논문 2004-41SD-11-16

32비트 3단 파이프라인을 가진 RISC 프로세서에 최적화된 Multiplier 구조에 관한 연구

(A Study on Multiplier Architectures Optimized for 32-bit RISC Processor with 3-Stage Pipeline)

정 근 영*, 박 주 성**, 김 석 찬***

(Geun Young Jeong, Ju Sung Park, and Suk Chan Kim)

요 약

본 논문에서는 32비트 3단 파이프라인을 가진 RISC 프로세서에 최적화된 곱셈기 구조의 연구에 대해 다룬다. 대상 프로세서인 ARM7은 3단의 파이프라인 구조로 되어 있으며 이 프로세서의 곱셈기는 파이프라인 상의 실행 단계에서 최대 7사이클이 소요된다. 내장된 곱셈기는 기능적으로 부스 알고리즘을 적용하여 32×32 곱셈 연산과 덧셈 연산을 하여 64비트 결과를 낼 수 있는 MAC(Multiplier-Accumulator) 구조로 되어 있으며 6가지 세부 명령어를 실행할 수 있다. ARM7의 파이프라인 및 ALU와 shifter 구조에 적합한 radix4-32×8 및 radix4-32×16 과 radix8-32×32의 곱셈기 구조를 비교 분석하였으며 면적, 사이클 지연시간, 수행 사이클 수를 성능 기준으로 최적화된 곱셈기를 결정하여 설계하였다. 프로세서 코어에 내장된 곱셈기의 동작을 검증하기위해 다양한 오디오 알고리즘을 이용하여 시뮬레이션을 수행하였다.

Abstract

This paper describes a multiplier architecture optimized for 32 bit RISC processor with 3-stage pipeline. The multiplier of ARM7, the target processor, is variably carried out on the execution stage of pipeline within 7 cycles. The included multiplier employs a modified Booth's algorithm to produce 64 bit multiplication and addition product and it has 6 separate instructions. We analyzed several multiplication algorithm such as radix4-32×8, radix4-32×16 and radix8-32×32 to decide which multiplication architecture is most fit for a typical architecture of ARM7. VLSI area, cycle delay time and execution cycle number is the index of an efficient design and the final multiplier was designed on these indexes. To verify the operation of embedded multiplier, it was simulated with various audio algorithms.

Keywords: Multiplier, RISC, ARM7, Booth.

I. 서 론

디지털 신호처리 시스템에서 곱셈기는 중요한 역할을 하고 있다. 곱셈연산은 오디오/비디오 변환 및 상관관계처리, 다양한 필터, 범위 측정 등 대부분의 알고리즘에 이용되고 있다. 최근 멀티미디어 처리의 복잡화로 곱셈기의 처리능력이 차지하는 비중은 증가하고 있다. 활용의 분야가 매우 다양하기 때문에 그 목적 및 내장된 시스템에 따라 곱셈기의 형태는 다양하게 존재하며 곱셈 연산은 하드웨어로 처리되든 소프트웨어로 처리되든 비교적 복잡한 연산이다.

3단 파이프라인 구조인 RISC 프로세서의 경우 명령어의 실행 영역은 파이프라인 중 한 단계에 제한되므로 시간지연이 길게 발생하는 곱셈명령의 실행은 다중 사이클을 요구하게 된다[1,2]. 이러한 사이클 지연으로 인해 인접한 명령어도 동시에 지연되므로 명령어 실행 사이

* 학생회원, ** 평생회원, *** 정회원
부산대학교 전자공학과
(Dept. of Electronics Engineering, Pusan National University)

※ 본 연구는 반도체설계교육센터(IDEC)의 설계지원과 소프트웨어진흥원의 IT-SOC 핵심설계인력양성사업 및 한국산업기술재단의 지역혁신인력양성사업의 연구비 지원으로 수행되었음.
접수일자: 2004년7월20일, 수정완료일: 2004년10월28일

클 수가 성능에 중요한 영향을 미친다. 그러나 사이클 수를 줄이기 위해 곱셈기의 알고리즘을 무리하게 압축하면 최대 시간지연이 곱셈기에서 발생하여 프로세서 전체의 성능을 떨어뜨린다. 따라서 프로세서 내에서 최대 지연시간(maximum delay time)과 곱셈기의 최대 사이클 수를 균형 있게 배분하는 것이 중요하다.

