개선된 고속 제곱 발생기 설계

송 상 훈†

요 약

제곱 테이블을 이용한 곱셈 방법은 처리과정을 간단히 하고 속도도 향상시킨다. 그러나, 비트 길이가 증가함에 따라 테이블 크기는 지수 승으로 증가하게 된다. 최근에 Wey와 Shieh는 고속 곱셈이 요구되는 응용분야에 적합한 폴딩 기법을 이용한 우수한 제곱 발생기를 제안하였다. 이 기법은 ROM 주소에 대한 1의 보수 값을 이용하여 제곱 값을 위한 거대한 테이블을 계속 폴딩함으로써 필요한 테이블의 크기를 작게 만들어 ROM의 크기를 줄일 수 있도록 한다. 본 논문에서는 Wey와 Shieh의 기법에서 1의 보수 부분이 필요 없는 개선된 폴딩 기법을 제안한다. 그리고 제안된 방법은 중간 과정에서 필요한 부분 합의 비트 길이를 줄임으로써 하드웨어 구현을 쉽게 하고 성능을 더욱 향상시킨다.

Improved Design of a High-Speed Square Generator

Sang-Hoon Song†

ABSTRACT

The square-based multiplication using look-up table simplifies the process and speeds-up the operating speed. However, the look-up table size increases exponentially as bit size increases. Recently, Wey and Shieh introduced a noble design of square generator circuit using a folding approach for high-speed performance applications. The design uses the ones complement values of ROM addresses to fold the huge look-up ROM table repeatedly such that a much smaller table can be sufficient to store the squares. We present new folding techniques that do not require a ones complement part, one of three major parts in the Wey and Shiehs method. Also the proposed techniques reduce the bit size of partial sums such that the hardware implementation be simplified and the performance be enhanced.

1. 서 론

고속 연산은 디지털 신호처리, 영상처리 등 많은 응용분야에 필요한 기술이다. 특히, 곱셈 연산은 시스템 성능을 좌우하는 가장 중요한 연산중의 하나이다. 고속 곱셈을 위하여 연산이 필요 없는 테이블 검색 방법을 이용하는 것은 이미 잘 알려져 있다[1, 2, 3, 8]. 테이블을 이용한 곱셈 방법에서 메모리 용량을 줄이기 위하여 Ling[4]은 한 개의 데이터에 기반한 변환 방법을 제안하여 필요한 메모리 용량을 2^{2b}에서 2^b로 줄였다. 여기서 b는 데이터 비트 길이 이다. 또 다른 방법으로는 제곱 테이블을 이용하는 방법이 있다. 즉, 입력 범위에 있는 모든 수에 대한 제곱들을 단일 테이블에 저장하여 두고, 곱할 두수로부터 단일 테이블에 대한 두 개의 인덱스를 생성하여 이에 대한 제곱 값들을 테이블로부터 찾고 간단한 연산을 통하여 두수의 곱을 구한다. 곱할 두수를 A와 B라고 하고, $x = (A+B)/2$, 그리고 $y = (A-B)/2$라고 할 때 A*B는 다음과 같이 구할 수 있다. ⌈⌉는 천정(ceiling) 함수를 뜻한다.

$$A*B = x^2 - y^2, \quad A+B \text{ 가 짝수 일 때}$$
$$A*B = x^2 - y^2 + B, \quad A+B \text{ 가 홀수 일 때} \qquad (1)$$

† 정 회 원 : 세종대학교 컴퓨터공학과 교수
논문접수 : 1999년 5월 24일, 심사완료 : 2000년 1월 4일

위 식은 비트 길이가 n인 수에 대한 제곱 테이블을 이용하여 비트 길이가 n인 두수의 곱셈을 할 수 있음을 보여 준다. 이와 같이 제곱 테이블에 기반한 곱셈 방식은 간단한 덧셈, 뺄셈, 그리고 테이블 참조 작업만으로 이루어져 고속 연산을 가능하게 한다. 그러나 필요한 제곱 테이블의 크기는 $2^n * 2n$ 만큼 요구되어 비트 길이가 증가함에 따라 급격하게 커진다.

Vinnakota[5]는 제곱 테이블을 이용하는 방법에서 테이블의 크기를 줄일 수 있는 분리된 ROM 방법을 제안하였다. 즉, 테이블에 중복자료를 없애고 여러 개의 ROM에 분산시켜 용량을 줄이는 것이다. 제곱 값은 각 ROM 출력들을 연결된 비트로 간주하여 얻으면 된다. 메모리 크기를 최소화하기 위하여 n 개의 블록으로 나누게 되는데, n 비트 크기의 수에 대한 필요한 테이블의 크기가 $2^n * (n+1) - 4$만큼으로 줄어 든다.

