論文2001-38SD-12-6

멀티미디어 데이터 처리에 적합한 SIMD MAC 연산기의 설계

(SIMD MAC Unit Design for Multimedia Data Processing)

洪 仁 杓 *, 鄭 愚 暻 *, 鄭 在 元 *, 李 溶 錫 *

(In-Pyo Hong, Woo-Kyong Jeong, Jaewon Jeong, and Yong-Surk Lee)

요 약

MAC(Multiply and ACcumulate) 연산은 DSP와 멀티미디어 데이터 처리의 핵심이 되는 연산이다. 기존의 DSP 혹은 내장형 프로세서의 MAC 연산기들은 주로 3사이클의 latency를 가지며, 한번에 하나씩의 데이터를 처리하므로 성능에 한계를 보인다. 따라서 고성능의 범용 프로세서들은 SIMD(Single Instruction Multiple Data) 연산을 지원하는 MAC 연산기를 실행 유닛으로 내장하는 추세이다. 하지만 이러한 고성능의 연산기는 고성능 범용 프로세서의 특성상 다양한 동작 모드를 지원해야 하고 clock 주파수가 높아야 하므로 파이프라인 기법을 사용하고 이에 따른 컨트롤이 복잡하여 하드웨어 설계가 까다롭고 면적이 큰 문제가 있다. 본 논문에서는 내장형 프로세서에 적합한 64비트 폭을 갖는 SIMD MAC 연산기를 설계하였다. 한 사이클에 누적연산까지 모두 완료하도록 하여 파이프라인 제어의 필요성을 없앴고, 기존의 Booth 곱셈기 구조에 기반하여 약간의 회로 추가로 SIMD 연산이 가능하도록 하였다.

Abstract

MAC(Multiply and ACcumulate) is the core operation of multimedia data processing. Because MAC units implemented on traditional DSP units or embedded processors have latency of three cycles and cannot operate on multiple data simultaneously, then, performances are seriously limited. Many high end general purpose microprocessors have SIMD MAC unit as a functional unit. But these high end MAC units must support pipeline structure for various operation modes and high clock frequency, which makes control logic complex and increases chip area. In this paper, a 64bit SIMD MAC unit for embedded processors is designed. It is implemented to have a latency of one clock cycle to remove pipeline control logics and a minimal area overhead for SIMD support is added to existing Booth multipliers.

Ⅰ. 서 론

최근 컴퓨팅 환경의 변화로 멀티미디어 자료 처리 수요가 증가하고, 무선 통신에의 적용 등으로, 프로세서에 있어서 DSP 기능의 중요성이 크게 부각되고 있다. 여러 DSP 관련 연산 중에서도 MAC (Multiply and ACcumulate) 연산이 가장 핵심이 되는 부분이라고 할 수 있는데, 주로 8~24비트 폭을 갖는 데이터에 대한 MAC 연산을 효율적으로 처리하는 연산기의 구조에 대한 연구가 필요하다.

기존의 연산기들은 작은 폭을 가진 데이터에 대하여도 연산기의 데이터 폭을 그대로 적용하여 데이터를 부호 확장하여 연산하므로, 데이터의 폭이 작다고 하여도 동시에 여러 개의 데이터를 처리할 수는 없다. 이것은 결국 연산기의 회로를 낭비하는 것이 된다. 따라서

* 正會員, 延世大學校 電氣電子工學科 프로세서 硏究室
(Processor Laboratory, Dept. of Electrical and Electronic Engineering, Yonsei University)

接受日字:2001年2月7日, 수정완료일:2001年10月30日

연산기의 회로를 효율적으로 이용함으로써 64비트 정도의 큰 폭을 갖는 연산기로 8~24비트의 작은 데이터 여러 개를 동시에 처리할 수 있도록 하는 SIMD 연산기가 활발히 개발 중이며, 대부분의 고성능 프로세서들은 이런 연산기를 별도의 실행 유닛으로 내장하는 추세이다. 그러나 현재 사용되는 SIMD 연산기들은 주로 고성능의 수퍼스칼라(Superscalar) 마이크로프로세서에 내장된 것들로서 범용 마이크로 프로세서의 특성상 다양한 동작모드와 명령을 지원해야 하고 동작 주파수를 높이기 위해 파이프라인 기법이 사용되므로 컨트롤이 복잡하고 설계가 까다로우며 면적인 큰 특징이 있다. 이러한 문제점으로 기존의 SIMD 연산기를 저가의 내장형 프로세서에 첨가하는 것은 효율적이지 못하다. 그러나 점차 내장형 프로세서에서도 이러한 용도의 연산기가 요구되어지고 있는 추세이므로, 비교적 규모가 작고 설계가 간단하여 내장형 프로세서에 적합한 SIMD MAC 연산기를 설계하였다. 본 연산기는 100MHz이하의 동작 주파수를 갖는 내장형 프로세서에 내장하기 위한 목적으로 설계되었으며, 복잡한 파이프라인 제어를 피하기 위하여 누적연산까지를 한 사이클에 처리하도록 하였다. 내장형 프로세서에는 단순한 핵심 기능만을 내장하여야 하므로 32비트 부호/무부호(signed/unsigned) SIMD, 16비트 부호(signed) SIMD 연산의 세가지 동작 모드만을 지원한다. 또한 내장형 프로세서는 개발 비용이 적어야 하고 동작 주파수도 그리 빠르지 않으므로, 설계와 검증이 용이한 HDL을 이용하여 표준 셀 공정으로 회로를 합성하는 설계 방법을 선택하였다.

본 논문에서는 2장에서 기존의 곱셈 알고리즘과 MAC 연산기들을 살펴보고, 현재 구현되어 있는 SIMD 연산기들에 대하여 알아본다. 3장에서 본 논문에서 제시한 SIMD MAC 연산기의 세부적인 구조를 설명하였다. 4장에서 하드웨어 구현 방법을 기술하였다. 5장에서는 설계된 연산기의 성능을 평가하고, 6장에서 결론을 맺는다.

II. MAC 연산기의 개요

1. MAC 연산의 개요

MAC 연산은 곱셈을 수행한 결과를 accumulator register에 누적하여 저장하는 연산으로 다음과 같은 수식으로 정의될 수 있다.

$$acc' = acc + x \times y \quad (1)$$

식 (1)에서 보듯이 MAC 연산은 곱셈 부분과 덧셈 부분으로 구성되어 있음을 알 수 있다. 따라서 일반적인 MAC 연산기는 곱셈기와 곱셈의 결과를 덧셈하는 덧셈기를 가지게 되고 이를 block diagram으로 나타내면 그림 1과 같다.