곱셈은 기본적으로 크게 두 가지 작용을 한다. 하나는 최종 결과가 나올 때까지 더해지는 피승수의 배수인 부분곱의 생성이고 다른 하나는 생성된 부분곱을 누산하는 것이다. 곱셈기 자체의 성능향상을 꾀하는 가장 이상적인 방법은 부분곱 생성 시간을 줄이면서 부분곱 누산 횟수를 줄이는 것이다[3]. 그러나 누산 횟수를 줄인 알고리즘을 채택할 경우 선행연산인 부분합의 생성에 시간지연이 가중되므로 곱셈기를 제외한 프로세서 내의 최대 시간지연과 곱셈기 자체의 시간지연을 비교하여 알고리즘을 결정할 필요가 있다.

곱셈기를 최적화할 대상 프로세서인 ARM7은 대표적인 32비트 RISC 프로세서이다. 3단 파이프라인에 32비트 ALU와 베럴쉬프터(Barrel shifter)를 포함하고 있다. 기존 설계상에 내장된 곱셈기는 변형된 부스알고리즘이 적용된 radix-4 32×8 구조이다. 한 사이클에 8비트씩 연산하며 최종 결과를 얻기까지 최대 7 사이클이 소요된다[1].

본 논문에서는 상위에 기술된 구조와 기능적으로 동일하게 설계된 ARM7 호환 프로세서에 다양한 곱셈기를 적용하여 논리회로 합성 및 시뮬레이션을 수행하였다. 최종적으로 곱셈기 구조와 결과를 비교 분석하여 최적화된 구조를 검증하였다.

Ⅱ. 대상 프로세서의 구조 분석

곱셈기가 내장될 프로세서의 중요 기능블럭으로는 그림 1에 나타낸 바와 같이 ALU, 곱셈기, 레지스터 뱅크, 명령어 디코더, 컨트롤 로직, 명령어 파이프라인 등이 있다. 3개의 내부 데이터 버스는 32비트로 구성되어 있다. 베럴쉬프터(barrel shifter)와 ALU가 순차 연결되어 쉬프트(shift) 연산과 논리/산술 연산을 동시에 수행할 수 있다. 곱셈연산에서는 부분합들이 ALU를 통해 최종 값으로 더해져 레지스터에 저장된다[1,2].

대상 프로세서는 패치, 디코드, 실행으로 단순하게 구성된 3단계의 파이프라인을 내장하고 있다. 대부분의 명령어는 단일 사이클에 수행되나 특정 명령어의 수행에는 다수의 사이클을 필요로 한다. 수행 사이클은 1~

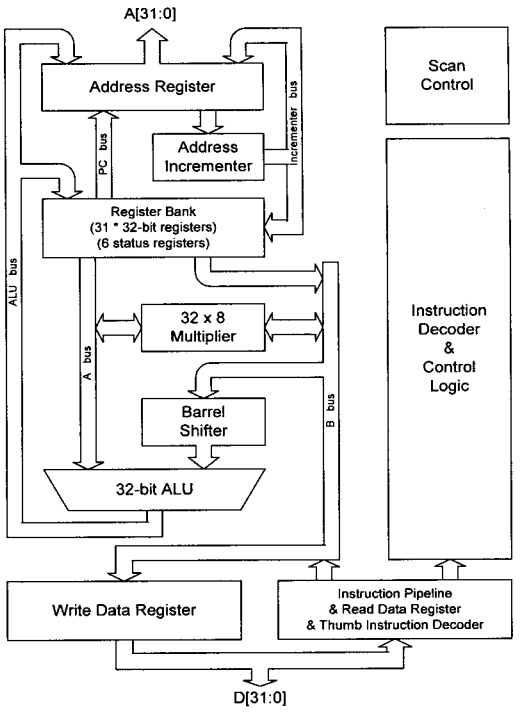

그림 1. 설계대상 프로세서의 구조
Fig. 1. Block diagram of target processor.