최근에 테이블 크기를 줄이기 위한 폴딩 기법이 제안되었다[9]. 폴딩 기법은 테이블을 반으로 접어서 MSB가 0일 때의 테이블만을 가지고 MSB가 1일 때의 제곱 값을 구하도록 하는 것이다. MSB가 0일 때는 직접 테이블에서 제곱 값을 구하면 된다. MSB가 1일 때는 테이블 주소에 대한 1의 보수 값을 취하여 이에 대한 제곱 값을 테이블에서 구하고, 구하려는 제곱 값과의 차(보정 값)를 주소로부터 구하여 합을 만들어 구하게 된다. 즉, 보정 값을 구하기 위한 하드웨어가 여분으로 추가된다. 이 기법은 반으로 줄어든 테이블에 대하여도 회귀적으로 적용하여 테이블의 크기를 계속 반으로 줄여나가 가용 가능한 메모리에 저장할 만한 크기까지 줄일 수 있다. 물론 보정 값을 위한 하드웨어 복잡성은 커지게 된다.

본 논문에서는 폴딩 기법에서 필요한 1의 보수를 취하는 부분을 없애고, 보정 값의 데이터 비트 길이를 줄여 하드웨어 복잡성을 줄이고 지연시간도 감소시킬 수 있는 개선된 폴딩 기법을 제안한다.

본 논문의 구성은 다음과 같다. 2절에서 폴딩 기법에 대한 내용을 기술한다. 3절에서는 폴딩 기법에 대하여 개선된 방법을 제안하여 기존의 폴딩 기법과 비교하고, 4절에서 결론을 맺는다.

2. 폴딩 기법

이 절에서는 Wey 와 Shieh가 제안한 폴딩 기법[9]에 대하여 설명한다. 제곱 연산을 위한 테이블은 너무 크기가 커서 실용적이지 않으므로, 테이블을 반으로 접어서 MSB가 0 일 때의 테이블만을 가지고 MSB가 1 일 때의 제곱 값을 구하도록 하는 것이다. MSB가 0 일 때는 직접 테이블에서 제곱 값을 구하고, MSB가 1 일 때는 테이블 주소에 대한 1의 보수 값을 취하여 이에 대한 제곱 값을 테이블에서 구하고 주어진 수를 이용하여 계산된 보정 값 값과 합하여 구하게 된다. 폴딩 기법은 다음과 같은 수학적 특성에 기반하여 제안되었다.

특성 1. 두 2 진수 $A = (a_{n-1}a_{n-2}...a_1a_0)$와 $B = (b_{n-1}b_{n-2}...b_1b_0)$에서 B는 A에 대한 1의 보수와 같을 때, 즉 $b_i = 1-a_i$, $0 \le i \le n-1$, 두 이진수 제곱의 차는 다음과 같다.

$$D = |A|^2 - |B|^2 = -b_{n-1}\,2^n\,(2^n+1) + |(a_{n-2}...a_1a_0\bar{a}_{n-2}...\bar{a}_1\bar{a}_01)| \tag{2}$$

위 특성에 대한 증명은 [9]에 나와 있다. 위에서 a_{n-1}이 1인 경우에 $b_{n-1} = 1 - a_{n-1} = 0$이므로

$$D = |A|^2 - |B|^2 = (a_{n-2}...a_1a_0\bar{a}_{n-2}...\bar{a}_1\bar{a}_01) \tag{3}$$

특성 2. 주어진 n 비트 수 $N = (a_{n-1}a_{n-2}...a_1a_0)$에 대하여 다음과 같이 제곱 값을 구할 수 있다.

$$N^2 = (N^*)^2 + a_{n-1}*(a_{n-2}...a_1a_0\bar{a}_{n-2}...\bar{a}_1\bar{a}_01), \tag{4}$$

여기서 $N^* = (a_{n-2}...a_1a_0)$ if $a_{n-1} = 0$,

$N^* = (\bar{a}_{n-2}...\bar{a}_1\bar{a}_0)$ if $a_{n-1} = 1$.

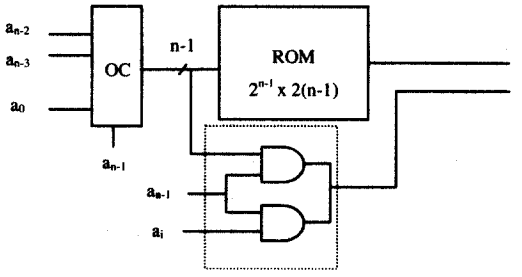
증명. $a_{n-1} = 0$인 경우, $N = N^*$ 이다. $a_{n-1} = 1$인 경우, N^*가 N에 대한 1의 보수이다. 즉, $N^* = (\bar{a}_{n-2}...\bar{a}_1\bar{a}_0)$이므로 식 (3)로 부터 $N^2 - (N^*)^2 = (a_{n-2}...a_1a_0\bar{a}_{n-2}...\bar{a}_1\bar{a}_01)$이다. □

예를 들면, 5 비트 수 $N = (a_4\ a_3\ a_2\ a_1\ a_0) = (\ 10011\ ) = 19$에 대하여 N^2을 다음과 같이 얻을 수 있다.

$$\begin{array}{ll} & 1\,0\,0\,1\,0\,0\,0\,0 \quad (\bar{a}_3\bar{a}_2\bar{a}_1\bar{a}_0)^2 = (1100)^2 \\ +\ & 0\,0\,1\,1\,0\,1\,1\,0\,0\,1 \quad (a_3a_2a_1a_0\bar{a}_3\bar{a}_2\bar{a}_1\bar{a}_01) \\ \hline & 0\,1\,0\,1\,1\,0\,1\,0\,0\,1 \quad (361) \end{array}$$

