식(28)이며, 두 개의 검색테이블의 크기는 식(29)와 같다. 여기서 k와 p는 프로그램을 이용하여 구하였으며, 결과는 표 1과 같다.

$$\text{M1_Table} = 2^k \times (b+2) \ \text{(Bit)} \tag{27}$$

$$\text{여기서} \ (2 < k < b+1)$$

$$\text{M2_Table} = 2^f \times p \ \text{(Bit)} \tag{28}$$

$$\text{여기서} \ p = \lceil \log_2(\widehat{M} - M_1) 2^g \rceil$$

$$\text{M_Table(M1+M2)} = \text{M1_Table} + \text{M2_Table (Bit)} \tag{29}$$

표 1. 각 자릿수에 대한 M_Table의 k와 p
Table 1. k and p of M_Table with radix.

b	4	5	6	7	8	9	10	11	12	13	14	15
k	2	3	3,4	4	5	6	7	8	9	10	10,11	12
p	5	5	6,5	6	6	6	6	6	6	6	6,7	7

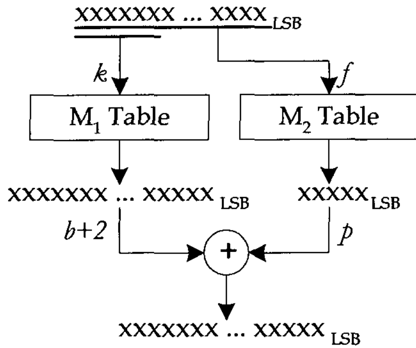
여기서 $\widehat{M}$을 구하는 데에 있어서 M_1과 M_2의 합을 구하는 가산기가 필요하나, 캐리 저장형에서는 두 개의 데이터 흐름이 있으므로, 각 테이블의 값을 합과 캐리선에 전송만 하면 된다. 따라서 $\widehat{M}$을 구하기 위한 특별한 연산기가 필요 없으면서 검색테이블의 크기 또한 줄일 수 있다.

$$\text{Sum} = M_1, \quad \text{Carry} = M_2 \tag{30}$$

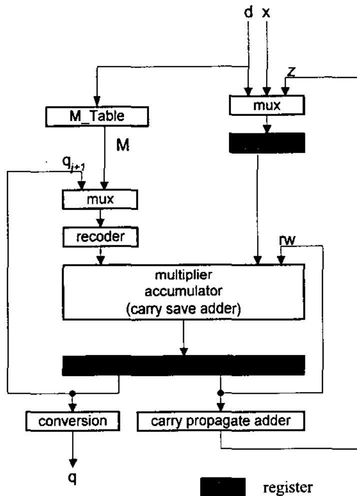
두 개의 검색테이블 블록도는 그림 5의 (a)와 같으며, 검색테이블을 이용한 고속나눗셈 연산기의 블록도는 그림 5의 (b)와 같다.

IV. 고속 나눗셈 연산기의 구현 및 성능 평가

기본자릿수를 512로 하여, 기본자릿수가 같은 나눗셈 연산기와 연산순환주기가 같은 나눗셈 연산기를 구현 및 평가하였다. 구현 환경을 보면 각 검색테이블은 UltraSPARC1, Solaris 2.5.1, WORKSHOPTM SPARCompiler C++4.1을 이용하여 검색테이블 VHDL 코드 생성기를 만들었으며, Synopsys 98.02와 Compass v8r4.10(VTI cmn12 (1.2㎛) 라이브러리)를 이용하여 합성, 동작 확인 및 속도와 크기를 구하였다.



(a) M_Table(M1 + M2)의 블록도



(b) 검색테이블을 이용한 고속 나눗셈 연산기 블록도

그림 5. M_Table(M1+M2)과 검색테이블을 이용한 고속 나눗셈 연산기 블록도
Fig. 5. Block diagram of very-high radix divider with pre-scaling look-up table.