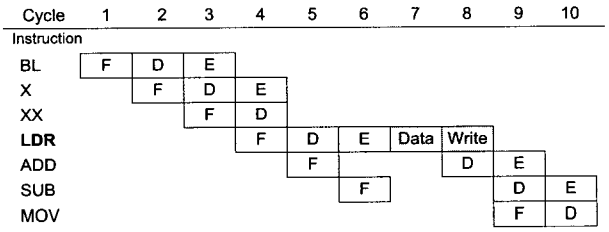

그림 2. 명령어 파이프라인 상태
Fig. 2. Instruction pipeline operation.

17 사이클로 가변적이다. 메모리로부터 데이터를 읽어내는 LDR 명령어의 파이프라인 단계를 그림 2에 나타내었다. LDR 명령어는 실행단계에서 execution, data, write 순으로 3 사이클에 걸쳐서 수행된다. 이러한 다중 사이클의 실행 뒤에 따르는 명령어는 자동적으로 디코더 이후의 과정이 지연되게 된다. 두 사이클이 지연되므로 LDR 명령어 이후 ADD 명령어와 SUB 명령어가 지연되고 다음 MOV 명령어부터 정상적인 파이프라인 동작이 이루어진다. 곱셈명령어인 MUL의 경우 최소 2에서 최대 7 사이클이 소요되므로 실행 사이클 수를 줄여 곱셈 연산시간을 단축하는 알고리즘이 요구된다[2].

Ⅲ. 곱셈기의 구조와 설계

3.1 곱셈기의 기능적 구조

곱셈기는 승수와 피승수인 2개의 32비트 값을 입력

받아서 부분곱을 단계적으로 누산하여 곱셈결과인 sum 32비트와 carry 32비트로 분리하여 출력한다. 부분곱의 누산 결과인 sum과 carry는 프로세서에 포함된 고속 ALU와 쉬프터를 통해 합산되어 최종 64비트의 결과 값을 얻게 된다. 64비트 결과 값은 상위와 하위 32비트씩 각각의 레지스터에 저장된다. 그림 3에 이러한 부스 알고리즘이 적용된 곱셈기의 구조도를 나타내었다.

그림 3에 나타낸 세부 동작은 radix-4(이후 R4) 구조에 맞춰진 것이다. 한 사이클에 승수를 8비트씩 부스 부호화하여 4개의 피승수 부분곱이 누산된다[2]. Radix-8(이후 R8) 알고리즘으로 변환할 경우 쉬프트 값이 16비트로 늘어나고 부스 부호기와 CSA (Carry Save Adder)의 내부구조를 변경해야 된다[3,4]. 그러나 전반적인 구조 크게 달라지지 않는다. 덧셈연산의 동시실행, 64비트 결과출력, 부호연산 여부에 따라 6가지의 명령어로 구별된다. 명령어에 따른 기능을 나타내면 표 1과 같다.