(그림 1)은 폴딩 기법에 대한 기본 구조를 보여준다. a_{n-1}이 0인 경우에 OC 부분은 $(a_{n-2}...\ a_1a_0)$을 그대로 통과시켜 ROM 테이블의 주소로 사용하고, 보정 값을 발생하는 부분에서는 0 값을 전달한다. $a_{n-1} = 1$인 경우는 OC 부분에서 $(\bar{a}_{n-2}...\ \bar{a}_1\bar{a}_0)$을 만들어 ROM 테이블 주소로 사용하고, 보정 값을 발생하는 부분에서는 $(a_{n-2}...a_1a_0\bar{a}_{n-2}...\bar{a}_1\bar{a}_01)$

을 발생시킨다. 곱셈 연산 응용에서 ROM 출력과 보정 값은 섬과 캐리의 짝으로 생각하여, 식 (1)에 의하여 다른 제곱 발생기의 출력과 더해지게 된다. ROM 테이블 크기를 더욱 줄이기 위하여 식 (4)를 다음과 같이 일반화 할 수 있다.



(그림 1) 폴딩 기법 구조

특성 3 (2^k 폴딩). 주어진 n 비트 수 $N = (a_{n-1}a_{n-2}...a_1a_0)$에 대하여 다음과 같이 제곱 값을 구할 수 있다.

$$N^2 = (N^*)^2 + D, \quad (5)$$

여기서,

$$N^* = (a_{n-k-1}...a_1a_0) \text{ if } a_{n-k} = 0$$

그리고

$$N^* = (\bar{a}_{n-k-1}...\bar{a}_1\bar{a}_0) \text{ if } a_{n-k} = 1$$

그리고 $D = (d_{2n-1}d_{2n-2}...d_{n-k+2}|d_{n-k+1}...d_1d_0) = (D_1 \mid D_2)$;

$D_1 = a_{n-k}\triangle_k + ... + a_{n-2}\triangle_2 + a_{n-1}\triangle_1$, 그리고 $D_2 = a_{n-k}\triangle_0$,

여기서,

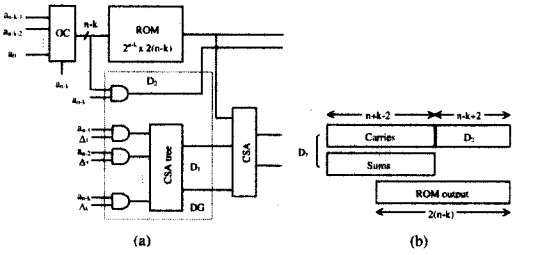
$\triangle_k$ = (0 0..... 0 a_{n-k-1} a_{n-k-2}.....a_1 a_0)
$\triangle_{k-1}$ = (0 0.....0 a_{n-k} $\bar{a}_{n-k}$ a_{n-k-1}.....a_1 a_0)
$\triangle_{k-2}$ = (0 0.....0 a_{n-k+1} $\bar{a}_{n-k-1}$ a_{n-k} a_{n-k-1}..... a_1 a_0 0)
...
$\triangle_2$ = (0 0 a_{n-3} $\bar{a}_{n-3}$ a_{n-4}.....a_1 a_0 0 0.....0)
$\triangle_1$ = (a_{n-2} $\bar{a}_{n-2}$ a_{n-3} .....a_1 a_0 0 0.....0)

그리고

$\triangle_0$ = (0 $\bar{a}_{n-k-1}$ ... $\bar{a}_1$ $\bar{a}_0$ 1)

위에서 $D_1|D_2$는 두개의 데이터가, D_1과 D_2, 연결된 것을 뜻한다. 이 특성에 대한 증명은 [9]에 있다. (그림 2a)는 2^k-폴딩에 대한 일반화된 구조를 보여 주는데 3개의 부분으로 구성되어 있다.

1) 1의 보수화 부분(OC)
2) ROM 테이블
3) D-값 발생기(DG).



(그림 2) 2k-폴딩 기법 구조 (a) 구성도 (b) 데이터 길이

(그림 2)의 OC 부분은 a_{n-k}에 의하여 $(a_{n-k-1}...a_1a_0)$을 그대로 통과 시키거나 1의 보수 값을 만들어 ROM 테이블 주소 N^*-값으로 사용하는데 *(n-k)*개의 XOR 게이트(gate)로 이루어 진다. 그리고 a_{n-k}에 의하여 보정 값의 D_2 부분이 *(n-k+2)* 비트의 0 또는 $(0\bar{a}_{n-k-1}...\bar{a}_01)$로 발생된다. 보정 값의 D_1 $(a_{n-k}\triangle_k + ... + a_{n-2}\triangle_2 + a_{n-1}\triangle_1)$ 부분은 CSA 트리를 통하여 값이 더해져서 *(n+k-2)* 캐리 비트와 *(n+k-2)* 섬 비트의 짝으로 만들어 진다. *2(n-k)* 비트의 ROM 출력은 *2n* 비트 길이의 D와 더해지는데, D_1이 *(n-k+2)* 비트 위치에서 시작하기 때문에 ROM 출력의 상위 *(n-k-2)* 비트들은 D_1과 CSA(carry save adder)에서 더해진다. ROM 출력의 나머지 하위 *(n-k+2)* 비트들과 *(n-k+2)* 비트의 D_2는 CSA 출력처럼 캐리와 섬 비트의 짝으로 생각할 수 있다. 제곱발생기의 성능은 신호 지연시간에 의해 나타낼 수 있는데, 제곱 발생기의 지연시간은 다음과 같이 나타낼 수 있다.