4.1 검색테이블의 성능 평가

식(9), 식(10), 식(25), 식(29)를 이용하여 각각의 검색테이블 크기를 비교하면 표 2와 같다. 표 2에서 M_Table(M1 + M2)이 기타 다른 영역변환상수 검색테이블보다 작다는 것을 알 수 있다. 그리고 감마 검색테이블보다 약 1 ~ 25배 정도의 크기 차이가 있지만, 영역변환상수 검색테이블을 이용하면 연산순환주기가 감소하므로 연산순환주기가 같은 감마 검색테이블과 비교하면 약 0.5 ~ 2.7배정도의 차이만이 있다.

표 2. 각 검색테이블의 비교
Table 2. Comparison of look-up table.

Table \ b	Gamma_Table 크기(비트)	M_Table 비율(배)	M'_Table (영역분석 후) 비율(배)	M_Table (M_1+M_2) 비율(배)
4	152	13.5	1.3	0.7
5	168	27.4	2.7	1.3
6	368	27.8	2.8	1.2
7	400	56.3	5.8	2.3
8	864	56.9	5.9	2.1
9	928	114.8	12.1	4.1
11	2,112	232.7	25.2	7.4
14	9,984	472.6	52.5	13.1
18	48,128	1917.3	217.9	44.9
⋮	⋮	⋮	⋮	⋮

(비율 : 각 자릿수에 대한 Gamma_Table이 기준)

4.2 고속 나눗셈 연산기의 성능 평가

연산기의 성능 평가는 표3과 같다. 영역변환상수 검색테이블을 이용한 연산기는 기본 자릿수가 같은 나눗셈 연산기와 비교하여 크기는 약 1.12배, 속도는 0.92배 차이가 나지만, 같은 순환 주기의 연산기와 비교하면 크기는 약 0.9배, 속도는 0.91배 차이가 발생한다. 여기서 높은 자릿수 나눗셈 연산기의 비교는 순환주기를 기준으로 비교하기 때문에 전체적인 성능을 보면 크기는 약 0.9배 감소하면서 수행 시간은 약 0.9배 작아져 빨라졌음을 확인 할 수 있다. 특히 성능 향상을 위하여 사용해야 할 자릿수가 갑자기 커질 경우 더 큰 성능의 향상을 가져올 수 있다. 여기서 radix-2048의 경우 누적 연산기의 수행시간이 늘어나는 현상이 발생하므로 가장 긴 구간(Critical Path)의 시간이 증가하였다.

표 3. 각 고속 나눗셈의 성능 평가

Table 3. Performance analysis for radix divider.

	radix-512	radix-512 (Table M_1+M_2)	radix-2048
Area	15,479 (NAND2)	17,335 (NAND2)	19,348 (NAND2)
Cycles	10	9	9
Critical Path	53.58ns	54.61ns	60.00ns
Speed	535.8ns	491.49ns	540.09ns
Normalized Area	1	1.12	1.25
Normalized Speed	1	0.92	1.01

(기준: radix-512 나눗셈 연산기)

V. 결　론

지금까지 고속 나눗셈 연산기에서 최적화를 위하여 검색테이블 방식을 이용한 성능 향상을 논하였다. 연산기의 설계에서 언제나 신경을 써야 할 부분은 연산 속도와 크기에 대한 문제이다. 이 속도와 크기는 서로 상반된 관계를 갖고 있으므로 적절히 조절을 하면서 설계를 해야 한다. 본 논문은 높은 자릿수 나눗셈 연산기에서 주기를 감소하는 방법으로 영역변환상수를 검색테이블로 구현하는 방법을 사용하였다. 그리고 검색테이블 구현에서 문제가 된 크기는 영역변환상수의 분석 및 캐리 저장형을 이용한 두 개의 검색테이블을 설계함으로서 상당히 감소시킬 수 있음을 보였다. 그리고 자릿수가 같은 연산기와 연산순환주기가 같은 연산기와의 비교에서 자릿수가 같은 연산기에 비해 크기는 늘어났지만 연산 속도는 빨라지는 것을 확인하였고, 연산순환주기가 같은 연산기보다 크기는 줄어들면서 연산 속도는 빨라지는 것을 확인하였다. 그래서 전체적인 성능에서 크기는 약 0.9배 감소하면서, 연산시간 또한 약 0.9배 감소하므로, 전체적인 성능이 향상되었다.