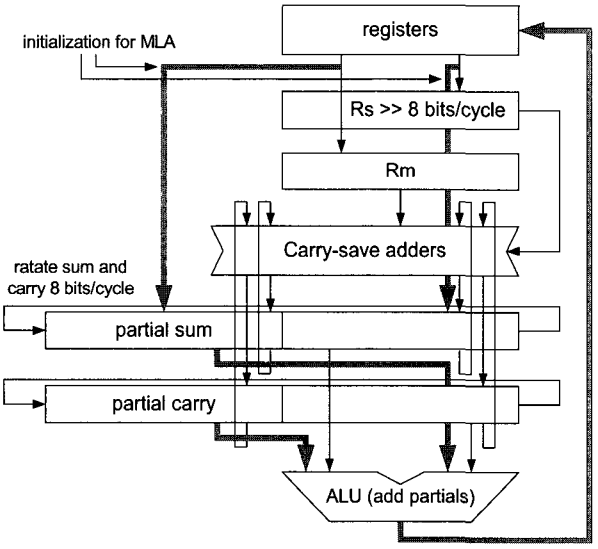

그림 3. ARM7의 곱셈기 구조
Fig. 3. Architecture of ARM7 multiplier.

표 1. Multiply 명령어별 기능
Table 1. Function of each multiplier instruction.

Mnemonic	Meaning
MUL	Multiply (32bit result)
MLA	Multiply-Accumulate (32bit result)
UMULL	Unsigned Multiply Long
UMLAL	Unsigned Multiply-Accumulate Long
SMULL	Signed Multiply Long
SMLAL	Signed Multiply-Accumulate Long

3.2 Radix-4 32×8 곱셈기 구조

R4 구조의 부스 알고리즘은 Wallace tree에 들어가는 부분합의 수를 줄이기 위한 것으로 변형된 부스 알고리즘이 사용되었다[5]. 일반적인 변형 부스알고리즘에서는 x_0(승수의 최하위 비트)의 하위에 0값을 가지는 x_{-1}을 두어 3비트 묶음으로 부스 부호화(recode)의 첫 단계가 시작된다. 그리고 상위로 2비트씩 쉬프트하면서 1비트씩 중첩되어 부호화가 이루어진다. 이러한 동작은 그림 4에 나타내었고 부호화 논리표는 표 2에 나타내었다. 그림 4에서 보면 첫 번째 부분곱을 생성하는 CSA1의 최하위 비트는 앞서 설명한 경우와 달리 x_0인데 이러한 방법은 x_0의 값에 따라 피승수의 값을 감산하는 보정연산이 필요하다 그러나 별도의 추가 회로 없이 x_0 값이 1인 경우 피승수의 반전된 값을 CSA1의 carry 초기값으로 입력하고 아닌 경우는 0 값을 입력하면 된다.

설계된 곱셈기의 구조인 R4-32×8에서 8이 의미하는 바는 한 사이클에 승수로부터 8비트씩 받아들여서 부호화한다는 것이다. 실제로는 9비트를 읽어서 4번의 부스 부호화를 하며 4개의 부분곱을 누산해야 된다. 즉 32비트의 곱셈 연산을 하기 위해서는 4번의 실행 사이클이 필요하다. 그리고 8비트 부스 부호화를 통해서 만들어진 부분곱은 한 사이클 내에 CSA의 입력으로 들어가게 되고 계산된 결과는 sum과 carry로 나뉘어서 저장된다. 이러한 과정을 4번 반복하면 32비트 피승수의 곱

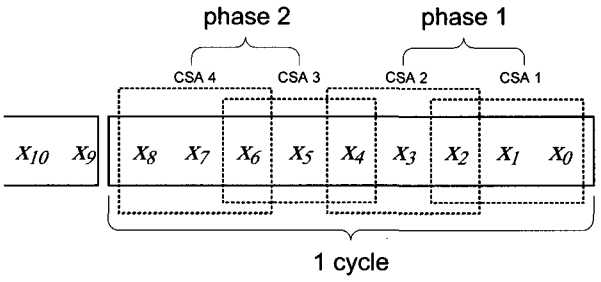

그림 4. R4-32×8 구조의 부스 부호화 동작
Fig. 4. Booth encoding operation of R4-32×8 architecture.

표 2. R4 부스 부호화의 피승수 배수 선택
Table 2. Partial product selection of R4 Booth's encoding.