$$Pd = Pd_{xor} + \max(Pd_{ROM}, Pd_{DG}) + Pd_{CSA} \quad (6)$$

여기서 Pd_{xor}, Pd_{ROM}, Pd_{DG}, 그리고 Pd_{CSA}들은 XOR 게이트, ROM, DG, 그리고 CSA의 지연시간을 뜻한다. 실제 구현시 성능을 최대화하기 위하여 DG의 지연시간이 $Pd_{DG} < Pd_{ROM}$가 되도록 만든다. 즉, ROM의 크기를 줄이기 위하여 추가된 DG부분 때문에 전체 성능이 감소되지 않도록 하기 위함이다. ROM의 크기가 줄어 Pd_{ROM}도 작아지므로 전체 지연시간은 줄어들게 된다.

$$Pd = Pd_{xor} + Pd_{ROM} + Pd_{CSA} \quad (7)$$

DG의 지연시간에 대한 조건, $Pd_{DG} < Pd_{ROM}$,을 만족

시키기 위하여 CSA트리의 최대 레벨 수는, L_{max}, 다음과 같이 제한된다.

$$L_{max} \le \lfloor (Pd_{ROM} - Pd_{AND}) / Pd_{CSA} \rfloor \quad (8)$$

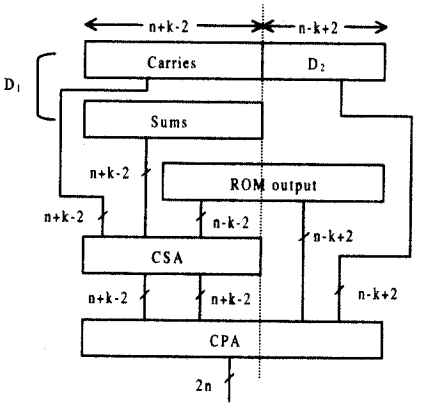
CSA의 입력 데이터의 수(k)에 의하여 CSA트리 레벨의 수가 결정되는데 이는 <표 1>를 참조하면 된다.

〈표1〉 CSA 트리 레벨 [1]

입력 데이터 수(k)	레벨 수
3	1
4	2
$5 \le k \le 6$	3
$7 \le k \le 9$	4
$10 \le k \le 13$	5
$14 \le k \le 19$	6
$20 \le k \le 28$	7

예를 들어 2^{10}*20 비트의 ROM, AND 게이트, 전가산기(CSA 기본 소자)의 지연시간이 각각 5.96ns, 0.9ns, 그리고 1.2ns일 때[9], 식 (8)로 부터 L_{max}는 4가 된다. <표 1>로부터 k_{max}는 9가 된다는 것을 알 수 있다. 이 경우에 19(10+9) 비트의 데이터에 대한 제곱 발생기를 만들 수 있다.

제곱 발생기의 출력은 2n비트의 캐리와 섬을 만들어 내는데, 곱셈 연산 응용에서는 식 (1)에서 처럼 다른 제곱 발생기의 출력과 더하기 위해 최종 단에서만 CPA가 요구되기 때문이다. 곱셈기 응용이 아닌 제곱 발생기 그 자체의 응용에서는 (그림 3)과 같이 최종 단계에서 캐리와 섬으로부터 최종 결과를 얻기 위한 2n비트의 CPA(carry propagation adder)가 요구된다.



(그림 3) 제곱 발생을 위한 출력

CPA는 일반적으로 CLA(Carry Lookahead Adder)로 구현하여 비트길이에 따른 지연을 완화한다.

3. 개선된 폴딩 기술

Wey와 Shieh가 제안한 폴딩 방법은 1의 보수화 부분에 의한 하드웨어 복잡성과 주소로부터 변환된 보정 값의 비트 길이가 길어져 하드웨어 복잡성이 크고 지연시간이 길어진다. 이 절에서는 제곱 발생기의 구조를 단순화하고 성능을 개선할 수 있는 새로운 폴딩 구조에 대하여 설명한다. 폴딩 구조는 ROM 주소의 MSB 또는 LSB를 분리하느냐에 따라 두 가지 구조를 갖게 된다.

3.1 MSB 분리

이진수 제곱 값을 발생시키기 위한 ROM 테이블을 이용하는 방법에서, 앞에서 설명한 폴딩 기술은 보정 회로를 두어 ROM 테이블 크기를 반으로 줄일 수 있었다. 본 절에서는 1의 보수 값을 사용하지 않고 ROM 테이블의 MSB를 분리 시켜 전혀 다른 보정 값을 필요로 하는 폴딩 기법에 대하여 설명한다. 이진수 제곱에서 MSB를 분리하여 전개하면 다음과 같은 특성 4를 얻을 수 있는데, 이 특성에 의한 보정 값 회로는 출력 비트 길이가 짧아져 회로가 더욱 간단해 짐을 알 수 있다.