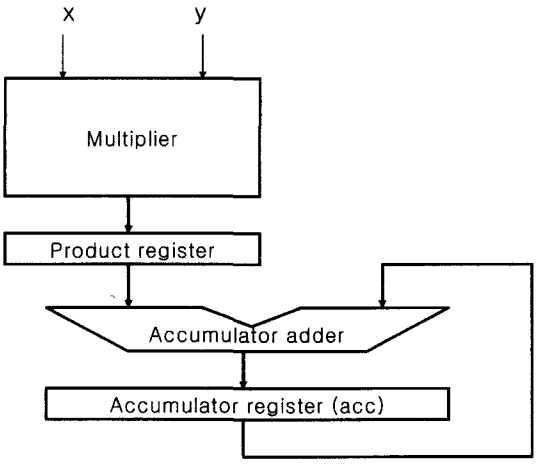


그림 1. 일반적인 MAC 연산기의 block diagram
Fig. 1. Block diagram of general MAC unit.

이러한 구조의 연산기는 현재 대부분의 DSP에 내장되어 있는 것이다. 이는 SIMD 연산이 불가능할 뿐 아니라, 캐리전파(carry propagation)가 발생하는 누적덧셈기(accumulator adder)가 큰 bit 폭을 갖고 있으므로 latency가 늘어나 연산기의 성능은 좋지 못하다. 이를 극복하고 throughput을 높이기 위하여 3 stage 정도의 파이프라인을 적용하고 있다. 그러나 파이프라인을 적용하게 되면, data dependency나 exception에 따른 성능 저하 문제가 발생할 수 있다.

2. 곱셈기의 개요

곱셈기는 MAC 연산기에서 가장 큰 비중을 차지하고 있는 부분이며 곱셈 연산은 전체 연산기의 latency의 거의 대부분을 차지한다.

과거의 곱셈기는 shift and add를 이용한 방법이나, 이를 약간 개량한 adder array를 사용한 구조를 채택하였다. shift and add를 이용하는 방법은 2진수 곱셈을 손으로 계산하는 과정을 하드웨어로 그대로 옮겨 놓은 것으로 n-bit 곱셈시에 n개의 부분곱들을 계속 shift하여 누적 덧셈하여 구현된다. 이 구조는 n-bit 곱셈을 위해서는 n-cycle이 걸리게 된다. 이는 초기의 곱셈기

구조로 현재에는 곱셈 시간이 중요하지 않은 계산기와 같은 응용에 사용되거나, 마이크로콘트롤러에서 ALU만으로 곱셈을 구현하는 경우 사용된다.

곱셈을 빠르게 처리하기 위하여 곱셈시 소요되는 덧셈의 횟수를 줄일 수 있도록 고안된 것이 Booth 알고리듬이다.[1]

$$01111111_2 = 10000000_2 - 00000001_2 \quad (2)$$

식 (2)와 같이 Booth 알고리듬은 승수에 '1'의 반복을 두 번의 덧셈으로 대체시킨다. '0'의 반복은 아무런 동작을 하지 않는다. 이런 방법을 사용하면 곱셈에 소요되는 덧셈의 수를 줄일 수 있다. 이런 원리를 하드웨어에 적용하기 용이하도록 변형한 것이 수정 Booth 알고리듬이다. Radix-4 Booth 알고리듬은 승수를 한 비트 중첩하여 세 비트씩 묶어 Booth 알고리듬을 적용한다. 세 비트의 조합 각각에 대한 세부 동작과 곱셈기의 분할방법은 표 1과 그림 2에 각각 제시하였다.

표 1. Radix-4 Modified Booth 알고리듬
Table 1. Radix-4 Modified Booth Algorithm.

	동작 설명	동작
0 0 0	연속된 '0', no operation	0
0 0 1	String의 시작, X를 덧셈	+X
0 1 0	X를 덧셈	+X
0 1 1	String의 시작	+2X
1 0 0	String의 끝	-2X
1 0 1	String의 끝, 새 string의 시작	-X
1 1 0	String의 끝, X를 뺄셈	-X
1 1 1	연속된 '1'의 중간	0

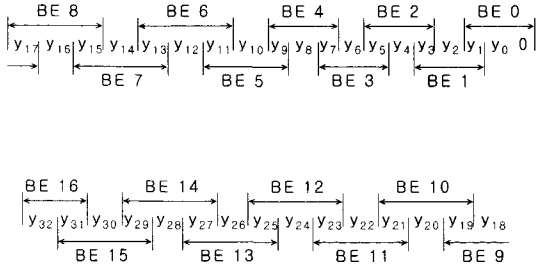


그림 2. 승수의 분리
Fig. 2. Breaking multiplier in 32bit radix-4 modified Booth algorithm.

위와 같은 Booth 알고리듬을 사용하면 곱셈시에 소요되는 덧셈의 수를 반으로 줄일 수 있다. 만약 승수를 한 비트 중첩하여 네 비트씩 묶는 radix-8 Booth 알고리듬을 사용하면 덧셈의 수를 약 1/3로 줄일 수 있게 된다.

3. SIMD 연산기의 개요

SIMD 연산기는 하나의 명령으로 여러 개의 데이터에 동일한 연산이 가해지도록 구성된 연산기이다. 현재는 대부분의 고성능 범용 마이크로프로세서에 내장되어 있으며, 주로 64비트 혹은 128비트의 폭을 갖는 연산기로, 8비트, 16비트, 32비트 데이터를 여러 개 pack하여 연산을 취하는 방식을 채택하고 있다. 그림 3에서는 64비트 연산기로 4개의 16비트 데이터를 덧셈하는 예를 보여준다.

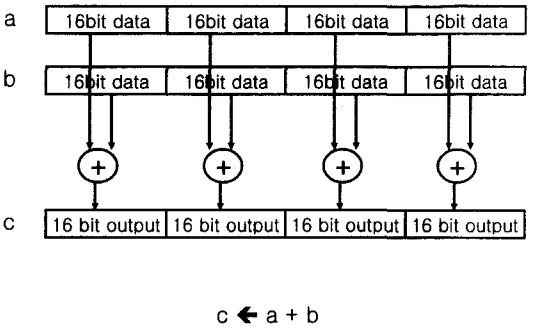


그림 3. SIMD 덧셈 연산의 예
Fig. 3. Example of SIMD add operation.

대부분의 SIMD 연산기들은 SIMD ALU, SIMD 곱셈기, SIMD FPU등을 내장하고 있다. 이들은 SIMD 곱셈기로 MAC 연산을 처리하도록 되어 있다. 이 경우 별도의 accumulator register를 두지 않고, 일반 SIMD register에 누적된 값을 저장하는 방법을 쓰고 있는데, 이 경우 오차의 누적 문제가 있을 수 있다.

그리고 대부분의 SIMD MAC 연산기들은 8비트, 16비트의 곱셈을 수행할 경우에 대비하여 8비트, 16비트, 32비트 각각의 월레스 트리를 따로 두고, 여기서 나온 결과를 선택, 조합하여 최종 결과를 생성하도록 되어 있어 MUX와 추가적인 덧셈기 등이 필요하게 된다.