Multiplier bits	Selection	Multiplier bits	Selection
000	0Y	100	-2Y
001	+1Y	101	-1Y
010	+1Y	110	-1Y
011	+2Y	111	0Y

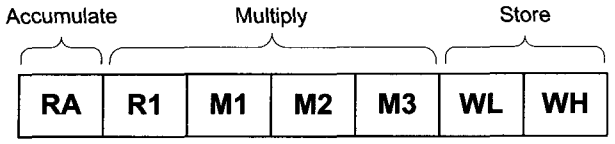

그림 5. R4-32×8 곱셈기의 파이프라인 동작
Fig. 5. Pipeline operation of R4-32×8 multiplier.

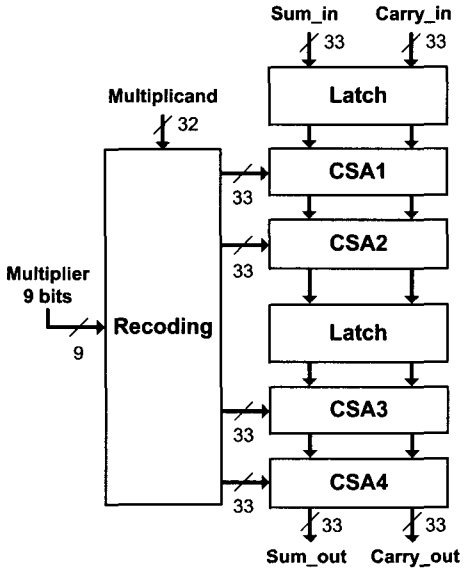

그림 6. R4-32×8 곱셈기의 구조
Fig. 6. R4-32×8 multiplier architecture.

비트이므로 32비트 결과만을 저장할 때는 한 사이클이 소요되고 64비트 결과를 저장할 때는 두 사이클이 소요된다. 만약 덧셈연산(accumulation)이 필요할 경우 선처리로서 덧셈 값이 저장되어야 하기 때문에 가장 첫 사이클에 한 사이클이 소요된다. 결론적으로 SMLAL의 명령어의 경우에 7 사이클이 걸린다. 이를 그림 5에 나타내었다. 그러나 반드시 7번의 실행 사이클을 가지는 것은 아니다. 승수 값의 크기에 따라 곱셈실행 부분은 1~4 사이클로 가변적이다. 즉 선 종료가 가능하며 중간 값 보정은 최종 결과 값 저장 시에 처리된다[6-8].

한 사이클에 4번의 부스 부호화를 하므로 곱셈기 구조는 연산부분에 4개의 32비트 CSA (Carry Save Adder)가 필요하다. 이들은 2개씩 순차 연결되어 있으며 한 사이클에 두개의 위상이 존재하므로 사이에 래치를 이용하여 중간 값을 저장한다. 곱셈기 R4-32×8 구조에서 실제로 가장 스피드와 면적에서 중요한 부분을 차지하는 부분의 구조를 그림 6에 나타내었다.

3.3 Radix-4 32×16 곱셈기 구조

R4-32×16 구조의 곱셈기는 R4-32×8 곱셈기와 부스 부호화 및 CSA 구조는 동일하게 R4 방식을 사용한다.

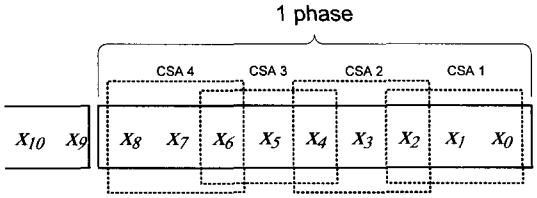

그림 7. R4-32×16 구조의 부스 부호화 동작
Fig. 7. Booth encoding operation of R4-32×16 architecture.

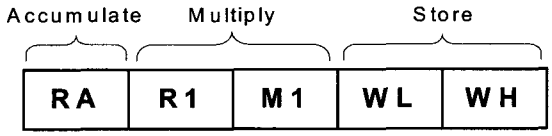

그림 8. R4-32×16 구조의 파이프라인 동작
Fig. 8. Pipeline operation of R4-32×16 architecture.