특성 4. 주어진 n 비트 수 $N = (a_{n-1}a_{n-2}...a_1a_0)$에 대하여 다음과 같이 제곱 값을 구할 수 있다.

$$N^2 = (a_{n-2}...a_1a_0)^2 + a_{n-1}*(a_{n-2}\bar{a}_{n-2}a_{n-3}...a_1a_0)*2^n \quad (9)$$

증명.
$$\begin{aligned} N^2 &= (a_{n-1}a_{n-2}a_{n-3}...a_1a_0)^2 \\ &= \{a_{n-1}*2^{(n-1)} + (a_{n-2}a_{n-3}...a_1a_0)\}^2 \\ &= a_{n-1}{}^2*2^{2(n-1)} + a_{n-1}*(a_{n-2}a_{n-3}...a_1a_0)*2^n + \\ &\quad (a_{n-2}a_{n-3}...a_1a_0)^2 \\ &= a_{n-1}*\{2^{2(n-1)} + (a_{n-2}a_{n-3}...a_1a_0)*2^n\} + (a_{n-2}a_{n-3}...a_1a_0)^2 \\ &= a_{n-1}*\{2^{n-2} + (a_{n-2}a_{n-3}...a_1a_0)\}*2^n + (a_{n-2}a_{n-3}...a_1a_0)^2 \\ &\quad (2^{n-2} + a_{n-2}*2^{n-2} = (a_{n-2}\bar{a}_{n-2})*2^{n-2}) \\ &= a_{n-1}*(a_{n-2}\bar{a}_{n-2}a_{n-3}...a_1a_0)*2^n + (a_{n-2}a_{n-3}...a_1a_0)^2. \square \end{aligned}$$

예를 들어 5 비트의 수 $N = (a_4\,a_3\,a_2\,a_1\,a_0) = (10011) = 19$을 고려하자. 식 (9)에 의해 제곱 값은 다음과 같이 구할 수 있다.

$$\begin{array}{lll} 0\,1\,0\,1\,1 & & (a_3\bar{a}_3a_2a_1a_0)*2^n \\ +\quad 0\,0\,0\,0\,1\,0\,0\,1 & & (0\,0\,1\,1)^2 \\ \hline 0\,1\,0\,1\,1\,0\,1\,0\,0\,1 & & (361) \end{array}$$

첫 번째 데이터, $(a_3\bar{a}_3a_2a_1a_0)$,의 웨이트(weight)가 2^5 이기 때문에 비트 5의 위치 맞추어져 있다. 하드웨어를 구현할 때, ROM 출력의 하위 5 비트는 최종 결과가 되므로 10 비트가 아닌 단지 5 비트 크기의 CPA가 요구된다. ROM 크기를 더욱 줄이기 위하여 식 (9)를 다음과 같이 일반화 할 수 있다.

특성 5 (2^k folds). n 비트 숫자 $N = (a_{n-1}a_{n-2}...a_1a_0)$ 이 주어 지면, 다음과 같이 제곱 값을 구할 수 있다.

$$N^2 = (N^*)^2 + D*2^{n-k+1}, \qquad (10)$$

여기서

$$N^* = (a_{n-k-1}...a_1a_0)$$

그리고

$$D = a_{n-k}\triangle_k + ... + a_{n-2}\triangle_2 + a_{n-1}\triangle_1,$$

여기서

$\triangle_k = (\ 0\ 0 .....0\ a_{n-k-1}\ \bar{a}_{n-k-1}\ a_{n-k-2}\ .....a_1\ a_0)$
$\triangle_{k-1} = (\ 0\ 0 .....0\ a_{n-k}\ \bar{a}_{n-k}\ a_{n-k-1}\ a_{n-k-2}.....a_1\ a_0\ 0)$
$\triangle_{k-2} = (\ 0\ 0 .....0\ a_{n-k+1}\bar{a}_{n-k+1}\ a_{n-k}\ a_{n-k-1}.....a_1\ a_0\ 0\ 0)$
...
$\triangle_2 = (\ 0\ 0\ a_{n-3}\ \bar{a}_{n-3}\ a_{n-4}.....a_1\ a_0\ 0\ 0.....0)$
$\triangle_1 = (\ a_{n-2}\ \bar{a}_{n-2}\ a_{n-3}\ .....a_1\ a_0\ 0\ 0.....0)$

증명. $N^2 = (a_{n-1}a_{n-2}\,a_{n-3}...a_1a_0)^2$

$= a_{n-1}*(a_{n-2}\bar{a}_{n-2}a_{n-3}...a_1a_0)*2^n + (a_{n-2}a_{n-3}...a_1a_0)^2$

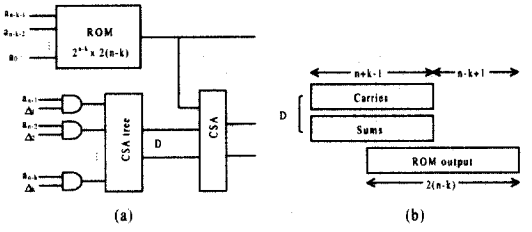
$= a_{n-1}*\triangle_1*2^{n-k+1} + (a_{n-2}a_{n-3}...a_1a_0)^2$