또한 이들은 SIMD 연산을 하도록 분할된 8, 16비트 각각의 필드들을 조합하여 연산하는 방식에 따라 MAC 연산에도 많은 종류의 명령어가 존재하고, 이를 구현하기 위하여 많은 MUX들이 사용되므로, 회로가 복잡하

고 지연시간이 증가한다. 위에 언급한 모든 연산기들은 MAC 연산을 3 사이클의 지연시간을 가지도록 파이프라인으로 구현한 것이다.[2]

이러한 연산기들은 위에 말한 문제점으로 인하여 내장형 프로세서에는 적합치 않다. 따라서 본 논문에서는 내장형 프로세서에 맞는 간단한 구조로 핵심 기능만을 수행하는 연산기를 설계하였다.

Ⅲ. SIMD MAC 연산기의 구조

본 연산기는 기존의 32비트 radix-4 Booth 알고리듬을 적용한 곱셈기를 기반으로 하여 최소의 면적/latency overhead로 SIMD 연산을 할 수 있도록 하는 데에 중점을 두고 설계되었다. Booth encoder, 부분곱 생성기, 월레스 트리 블록은 적절한 캐리전파 조절을 통하여 하나의 회로로 32비트 연산, SIMD 16비트 연산을 모두 지원할 수 있도록 설계되었고, 누적 덧셈기와 최종 덧셈기는 곱셈기 하나당 두 개씩의 누적 저장기를 지원해야 하므로, 이에 맞추어 추가하였다. 또한 복잡한 파이프라인 제어를 피하기 위하여 한 사이클에 누적 덧셈까지 모든 연산을 완료할 수 있도록 하였다. 이를 위하여 누적 덧셈기는 곱셈기의 adder tree에 통합하여 부분곱들과 함께 누적 저장기에 누적된 값을 덧셈할 수 있도록 하여 누적 덧셈시에 소요되는 지연시간을 상당 부분 감소시킬 수 있도록 하였다.

설계된 연산기는 16비트 SIMD 연산을 할 수 있도록 설계된 32비트 곱셈기 두 개와 네 개의 64비트 누적 저장기, 그리고 곱셈 후 뺄셈 동작을 위한 입력 조절기로 구성되어 있다. 이 연산기는 32비트 부호/무부호 MAC 연산 두 개, 16비트 부호 MAC 연산 네 개를 한 사이클에 처리할 수 있다. 연산기의 전체적인 block diagram은 그림 4에서 볼 수 있다.

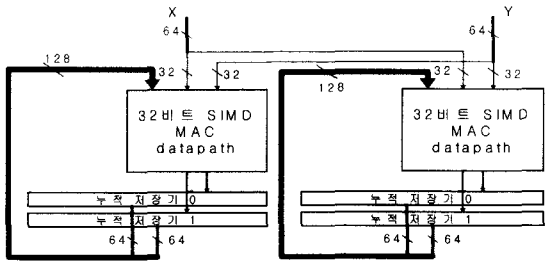


그림 4. SIMD MAC 연산기의 block diagram
Fig. 4. Block diagram of SIMD MAC unit.

1. Booth encoder의 구조

일반적인 Booth Encoder는 단순히 승수를 한 비트씩 중첩하여 세 비트씩 분리하고 decoder를 거쳐 다섯 가지의 동작(nop, X, 2X, -X, -2X)을 표현할 수 있는 신호를 만들어 부분곱 생성기(Partial Product Generator)로 보낸다.

본 논문에서 설계한 곱셈기는 32비트의 승수를 두 부분으로 분할하여 곱셈을 수행할 수 있어야 하므로 기본적인 Booth encoder[3]를 약간 수정하여 사용한다. 16비트 연산을 하는 경우, 16개의 기본 블록들을 8개씩 묶어 하나의 연산에 사용하게 된다. 즉, BE_0~BE_7까지는 LSB쪽 16비트 곱셈에 쓰이고, BE_8~BE_{15}까지는 MSB쪽 16비트 연산에 쓰인다. 이 때, BE_8의 Y_{i-1}은 32비트 연산을 하는 경우에는 Y_{15}가 입력되지만, 16비트 SIMD 연산을 하는 경우에는 이 부분이 MSB쪽 16비트의 Y_{-1} 위치가 되므로, '0'이 입력되어야 한다. 따라서 동작 모드에 따라서 이 부분의 입력을 '0'으로 조절할 수 있는 기능이 필요하다. 이 기능을 위하여는 일반적인 Booth encoder에 하나의 AND 게이트를 추가하였다. 그림 5에는 16비트 SIMD 연산을 하도록 수정된 Booth Encoder의 구조가 나타나 있다.

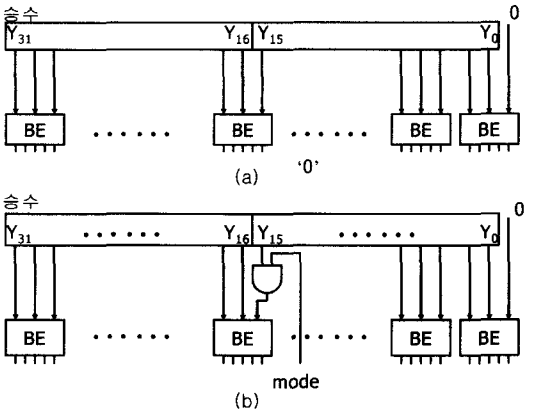


그림 5. Booth encoder (a)일반적인 Booth encoder (b) SIMD 지원을 위해 수정된 block
Fig. 5. Booth encoder (a) traditional Booth encoder (b) modified for SIMD operation.

2. 부분곱 생성기의 구조

부분곱 생성기는 Booth Encoder의 출력과 피승수를 입력으로 받아 0, X, 2X, -2X, -X 중 하나의 부분곱을 생성하여 출력한다. 일반적인 부분곱 생성기의 기본 블

록은 5:1 MUX이다. 2X는 X를 한 비트 왼쪽으로 쉬프트 하면 얻어지고, -X는 X를 반전시킨 후, 월레스 트리의 해당 위치의 LSB에 1을 입력하여 덧셈하여 주며, -2X는 -X를 왼쪽으로 한 비트 쉬프트 시켜주면 된다. 이러한 동작은 5:1 MUX로 수행할 수 있다.

부분곱 생성기는 월레스 트리에서 부분곱들 간의 부호확장을 미리 삽입하여 월레스 트리로 보내주게 되는데, 이때 사용한 방법은 부호생성 알고리듬이다.[4] 부분곱들은 기준 자리수에 따라 두 비트씩 왼쪽 쉬프트되어 더해지게 되는데, 이때 MSB에서 부호 확장이 일어나야 한다.

부호생성 알고리듬은 다음과 같은 과정으로 MSB의 부호 확장을 단축하는 방법이다.

1. 각 부분곱의 부호비트를 반전하여 MSB에 추가
2. 각 부분곱의 MSB에 '1'을 추가한다.
3. 곱셈 결과에 2^N을 더한다. (N은 피승수의 비트수)

SIMD MAC 연산기는 16비트 SIMD동작을 하기 위하여 월레스 트리의 일부를 분할하여 사용하므로, 일반적인 부분곱 생성기에 약간의 변형을 가하여 사용한다. 그림 6의 월레스 트리에서 흰색으로 나타난 부분은 SIMD 동작을 하는 경우에 '0'이 채워져야 하는 부분이다. 부호확장과 '0'을 채우는 동작을 월레스 트리에서 수행하면 설계와 검증이 힘든 월레스 트리의 복잡도가 증가하므로, 부분곱 생성기에서 이러한 동작을 미리 수행하여 월레스 트리로 보내주어 월레스 트리는 단순한 덧셈동작만을 담당할 수 있도록 하는 것이 합당하다.