다른 점은 곱셈실행에 4 사이클이 소요되기 때문에 이를 줄이기 위해서 부스 부호화를 한 사이클에 16비트씩 하도록 구조를 바꾼 것이다. 이렇게 처리하면 곱셈연산 실행에 2 사이클 소요된다. 첫 위상에서의 부스 부호화 순서를 그림 7에 나타내었다. 가장 긴 사이클을 가지는 명령어인 SMLAL와 UMLAL의 경우 그림 8과 같은 사이클을 가진다.

16비트씩 부스 부호화를 하게 되면 부분합의 수가 동시에 8개가 생성된다. 이 부분곱들을 동시에 더해 주기 위해서는 CSA의 개수 또한 8개가 필요하게 되며 CSA 면적의 비중이 크므로 전체 곱셈기의 면적이 크게 증가한다. 따라서 두 위상에서 CSA를 4개씩 할당하는 것이 아니라 4개의 CSA를 가지고 위상마다 그것을 반복해서 사용하는 구조로 설계하였다. 각 위상 별로 CSA를 공유하므로 사이클 별로 부스 부호화된 값을 위상에 맞추어 분배하는 멀티플렉서가 추가된다. 이와 같이 변경된 구조를 그림 9에 나타내었다.

그림 6과 비교하면 32비트 CSA가 동일하게 4개가 사용되었고 33비트 래치 부분과 부스 부호화 부분의 멀티플렉서가 수정되고 추가되었다. 선 종료를 처리하는 부분과 중간 값 처리 부분이 추가적으로 축소되어 전체 면적에서는 R4-32×8과 크게 차이가 나지 않는다.

한 위상에서 처리되는 CSA의 개수가 4개이고 부스 부호화나 기타 경로의 지연은 거의 동일하기 때문에 R4-32×8 구조보다 사이클 당 지연시간이 길어지게 된다. 따라서 전체 사이클 수는 2 사이클 줄일 수 있으나 사이클 당 지연시간의 증가는 감수해야 된다.

그러나 CSA에 의한 지연시간 상승은 전체 지연시간에 비해 큰 비중을 차지하는 것은 아니다. 곱셈기의 사

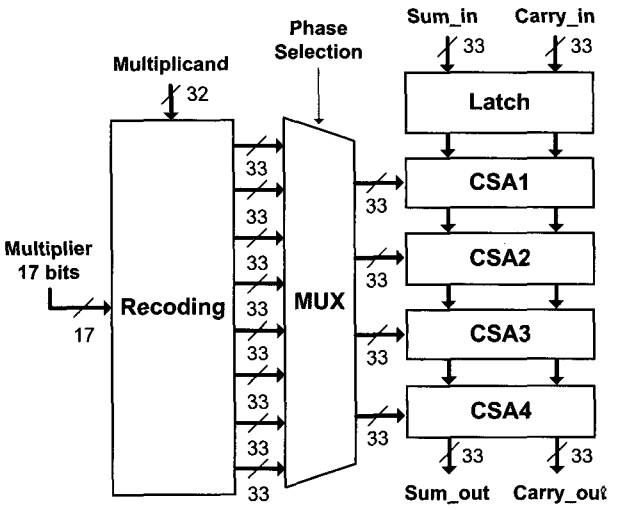

그림 9. R4-32×16 곱셈기의 구조
Fig. 9. R4-32×16 multiplier architecture.

이클 속도와 사이클 수는 내장될 프로세서의 구조 및 동작속도에 밀접하게 영향을 받으므로 곱셈기 자체의 성능과 프로세서 내의 곱셈연산 성능에는 차이가 있다.