$= a_{n-1}*\triangle_1\,2^{n-k+1} + a_{n-2}*(a_{n-3}\bar{a}_{n-3}a_{n-4}...a_1a_0)*2^{n-1} + (a_{n-3}...a_1a_0)^2$

$= a_{n-1}*\triangle_1\,2^{n-k+1} + a_{n-2}*\triangle_2*2^{n-k+1} + (a_{n-3}...a_1a_0)^2$

비슷한 방법으로 마지막 항에서 MSB를 분리하여 전개하는 것을 반복하면

$N^2 = (a_{n-1}\triangle_1 + a_{n-2}\triangle_2 + ... + a_{n-k}\triangle_k)*2^{n-k+1} + (a_{n-k-1}...a_1a_0)^2$
$= (N^*)^2 + D*2^{n-k+1}$. □



(그림 4) MSB 분리. (a) 2^k-폴딩 기법에 대한 구성도. (b) 데이터 길이.

(그림 4a)에 보여준 구성도는 두 부분으로 이루어진다.

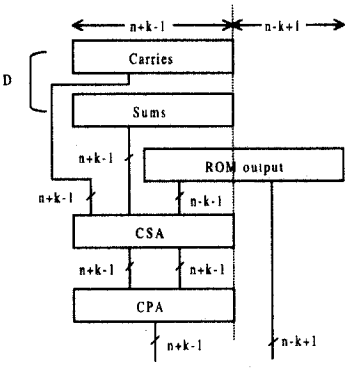
1) ROM 과
2) D-값 발생기(DG).

(그림 4a)에 보여준 구성도에서 주소 비트 $(a_{n-k-1}...a_1a_0)$는 1의 보수 부분이 필요 없이 직접 ROM의 주소로 공급된다. 따라서 1의 보수화 하기 위한 XOR 게이트 만큼의 지연이 감소된다. 보정 값을 위한 (그림 2)의 D^2 부분이 없어져서 *2(n-k)* 비트 ROM 출력은 *(n+k-1)* 비트의 D 값과 더해져야 하는데, D 값의 자리 수가 *(n-k+1)* 비트에서부터 시작하므로 단지 ROM 출력의 상위 *(n-k-1)* 비트만이 마지막 단의 CSA에서 더해져서 *(n+k-1)* 비트 캐리 비트와 섬 비트를 발생한다. 그리고 ROM에서 출력된 하위 *(n-k+1)* 비트는 캐리가 없는 섬 비트만이 출력 된 것으로 간주하면 된다. 캐리 비트가 없으므로 다른 제곱 값의 출력과 합을 구할 때 CSA가 간단해진다.

MSB 분리 방법에 의한 제곱 발생기의 지연시간은 Wey와 Shieh 가 제안한 폴딩 방법에서 1의 보수화 부분이 없어져서 식 (7)은 다음과 같이 나타낼 수 있다.

$$Pd = Pd_{ROM} + Pd_{CSA} \qquad (11)$$

$2^{10}*20$ 비트의 ROM, AND 게이트, 전가산기(CSA 기본 소자)의 지연시간이 각각 5.96ns, 0.9ns, 그리고 1.2ns라고 할 때, Wey와 Shieh 방법은 식 (7)에 의해 8.06ns의 지연시간이 요구되지만, MSB 분리 방법은 식 (11)에 의해 6.86ns 지연시간으로 줄어들게 된다. ROM의 크기를 줄이기 위하여 추가된 DG부분 때문에 전체 성능이 감소되지 않도록 DG의 CSA 트리의 최대 레벨 수는, L_{max}, 식 (8)에 똑같이 적용된다.



(그림 5) 제곱 발생을 위한 MSB 분리 방법

곱셈 연산 응용이 아닌 제곱 발생기 그 자체의 응용에서는 그림 5와 같이 최종 단계에서 캐리와 섬으로부터 최종 결과를 얻기 위한 CPA가 요구되는데, 2n 비트 보다 줄어든 단지 *(n+k-1)* 비트 크기의 CPA가 필요하다. 곱셈 연산 응용에서는 식 (1)과 같이 다른 제곱 발생기의 출력과 더하기 위해 최종 단에서만 CPA가 요구되지만, ROM 출력의 하위 *(n-k+1)* 비트는 최종 출력을 제공하므로, 두 개의 제곱 발생기의 출력을 더하기 위해 중간 과정에서 필요한 CSA의 입력 데이터 수가 줄어든다.

3.2 LSB 분리

ROM 주소의 MSB 대신에 LSB를 분리해서도 ROM 테이블 크기를 반으로 줄일 수 있는 비슷한 구조를 설계할 수 있다. 본 절에서는 이진수 제곱에서 LSB를 분리 시켜 또 다른 보정 값을 필요로 하는 폴딩 기법에 대하여 설명한다. 이진수 제곱에서 LSB를 분리하여 전개하면 다음과 같은 특성 6을 얻을 수 있는데, 이 특성에 의한 보정 값 회로도 3.1절의 MSB 분리 방법과 비슷하게 출력 비트 길이가 짧아져 회로가 간단해 짐을 알 수 있다. 그러나, 보정 값의 비트 위치가 비트 0에서부터 시작하여 제곱 값 자체 응용을 위한 CPA의 길이가 MSB 분리 방법보다 길어진다.