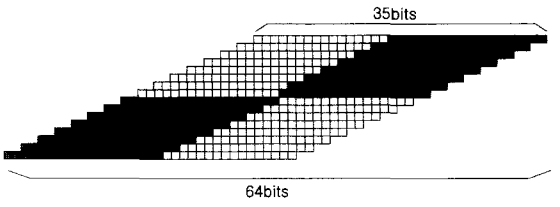


그림 6. SIMD 지원을 위한 부분곱의 조절
Fig. 6. Control of partial products for SIMD operation support.

위에 설명한 동작을 수행하기 위하여 6:1 MUX를 사용하였다. 일반적인 5:1 MUX에 하나의 입력을 추가하여, 추가된 입력에 SIMD 연산 시에 사용되어질 입력을 가하여 주는 것이다. 각 부분곱의 18, 19번째 비트는 32비트 연산인 경우는 피승수의 해당 위치 비트가 삽입되지만, 16비트 연산인 경우는 16비트 연산의 부호생성 알고리즘에 의한 비트들이 입력되어야 한다. 따라서 각 부분곱의 18, 19번째 비트의 6:1MUX 추가입력에는 16비트 연산에 해당하는 부호생성비트들이 입력된다. 이외의 경우는 SIMD연산시에 '0'으로 채워지는 부분이므로, '0'을 입력한다.

3. 월레스 트리의 구조

Radix-4 수정 Booth 알고리즘을 사용할 때, 32바트 곱셈이면 무부호 곱셈을 위해 필요한 것까지 총 17개의 부분곱을 더해야 한다. 월레스 트리는 이들 17개의 부분곱을 더하여 캐리 벡터(Carry vector)와 합 벡터(Sum vector)를 출력한다. 이들 벡터는 최종 덧셈기에서 더하여져 곱셈의 결과가 된다.

월레스 트리는 캐리전파를 덧셈기의 덧셈 지연시간과 겹치도록 함으로써 덧셈에 걸리는 시간을 줄이는 방법이다. 그림 7의 화살표로 표시된 캐리에 유의하여 보면, PP[i]의 첫 번째 단에서 8개의 부분곱 비트들을 더하여 4개의 캐리를 만드는 데, 이 캐리는 PP[i+1]의 두 번째 단의 덧셈기로 입력되므로, 이 캐리전파는 PP[i+1]의 첫 번째 단이 덧셈을 하는 시간과 중첩되는 것을 볼 수 있다. 즉, 각 부분곱 열의 두 번째 단이 덧셈 수행을 시작하는 순간에는 모든 캐리 비트들이 입력되어 있는 것이다. 따라서 덧셈기는 캐리 입력을 기다릴 필요가 없이 바로 연산할 수 있다. 이러한 일이 부분곱의 모든 열에 걸쳐 일어나므로 월레스 트리는 8개의 전가산기의 지연시간 만으로 17개의 35비트 부분곱들을 덧셈할 수 있다. 본 연산기의 월레스 트리는 캐리 중첩에 적합하고, 덧셈기 트리의 규칙적 배열이 가능한 4:2 CSA를 사용하여 구현되었다. 4:2 CSA는 두 단의 3:2 CSA를 합쳐 놓은 구조로 되어있는데, 기본적

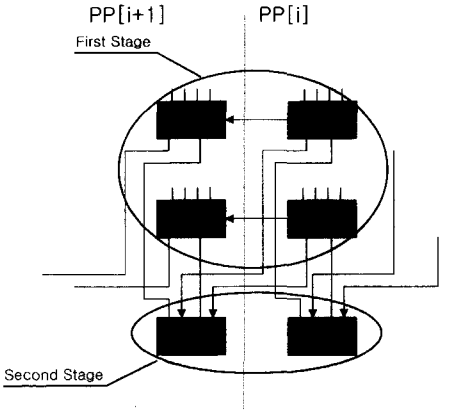


그림 7. 월레스 트리에서의 캐리 중첩
Fig. 7. Removing carry popagation in Wallace tree.

으로 캐리 입력과 캐리 출력간의 지연시간을 중첩시킬 수 있는 구조이므로 월레스 트리에 적합한 것이다. 그 구조가 그림 8에 나타나 있다.[5]

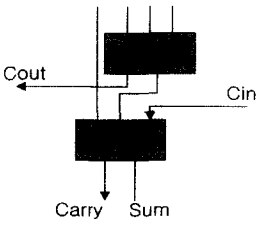


그림 8. 4:2 CSA
Fig. 8. 4:2 CSA.

16비트 SIMD 동작을 하는 경우에는, 16비트 곱셈의 결과가 32비트이므로, 곱셈기의 총 64비트 출력 중, LSB쪽 32비트와 MSB쪽 32비트가 각기 독립적인 연산이 되어야 한다. 이때, 월레스 트리의 32번째 열에서 만들어진 캐리가 33번째 열로 전파되면 하나의 곱셈 결과가 다른 곱셈에 영향을 미치게 되는 것이므로, 32번째 열에서 만들어진 캐리는 다음 열로 전파시켜서는 안 된다. 따라서 그림 9와 같이 월레스 트리의 가운데 열에 AND 게이트 7개를 추가하여 동작 모드에 따라 캐리를 조절할 수 있도록 하였다.

그림에서 mode 신호가 '0'이면 캐리가 차단되므로 월레스 트리가 독립적인 두 부분으로 분할되어 16비트 동작을 하고, '1'이면 캐리가 연결되므로 32비트 곱셈동작을 하게 된다. 월레스 트리에서 32번째 열에서 33번째 열로 입력되는 캐리는 총 14비트가 된다. 하지만, 월레스 트리에서 32번째 열에 '0'으로 채워지는 비트들을 덧셈하여 나온 캐리 비트들은 항상 '0'이므로 이 14비트의 캐리 입력 중에 실제 캐리 정보를 전달하는 7비트만을 선택하여 캐리를 조절할 수 있도록 하였다.

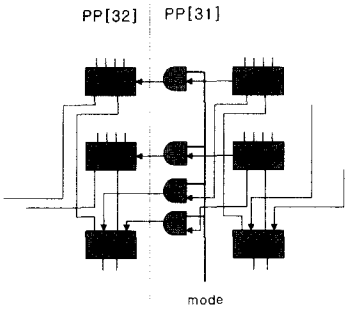


그림 9. 월레스 트리에서의 캐리 조절
Fig. 9. Carry control in Wallace tree.