본 연구의 결과는 고속 부동소수점 연산기 개발에 이용할 수 있으며, 멀티미디어 시스템을 위한 연산기 및 고성능의 그래픽 랜더링에 이용할 수 있으며, 나눗셈 연산이 시스템의 병목 현상이 되는 경우의 해결책으로 제시될 수 있다. 그리고 높은 자릿수 나눗셈에서 연산순환주기를 감소하기 위하여 기본 자릿수를 결정하는데 도움이 될 수 있다. 또한 검색테이블 방법에 있어서 캐리 저장형을 이용할 경우 두 개의 검색테이블을 이용함으로서 그 동안 크기 문제로 사용하기 힘든 검색테이블 방법을 좀 더 효율적으로 사용할 수 있었다.

향후, 각 자릿수에 대한 검색테이블의 정밀 분석과, 검색테이블의 최적화 문제, 그리고 고속 나눗셈 연산기에서의 크기와 속도에 대하여 구현한 후 보다 정밀한 분석에 관한 연구가 이루어져야 할 것이다.

참 고 문 헌

[1] David A. Patterson, John L. Hennessy, *Computer Architecture A Quantitative Approach*, Moragan Kaufmann Publishers, Inc., San Mateo, California, 1996.

[2] Stuart F. Oberman and Michael J. Flynn, "Design Issues in Division and Other Floating-Point Operations," *IEEE Trans. Compute.*, pp. 1-18, 1996.

[3] Israel Koren, *Computer Arithmetic Algorithm*, Prentice-Hall, Inc., 1993.

[4] M. D. Ergegovac, T. Lang, *DIVISION AND SQUARE ROOT*, Kluwer Academic Publishers, 1994.

[5] Stuart F. Oberman and Michael J. Flynn, "Division Algorithms and Implementations", *IEEE Trans. Compute.*, Vol. 46, No. 8, 1997.

[6] D. E. Atkins, "Higher-radix division using estimates of the divisor and partial remainders," *IEEE Trans. Comput.*, vol. C-17, pp. 925-934, 1968.

[7] M. D. Erecegovac and T. Lang, "A division algorithm with divisor scaling," *in Proc. IEEE 7th Symp. Comput. Arithmetic*, pp. 51-56, 1985.

[8] M. D. Ergegovac and T. Lang, *Division and Square*

Root: Digit-Recurrence Algorithm and Implementations, Kluwer Academic Publisher, 1st edition, 1994.

[9] M. D. Ergegovac, T. Lang and P.Montuschi, "Very-High Radix Division with Prescaling and Selection by Rounding," *IEEE Trans. Comput.*, vol. 43, pp. 909-918, 1994.

[10] M. D. Erecegovac and T. Lang, "On-the-fly conversion of redundant into conventional representations," *IEEE Trans. Comput*, vol. C-36, pp. 895-897, 1987.

[11] Peter Soderquist and Miriam Leeser, "Area and Performance Tradeoffs in Floating-Point Divide and Square-Root Implementations," *ACM Computing Surveys*, Vol. 28, No. 3, 1996.

[12] Standards Committee of the IEEE Computer Society, *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Standard 754-1985, 1985.

저 자 소 개

李 炳 錫 (學生會員)
1997년 2월 조선대학교 전자계산학과(학사), 1999년 8월 조선대학교 전자계산학과(석사), 현재 조선대학교 컴퓨터시스템실험실 인턴 연구원.
주관심 분야: Application-Specific Processor Design, Embedded System, Real-Time OS

李 禎 娥 (正會員)
1982년 2월 서울대학교 컴퓨터공학과(학사), 1985년 6월 미국 인디애나주립대학 컴퓨터학과(석사), 1990년 11월 미국UCLA 컴퓨터공학과(박사), 1990년 8월~1995년 2월 미국 휴스턴주립대학 전기전산공학과(조교수), 1993년 6월~1994년 4월 미국 국립초전도가속기연구소(객원연구원), 1995년 3월~현재 조선대학교 컴퓨터공학부(부교수).
주관심 분야: Computer Arithmetic, Application-Specific Processor Design, (Re)Configurable Computing