특성 6. n 비트 숫자 $N = (a_{n-1}a_{n-2}...a_1a_0)$이 주어 지면, 다음과 같이 제곱 값을 구할 수 있다.

$$N^2 = (a_{n-1}...a_1)^2 2^2 + a_0*(a_{n-1}a_{n-2}...a_10_1) \qquad (12)$$

증명. $N^2 = (a_{n-1}a_{n-2}\,a_{n-3}...a_1a_0)^2$

$= \{(a_{n-1}a_{n-2}...a_1)*2^1 + a_0\}^2$

$= (a_{n-1}a_{n-2}...a_1)^2 *2^2 + (a_{n-1}a_{n-2}...a_1) *a_0 *2^2 + a_0^2$

$= (a_{n-1}a_{n-2}...a_1)^2 *2^2 + (a_{n-1}a_{n-2}...a_100) *a_0 + a_0$

$= (a_{n-1}a_{n-2}...a_1)^2 *2^2 + (a_{n-1}a_{n-2}...a_101) *a_0$. □

예를 들어 5 비트의 수 $N = (a_4\,a_3\,a_2\,a_1\,a_0) = (10011) = 19$을 고려하자. 식 (12)에 의해 제곱 은 다음과 같이 구할 수 있다.

$$\begin{array}{rll} & 1\,0\,0\,1\,0\,1 & (a_4a_3a_2a_101) \\ + & 1\,0\,1\,0\,0\,0\,1 & (1\,0\,0\,1)\cdot 2^2 \\ \hline & 0\,1\,0\,1\,1\,0\,1\,0\,0\,1 & (361) \end{array}$$

두 번째 데이터는, (1010001), 비트 2의 위치에서 시작하도록 맞추어져 있다. 하드웨어를 구현할 때 첫 번째 데이터의 하위 2 비트는 최종결과가 되므로 8 비트 크기의 CPA 가 요구된다. ROM 크기를 더욱 줄이기 위하여 식 (12)를 다음과 같이 일반화 할 수 있다.

특성 7 (2^k folds). n 비트 숫자 $N = (a_{n-1}a_{n-2}...a_1a_0)$이 주어 지면, 다음과 같이 제곱 값을 구할 수 있다.

$$N^2 = (N^*)^2 *2^{2k} + D, \qquad (13)$$

여기서

$$N^* = (a_{n-1}...a_{k+1}a_k)$$

그리고

$$D = a_{k-1}\triangle_{k-1} + ... + a_1\triangle_1 + a_0\triangle_0$$

여기서

$\triangle_0 = (\,0.....0\;\; a_{n-1}\,a_{n-2}.....a_4\,a_3\,a_2\,a_1 0\,1)$

$\triangle_1 = (\,0.....0\;\; a_{n-1}\,a_{n-2}\;\; .....a_4\,a_3\,a_2\,0\;1\;0\;0)$

$\triangle_2 = (\,0.....0\;\; a_{n-1}a_{n-2}.....a_4\,a_3\,0\;1\;0\;0\;0\;0)$

...

$\triangle_{k-1} = (\,a_{n-1}a_{n-2}.....a_{k+1}a_k\;\; 0\;1\;0.....0)$

증명. $N^2 = (a_{n-1}a_{n-2}...a_1a_0)^2$

$= (a_{n-1}a_{n-2}...a_1)^2 *2^2 + (a_{n-1}a_{n-2}...a_101) *a_0$

$= (a_{n-1}a_{n-2}...a_1)^2 *2^2 + a_0\triangle_0$

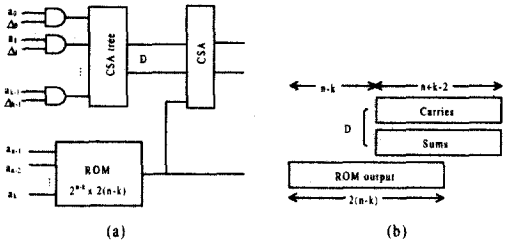
$= (a_{n-1}a_{n-2}...a_2)^2 *2^4 + (a_{n-1}a_{n-2}...a_201) *2^2 *a_1 + a_0\triangle_0$

$= (a_{n-1}a_{n-2}...a_2)^2 *2^4 + a_1\triangle_1 + a_0\triangle_0$

비슷한 방법으로 첫 번째 항에서 LSB를 분리하여 전개하는 것을 반복하면

$= (a_{n-1}a_{n-2}...a_k)^2 *2^{2k} + a_{k-1}\triangle_{k-1} + ... + a_1\triangle_1 + a_0\triangle_0$. □

(그림 6a)의 LSB 분리 구조는 (그림 4a)의 MSB 분리와 비슷한 구조를 보여준다. *2(n-k)* 비트의 ROM 출력은 *(n+k)* 비트의 D 값과 더해져야 하는데, ROM 출력 데이터는 *2k* 비트 위치에서 시작하므로 D 값의 상위 *(n-k)* 비트만이 ROM 출력과 *(n-k+1)* 비트의 CSA에서 더해진다.