4. 누적덧셈기의 구조

누적 덧셈기는 곱셈의 결과와 누적 저장기(Accumulator Register)의 데이터를 더하는 역할을 수행한다. 기존의 MAC 연산기들은 2장에서 알아본 바와 같이, 곱셈의 결과와 누적 저장기의 데이터를 더하는 과정에서 캐리전파가 일어나므로, 지연시간이 큰 단점이 있다.

본 논문에서 제시된 MAC 연산기는 누적 덧셈기를 곱셈기의 월레스 트리에 통합시킴으로써 누적 덧셈기의 캐리전파로 인한 지연시간의 영향을 감출 수 있도록 하였다. 그림 10, 그림 11과 같이 월레스 트리에서 나온 합 벡터와 캐리 벡터를 누적 저장기의 데이터와 한 번 더 더하여 새로운 합 벡터와 캐리 벡터를 만들고 이것을 최종 덧셈기의 입력으로 가하여 준다. 이렇게 하면 누적 덧셈 시에 캐리전파가 일어나지 않으므로, 하나의 전가산기의 지연시간만으로 누적 덧셈이 이루어질 수 있다.

16비트 SIMD연산의 곱셈 결과는 합 벡터와 캐리 벡터가 32비트이므로, 이들을 64비트인 누적 저장기의 데이터와 덧셈을 할 때는 합 벡터와 캐리 벡터를 부호확장할 필요가 있다. 따라서 월레스 트리와 추가된 덧셈기 사이에 부호 확장 블록이 필요하게 된다. 이 부호확장 블록은 합 벡터와 캐리 벡터의 MSB쪽 32비트 자리에 부호 확장을 하게 된다. 합 벡터는 부호생성 알고리즘에 의해 MSB가 항상 '1'이다. 따라서 합 벡터의 부호확장은 항상 '1'이 된다. 캐리 벡터는 월레스 트리의 32번째 열에서 차단하였던 캐리 비트들을 덧셈하여 캐리 벡터의 MSB에 추가하여 주는 방법으로 부호확장을 구현한다. 32 비트 MAC 동작을 하는 경우에는 곱셈의 결과가 64비트이므로 부호 확장을 하지 않고 월레스 트리에서 나온 64비트의 합 벡터와 캐리 벡터를 누적 덧셈기로 그대로 전달한다.

16비트 연산인 경우 두 개의 곱셈 결과가 각기 다른 누적 저장기에 누적되어야 하고, 각 누적 저장기는 64비트이므로, 하나의 32비트 MAC 연산기는 64비트의 누적 저장기와 최종 덧셈기, 누적 덧셈기를 두 개씩 가지고 있다. 최종적인 누적 덧셈기와 곱셈 블록의 통합은 그림 11에 나타나 있다. 이 중 누적 덧셈기 '1'은 16비트 SIMD 연산을 하는 경우에만 사용되는 것으로서, 32비트 곱셈 누적 연산을 하는 경우에는 합 벡터와 캐리 벡터를 AND 게이트에 연결하여 누적 덧셈기에 모

두 '0'이 입력될 수 있도록 하였다. 이렇게 하면 32비트 곱셈 누적 연산시에는 누적 저장기 1에는 자기 자신이 전번 사이클에 가지고 있던 데이터가 그대로 들어가므로, 본래의 데이터를 유지할 수 있다.

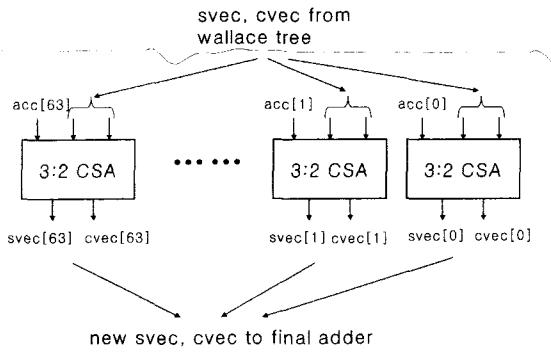


그림 10.누적 덧셈기의 구조
Fig. 10. accumulator adder.

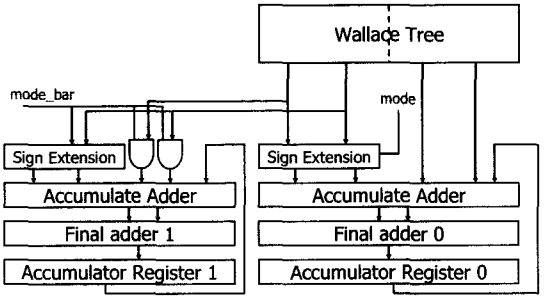


그림 11.누적 덧셈기와 곱셈기의 통합
Fig. 11. Combining accumulator adder and multiplier.

5. 입력 조절기의 구조

입력 조절기는 곱셈 후 뺄셈(Multiply and subtract) 연산을 위한 것이다. 곱셈 후 뺄셈 연산은 다음과 같은 연산이다.

$$acc = acc - x \times y \tag{3}$$

이러한 연산을 수행하는 데는 두 가지 방법이 있다. 첫 번째는 곱셈의 결과로 나오는 합 벡터와 캐리 벡터를 2의 보수화 하여 부호를 반전시키는 방법이고, 두 번째는 곱셈기의 승수 입력을 2의 보수화 하여 부호를 반전시키는 방법이다. 첫 번째 방법은 식 (3)을 그대로 구현한 것이라고 한다면 두 번째 방법은 다음 식과 같이 생각할 수 있다.

$$acc = acc + x \times (-y) \tag{4}$$

첫 번째 방법은 64비트의 sum vector와 carry vector를 모두 2의 보수화하여야 하므로, 그에 따른 overhead가 크다. 두 번째 방법은 승수 입력을 반전시키기만 하면 되므로, 32개의 XOR 게이트로 구현될 수 있다. 여기서 승수 입력을 부호 반전 하려면 2의 보수를 취해야 하지만, Booth 알고리듬의 특성상, 표 2와 같이 반전만 시키면 된다.

표 2. 승수 반전에 따른 곱셈 동작의 변화
Table 2. Effect of inverting multiplier.

원래 승수		반전된 승수	
	동작		동작
0 0 0	nop	1 1 1	nop
0 0 1	+X	1 1 0	-X
0 1 0	+X	1 0 1	-X
0 1 1	+2X	1 0 0	-2X
1 0 0	-2X	0 1 1	+2X
1 0 1	-X	0 1 0	+X
1 1 0	-X	0 0 1	+X
1 1 1	nop	0 0 0	nop

6. 최종 덧셈기의 구조

최종 덧셈기는 accumulator adder에서 나온 sum vector와 carry vector를 합하여 MAC 연산기의 최종 결과를 만들어낸다. 최종 덧셈기는 월레스 트리와는 달리 64비트의 캐리전파가 지연시간에 그대로 반영되므로, 덧셈 시간이 빠른 덧셈기를 선택하여 사용하는 것이 중요하다.