(그림 6) LSB 분리. (a) 2^k-폴딩 기법에 대한 구성도. (b) 데이터 길이.

제곱 값 자체를 구하는 응용에서 CPA는 MSB 분리 방법의 *(n+k-1)*에 비하여 늘어난 *2(n-1)* 비트 길이의 CPA가 필요하다.

3.3 폴딩 기법 비교

Wey와 Shieh의 방법에 비하여 제안된 MSB 분리 및 LSB 분리 방법은 1의 보수를 발생시키는 부분을 없애어 지연시간이 짧아지고 회로가 간단해진다. 전체 지연시간이 ROM의 크기에 따라 지연시간이 달라지지만 19 비트 크기의 수에 대한 제곱 발생기를 만들 때 8.06ns에서 6.86ns로 줄어 드는 것을 확인 하였다. 그리고 MSB 분리 및 LSB 분리 방법은 보정 값의 비트길이가 짧아져서 더욱 회로가 간단해진다. 그리고 제곱 자체 응용을 위해서는 (그림 3, 5, 6)에서 처럼 최종 단에 CPA가 필요하게 되는데, MSB 분리 방식이 가장 짧은 CPA를 요구한다. 표 2는 Wey와 Shieh의 방법, 그리고 제안된 MSB 분리 및 LSB 분리와의 비교를 보여준다. 표에서 OC는 ROM 테이블의 주소로 사용하기 위하여 제곱할 수에 대한 1의 보수를 발생시키는 부분이다. D 길이는 줄어든 테이블에서 구한 제곱 값에 더할 보정 값의 비트 길이를 나타낸다. CPA 길이는 제곱 자체 응용에서 요구되는 CPA의 비트 길이를 나타낸다.

〈표 2〉 폴딩 기법 비교

폴딩 기법	Wey & Shiehs 기법	MSB 분리	LSB 분리
OC	(n-k) 개의 xor 게이트	Not required	Not required
D 길이(bit)	2n	n+k-1	n+k
CPA 길이(bit)	2n	n+k-1	2(n-2)

4. 결 론

고속 곱셈 연산은 디지털 신호처리, 영상처리 등 많은 응용분야에 필요한 기술이다. 고속 곱셈을 위하여 제곱 테이블을 이용한 곱셈 방법은 처리과정을 간단히 하고 속도도 향상시킨다. 그러나, 비트 길이가 증가함에 따라 테이블 크기는 지수 승으로 증가하게 된다.

본 논문에서는 고속 곱셈이 요구되는 응용분야에 적합한 Wey와 Shieh의 우수한 제곱 발생기의 구조를 개선하여 회로를 간단히 하고 지연시간을 줄여 성능을 향상 시킬 수 있는 새로운 폴딩 기법을 제안 하였다. 제안된 MSB 분리 및 LSB 분리 기법은 [9]에서 요구되는 1의 보수화 부분을 없애고 보정 값 D의 비트 길이를 줄임으로써 성능을 개선 시키고 하드웨어 복잡성을 줄여 고속 제곱 발생기를 위한 회로를 간단히 할 수 있다. 제곱 연산 자체 응용을 위해서는 필요한 CPA 길이를 줄임으로써 더욱 회로가 간단해진다.

참 고 문 헌

[1] I. Koren, Computer Arithmetic Algorithm. Prentice Hall, 1993.

[2] K. Hwang, Computer arithmetic. New York : Wiley, 1979.

[3] M. A. Soderstrand, W.K. Jenkins, G.A.Jullien, and F.J. aylorr, Residue Number Systems Arithmetic: Modern Applications to Digital Signal Processing. New York : IEEE Press, 1986.

[4] H. Ling, "An Approach to Implementing Multiplication with Small Tables," IEEE Trans. Computers, Vol.39, No.5, pp.717-718, May 1990.

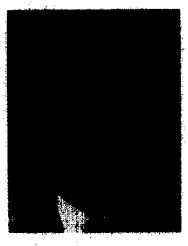
[5] B. Vinnakota, "Implementing Multiplication with Split Read-Only Memory," IEEE Trans. Computers, Vol.44, No.11, pp.1352-1356, Nov. 1993.

[6] C.L. Wey and T.Y. Chang, "Design and Analysis of VLSI-Based Parallel Multipliers," IEE Proc. Part E, Vol.137, No.4, pp.328-336, July 1990.

[7] C.L. Wey, "On the Design of Efficient Squares-Based Multipliers," proc IEEE Intl Conf. Computer Design (ICCD 96), pp.506-513, Austin, Tex., Oct. 1996.

[8] J.F. Cavanagh, Digital Computer Arithmetic. McGraw-Hill, 1984.

[9] C.L. Wey and M.D. Shieh, "Design of a High-Speed Square Generator," IEEE Trans. Computers, Vol.47, No.9, pp.1021-1026, Sep. 1998.



송 상 훈

e-mail : song@kunja.sejong.ac.kr

1977년 연세대학교 전자공학과 (학사)

1979년 KAIST 전산학과(석사)

1992년 University of Minnesota 전산학과 (박사)

1992년~현재 세종대학교 공과대학 컴퓨터공학과 부교수

관심분야 : 분산/병렬 처리, 컴퓨터 구조, computer arithmetic