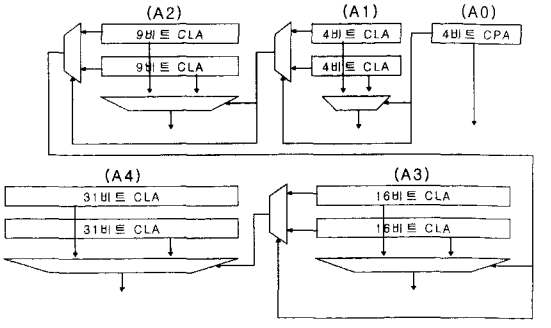


그림 12. 최종 덧셈기의 구조
Fig. 12. Final adder.

여기서는 캐리선택 덧셈기[6]와 캐리예견 덧셈기[1]를 혼합하여 사용하였는데, 캐리선택 덧셈기는 덧셈을 여러 부분으로 잘게 쪼개어 carry 입력이 있는 경우와 없는 경우 모두를 계산한 후, 나중에 carry 입력이 들어왔을 때 올바른 결과를 선택하는 것이다. 그림 12에서 본 연산기의 최종 덧셈기의 구조를 볼 수 있다.

그림 12에서 월레스 트리의 임계 경로가 아닌 LSB 4비트 부분은 월레스 트리의 연산이 끝나기 전에 덧셈을 시작할 수 있다. 월레스 트리의 임계 경로 연산이 끝난 후 임계 경로의 입력이 최종 덧셈기로 들어오면, 그 입력을 덧셈하고, 임계 경로의 캐리에 따라 나머지 부분의 덧셈결과를 선택하여 최종 출력을 구한다. 따라서 지연 시간은 월레스 트리의 임계 경로의 출력을 입력으로 받는 최종 덧셈기 블록의 덧셈 지연시간과 그 덧셈이 끝난 후, 결과를 MUX로 선택하는 데 걸리는 지연시간을 합한 만큼이 최종 덧셈기의 지연시간이 된다. 여기서, 64비트를 작은 단위로 쪼개면 덧셈 시간은 단축되나, 임계경로에 MUX가 많이 들어가므로 지연시간이 오히려 증가할 수 있다. 따라서 임계 경로의 위치에 따라 최종 덧셈기를 적절히 분할하여 설계하는 것이 중요하다.

Ⅳ. SIMD MAC 연산기의 설계과정

1. 하드웨어 구현 방법

본 논문에서 설계한 곱셈 누적 연산기는 SIMD 연산을 수행해야 하므로 일반적인 곱셈 누적 연산기와 부분곱 생성 방식이 다르고, 월레스트리에서의 캐리 조절 등이 추가되어 하드웨어를 구현하기 전에 이 부분에 대한 시뮬레이션이 필요하였다. 따라서 제안된 하드웨어 구조를 C program으로 작성하여 10만개의 test vector를 생성하였고, 이를 SUN SPARC processor에서 곱셈 후 누적 연산한 결과와 비교하였다. 이를 통하여 월레스트리에서의 캐리 조절을 해야 할 위치와 booth encoding, 부분곱 생성 방식 등에 대한 자세한 구조를 확정하였다.

C 언어를 이용한 시뮬레이션을 거친 후, Verilog HDL을 사용하여 게이트 level로 하드웨어를 기술하였다. 기술된 하드웨어는 무부호 32비트, 부호 32비트, 16비트 SIMD 연산 각각에 대하여 10만개씩, 총 30만개의 test vector를 가지고 Cadence Verilog-XL으로 시뮬레이션 하여 기능을 검증하였다.

기능 검증이 된 HDL code를 Synopsys Design Analyzer에서 컴파일 하여 합성하였다. 합성 후 면적과 지연시간을 확인하여 합성 결과가 목표에 미달한 경우, 임계경로 분석을 통하여 Verilog code를 수정함으로써 일부 구조를 변경하였으며, 변경된 Verilog code는 다시 기능 검증을 거쳐 재합성하는 과정을 거쳤다. 또한 합성시에 tool에서의 최적화 방식에 따라서도 지연시간과 면적이 달라지므로 구조 변경과 적절한 최적화 방법을 선택하여 합성을 하였다. 합성과 최적화에는 boolean optimization과 boundary optimization 기법을 적용하였다.

합성된 회로는 netlist를 Verilog 형식으로 출력한 후, 지연시간 정보를 sdf(standard delay file) 형식으로 저장하여 지연시간을 포함한 시뮬레이션을 다시 수행하였다. 이때는 초기 기능 검증에 사용된 test vector를 다시 사용하여 합성된 회로가 원래 작성된 Verilog 기술과 동일한 동작을 수행할 수 있는가를 검증하였다. 합성된 회로는 30만개의 test vector에 대하여 모두 올바른 동작을 보여주었다.

그림 13은 이러한 과정을 설계 흐름도로 나타낸 것이다.

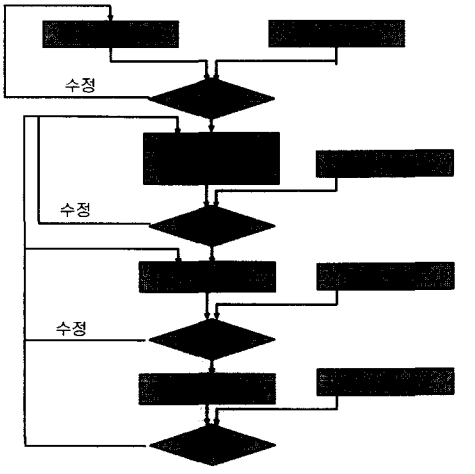


그림 13. SIMD MAC 연산기의 설계 흐름도
Fig. 13. Design flow of SIMD MAC unit.

2. 기능 검증

본 논문에서 설계된 SIMD MAC 연산기는 세가지의 동작 모드를 지원한다. 따라서 기능 검증은 세가지 동작 모드에 대하여 각각 10만개씩의 test vector로 수행

되었다.

검증을 위한 C 프로그램을 작성하여 test vector를 발생하였고, 검증 프로그램은 32비트 x, 32비트 y 입력 데이터와 곱셈 후 뺄셈 연산인지 곱셈 후 누적 연산인지를 나타내는 1비트 addsub 입력 데이터를 생성한다. 또한 각각의 입력 데이터에 해당하는 올바른 결과 데이터도 생성한다. 결과 데이터는 Sparc System에서 곱셈 기능을 이용하여 생성된다. addsub 이외의 동작 모드를 결정하는 조절신호들은 Verilog test block에서 입력하도록 하였다. 각각의 x, y test vector는 두 개의 16비트 random number를 조합하여 만들어지며, addsub 신호는 16비트 random number를 modulo-2연산하여 1비트를 얻어내는 방법으로 작성되었다. 64비트 accumulator register는 C 프로그램 상에서 64비트의 unsigned long long과 long long int 변수로 모델링 되었다.

본 기능검증에서는 위와 같은 방법으로 32비트 부호 연산, 32비트 무부호 연산, 16비트 SIMD 연산 각각에 대하여 별도의 프로그램을 작성하여 사용하였다. 각 프로그램으로 생성된 입력 벡터는 Verilog test block으로 입력되어지고, 시뮬레이션을 통하여 Verilog code에 의한 출력값을 만들어내게 된다. 최종적으로 Verilog에 의한 출력과 C 프로그램으로 수행된 출력을 각각 파일로 저장하여 두 파일을 비교함으로써 설계된 하드웨어가 올바른 연산을 하는가를 검증하였다. 같은 test vector를 가지고 합성된 netlist에도 동일한 과정을 거쳐 합성된 회로가 원래의 회로와 동일한 동작을 함을 확인하였다. 설계된 회로는 30만개의 test vector에 대하여 올바른 연산을 수행하였다.

30만개의 test vector가 본 연산기를 올바로 검증할 수 있는 정도의 양인지를 알기 위하여, Sysnopsys의 ATPG(Automatic Test Pattern Generate)를 참고로 이용하였다. ATPG는 설계된 회로의 각 node의 fault를 검출할 수 있도록 test vector를 자동으로 작성하여 주는 것으로, 기능 검증과는 관련이 없지만, ATPG로 생성된 test vector가 몇 개인가를 알면, 몇 개의 test vector로 본 연산기를 완전히 test할 수 있을 것인가를 간접적으로 알 수 있게 된다. ATPG 결과 본 연산기의 모든 node를 test 하는 데에 소요된 test vector 수는 216개로 나타났다. 이때의 fault coverage는 99.5%이다. 즉, 216개의 test vector로 본 연산기의 모든 node에서 발생할 수 있는 fault의 99.5%를 찾아낼 수 있다는 것이다. 기능 검증에 사용된 test vector는 30만개로 이에 비하면 월등히 많은 수이다. 비록 test vector가 random으로 발생되었지만, 본 연산기를 검증하는 데에는 충분한 수준이라고 생각된다.

V. SIMD MAC 연산기의 성능 평가

1. 하드웨어 합성 결과

하드웨어는 삼성 0.35㎛ 공정 standard cell library를 이용하여 수행되었으며, 32비트 곱셈 누적 연산기 하나의 면적은 combinational 회로의 면적이 21,800, sequential 회로의 면적이 811, 블록간의 연결을 위한 면적이 1,036, 총 23,646으로 측정되었다. 이 면적은 합성 라이브러리의 기본 NAND cell의 면적을 1.0으로 보았을 때의 수치이다. NAND 게이트 하나에 트랜지스터가 4개 들어있으므로, 대략적인 트랜지스터 수는 94,584개라 할 수 있다. 하지만, Synopsys에서는 같은 게이트라 하더라도 fanout의 크기에 따라 다른 크기의 트랜지스터를 사용하고, 면적 수치도 이에 따라 달라지므로, 이 트랜지스터 수가 정확한 것이라고 보기는 힘들다. SIMD 곱셈 누적 연산기는 이러한 32비트 연산기 두 개를 병렬로 연결하여 구현되므로, 약 20만개의 트랜지스터로 구성되어 있다고 할 수 있다. 이 면적 중 약 7,284 정도(대략 29,000개 가량의 트랜지스터)가 SIMD를 구현하는 데 소모되었고, 이것은 전체의 15%를 차지하는 것으로 나타났다. 이 SIMD area overhead는 주로 두 개의 누적 저장기와 누적 덧셈기를 사용함으로써 발생한다. 만약 64비트의 누적 저장기와 누적 덧셈기를 32비트 두 부분으로 분할하여 사용할 수 있도록 한다면 면적 overhead를 더욱 줄일 수 있지만, 이렇게 하면 누적 저장기에서의 오버플로우가 발생할 가능성이 높아지므로 두 개의 64비트 누적 저장기와 누적 덧셈기를 사용하는 방법으로 설계를 하였다.

총 지연시간은 3.0V, 85℃, 최악 공정 조건에서 12.05ns로, 3.3V, 25℃, 표준 공정 조건에서는 7.91ns로 측정되었다. 누적 저장기의 setup time을 고려한 최대 동작 주파수는 최악조건에서 80MHz이고, 표준조건에서는 125MHz이다.

2. 성능 비교

본 논문에서 설계한 SIMD 곱셈 누적 연산기는 최악조건에서 80MHz의 동작 주파수를 갖는다. 한 사이클에

최대 4개의 16비트 연산 혹은 두 개의 32비트 연산이 가능하므로, 이 연산기는 16비트 데이터의 경우 초당 최대 3억2천만번의 곱셈 누적 연산을 처리할 수 있다. 표 4는 16비트 데이터를 기준으로 본 논문에서 설계한 SIMD 곱셈 누적 연산기를 다른 곱셈 누적 연산기와 비교한 결과를 보여준다. 내장형 시스템에서 사용하는 멀티미디어 데이터는 16비트인 경우가 많으므로 16비트 데이터를 기준으로 성능을 비교하였다. 본 논문에서 제시한 MAC 연산기를 제외한 모든 연산기들은 full custom 방식으로 설계되었다.

표 4. 여러 MAC 연산기들의 성능 비교
Table 4. Performance of MAC units.

	SIMD MAC	TMS320 C5X[7]	32bit MAC[8]	AltiVec MAC
freq	80MHz	50MHz	100MHz	500MHz
공정	0.35㎛ 표준셀	0.8㎛ 풀커스텀	0.35㎛ 풀커스텀	0.18㎛ 풀커스텀
Latency	1 cycle	3 cycle	-	3 cycle
Datapath	64bit	32bit	32bit	128bit
SIMD	4 16bit 2 32bit	X	X	16 8bit 8 16bit 4 32bit
성능수치	320Mops	50Mops	100Mops	4000Mops

본 연산기는 AltiVec과 같은 고성능 범용 마이크로프로세서에 내장된 것보다는 성능이 떨어지지만, 내장형이나 DSP 프로세서들에 비하여는 높은 성능을 보이는 것을 볼 수 있다. AltiVec은 상용의 고성능 SIMD 연산기들 중에서도 가장 성능이 좋은 것으로서 128비트의 datapath를 full custom dynamic circuit으로 구현하였고, 3 stage의 파이프라인을 사용하여 동작 주파수가 빠르므로, 성능 차이는 구조적인 문제보다는 공정과 회로 설계 방식에 기인한 것으로 보인다.

또한 본 연산기는 한 사이클에 연산을 완료하는 구조이므로, 하나의 연산을 2~3 사이클의 파이프라인 방식으로 구현된 다른 연산기에 비하여 파이프라인 제어와 입력 데이터의 상호 의존 관계(dependency)에 따른 성능 저하 현상이 없다. 따라서 표에 나타난 비교자료보다 실제 성능은 좀 더 향상되었다고 볼 수 있다.

VI. 결 론

본 논문에서는 내장형 프로세서에 적합한 SIMD MAC 연산기를 설계하였다. 두 개의 16비트 MAC 연산을 동시에 할 수 있는 32비트 MAC 연산기를 두 개 내장하였고, 32비트 SIMD MAC 블록은 기존의 32비트 곱셈기의 구조를 크게 변경하지 않고도 설계가 가능하도록 하였다. 일반적인 곱셈기의 각 블록들을 두 부분으로 적절히 분할하여 사용할 수 있도록 함으로써 SIMD 연산을 구현하는 데에 필요한 overhead를 최소화 하였다. 각각의 16비트 곱셈 결과에 대하여 누적 연산을 할 수 있도록 32비트 SIMD MAC 연산 블록 하나에 두 개씩의 누적 연산기와 최종 덧셈기, 누적 저장기를 내장하였다.

또한 누적 덧셈기의 캐리전파를 없애 지연시간을 단축하였다. 곱셈의 결과가 최종 덧셈기로 들어가기 전에 합벡터와 캐리 벡터에 누적 저장기의 값을 더하여 새로운 합벡터와 캐리 벡터를 만들고, 이를 최종 덧셈기로 입력하여 최종 결과를 생성하는 방법으로 64비트의 누적 덧셈으로 인한 시간 지연을 상당부분 제거할 수 있었다.

SIMD MAC 연산기는 C 시뮬레이션으로 세부적인 구조를 확정함으로써 하드웨어 디버깅에 들어가는 시간을 단축하였으며, Verilog HDL로 하드웨어를 설계하여, 0.35㎛ 표준 셀 라이브러리를 이용하여 합성하였다. 최악 조건에서 80MHz, 표준 조건에서는 125MHz의 동작 주파수를 보이며, 80MHz로 동작할 때 초당 3억 2천만개, 125MHz로 동작하는 경우에는 5억번의 16비트 누적 연산을 수행할 수 있다. 이는 최신의 범용 고성능 마이크로 프로세서에 내장된 SIMD MAC 연산기보다는 떨어지는 성능이지만, 기존의 내장형 프로세서에 비하여는 월등한 성능이다. 반면에 회로의 크기는 최신의 SIMD 연산기들에 비하여 매우 작으므로, 멀티미디어 데이터를 처리해야 하는 중성능 이상의 내장형 프로세서에 내장하기 적합한 연산기라고 할 수 있다.

또한, 32비트 SIMD MAC 연산 블록 하나는 16비트 SIMD 연산 이외에도 32비트 부호/무부호 연산을 지원할 뿐 아니라, SIMD 기능이 없는 것에 비하여 15% 정도의 면적 증가만으로 구현 가능하므로, 기존의 내장형 프로세서에서 32비트 곱셈기 회로를 32비트 SIMD 연

산 블록 하나로 대체하는 것도 가능할 것이다.

멀티미디어 데이터를 처리하는 데 있어서 MAC 연산만이 요구되는 것이 아니므로, SIMD 방식의 ALU와, 16 혹은 32비트의 데이터를 SIMD 처리에 알맞도록 64비트 레지스터에 정렬할 수 있는 pack/unpack연산기를 설계하여 본 MAC 연산기와 통합하는 연구가 진행된다면, SIMD 방식의 장점을 좀 더 잘 살릴 수 있는 내장형 SIMD DSP 연산기를 구현할 수 있을 것이다.

참 고 문 헌

[1] Israel Koren, *Computer Arithmetic Algorithms*, Prentice-Hall, pp. 73-77, 86-91, 99-123, 1993

[2] Martin S. Schmookler, Michael Putrino, Charles Roth, Mukesh Sharma, Anh Mather, Jon Tyler, Huy Van Nguyen, Mydung N. Pham, and Jeff Lent, "A Low-power, High-speed Implementation of a PowerP^{CTM} Microprocessor Vector Extension", Computer Arithmetic, 1999. Proceedings. 14th IEEE Symposium on, 1999.

[3] 이용석, "고성능 마이크로프로세서 곱셈기 구조", 정보통신학술지원국 비디오 강좌시리즈, http://mpu.yonsei.ac.kr/Lecture/정통부_video_lect.htm, 1998

[4] Gensuke Goto, Atsuki Inoue, Ryoichi Ohe, Shoichiro Kashiwakura, Shin Mitarai, Takayuki Tsuru, and Tetsuo Izawa, "A 4.1-ns Compact 54×54-b Multiplier Utilizing Sign-Select Booth Encoders", IEEE Journal of Solid-State Circuits, Vol. 32, No. 11, November, 1997.

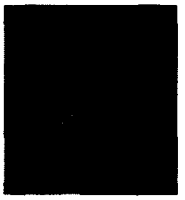
[5] Mark R. Santoro, "SPIM : A Pipelined 64x64-bit Iterative Multiplier", IEEE Journal of Solid-State Circuits, Vol. 24, No. 2, April, 1989.

[6] 이용석, "60MHz Clock 주파수의 IEEE 표준 Floating Point ALU", 전자공학회 논문지, 제28권, A편, 제11호, 1991년 11월

[7] *Buyer's Guide to DSP Processors*, Berkeley Design Technology Inc. pp. 351-361, 449-472, 1994.

[8] Bum-Sik Kim, Dae-Hyun Chung, and Lee-Sup Kim, "A New 4-2 Adder and Booth Selectorfor Low Power MAC unit", Proceeding of International Symposium on Low Power Electronics and Design, Monterey, pp. 100-103, 1997.

저 자 소 개

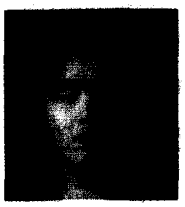


洪 仁 杓(正會員)

1976년 5월 28일생, 1999년 2월 연세대학교 전자공학과 공학사, 2001년 2월 연세대학교 전기컴퓨터공학과 공학석사, 2001년 3월~현재 연세대학교 전기전자공학과 박사과정. <주관심분야 : 마이크로프로세서 설계, VLSI 설계, 고성능 연산기 설계>

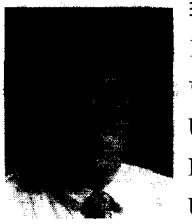
鄭 愚 暻(正會員)

1974년 2월 8일생, 1996년 2월 연세대학교 전자공학과 공학사, 1998년 2월 연세대학교 전자공학과 공학석사, 1998년 3월~현재 연세대학교 전기전자공학과 박사과정. <주관심분야 : 마이크로프로세서 설계, VLSI 설계, 부동소수점 연산기 설계, 고성능 연산기 설계>



鄭 在 元(正會員)

1972년 11월 10일생, 1999년 2월 연세대학교 전자공학과 공학사, 2001년 2월 연세대학교 전기컴퓨터공학과 공학석사, 현재 LG전자 액세스 망연구소에서 DSLAM 장치 개발. <주관심분야 : Computer Architecture, VLSI 설계, SOC, 고성능 연산기 설계>



李 溶 錫(正會員)

1950년 10월 23일생, 1973년 연세대학교 전자공학과 공학사, 1977년 2월 University of Michigan Electrical Engineering 공학석사, 1981년 2월 University of Michigan Electrical Engineering 공학박사, 1993년~현재 : 연세대학교 전기전자공학과 교수. <주관심분야 : 마이크로프로세서 설계, VLSI 설계, DSP 프로세서 설계, 고성능 연산기 설계>