3.4 Radix-8 32×32 곱셈기 구조

R8-32×32 구조는 부스 부호화 방법을 달리 하여 부분합의 수를 줄인다. R8 구조에서는 승수의 4비트 입력을 받아 16개의 비트 조합을 부호화하여 5가지(0, ±1, ±2, ±3, ±4)의 피승수 배수로 만들어진다[4]. 표 3에 R8 부호화의 피승수 배수를 나타내었다.

R8-32×32 구조에서는 부스 부호화를 4비트씩 하게 되므로 승수 32비트에 적용하면 전체 11개의 부분곱이 생성된다. 부분곱을 균등분할 하기 위해 승수를 확장하여 부분곱의 수를 12개로 늘리고 위상 별로 6개씩 두 부분으로 나누었다. 따라서 필요한 CSA의 개수는 6개가 되고 두 개의 위상으로 나누어 연산을 할 수 있다. R8 구조의 부스 부호화 과정을 그림 10에 나타내었다.

승수의 값에 상관없이 한 사이클 이내에 곱셈 연산을 수행하므로 가장 긴 사이클을 가지는 곱셈 명령어의 사이클 수를 4로 줄일 수 있다. 파이프라인에서 곱셈실행 사이클을 나타내면 그림 11과 같다.

R4 구조에서는 별도의 덧셈기가 필요 없지만 R8 구조에서는 부호화 과정에서 3의 배수 값이 필요하므로 CPA(Carry Propagate Adder)형태의 덧셈기가 포함된다. 피승수의 3배수는 한번의 쉬프트와 덧셈으로 만들 수 있다. R8 구조에서는 이와 같이 추가적인 덧셈기로 인해 부스 부호화과정의 지연시간을 크게 증가시켜 빠른 클럭을 사용하지 못하게 하는 문제가 있다. 이 문제

Multiplier bits	Selection	Multiplier bits	Selection
0000	0	1000	-4Y
0001	+1Y	1001	-3Y
0010	+1Y	1010	-3Y
0011	+2Y	1011	-2Y
0100	+2Y	1100	-2Y
0101	+3Y	1101	-1Y
0110	+3Y	1110	-1Y
0111	+4Y	1111	0

표 3. R8 부스의 피승수 배수 선택
Table 3. Partial product selection of R8 Booth's encoding.

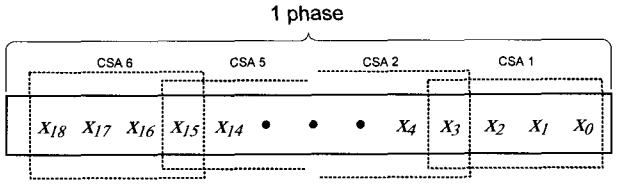

그림 10. R8-32×32 구조의 부스 부호화 동작
Fig. 10. Booth encoding operation of R8-32×32 architecture.

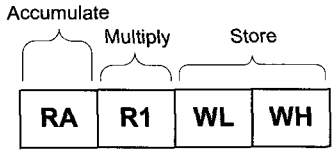

그림 11. R8-32×32 구조의 파이프라인 동작
Fig. 11. Pipeline operation of R8-32×32 architecture.

를 해결하기 위해서 partially redundant partial product 부스 알고리즘을 사용하기도 하지만 파이프라인 구조에서 보상해주는 부분에 추가적인 사이클이 필요하기 때문에 사용하기 어렵다[9]. 또한 게이트 단계(gate level)의 VLSI 설계에서는 고속의 덧셈기를 구현하는데 문제가 있다. CLA(Carry Lookahead Adder) 방식이나 carry-select adder를 사용할 수 있지만 트랜지스터 단계에서 설계된 고속 덧셈기에 비해 성능이 저하된다. 최종 설계된 덧셈기는 대상 프로세서에 포함된 ALU의 carry-select adder를 확장하여 구현하였다. R8-32×32 곱셈기 구조는 다음 그림 12에 나타내었다.

3 배수를 형성하는 부호기, 멀티플렉서 및 CSA가 추가 된 반면, 데이터를 저장하는 부분들은 줄어서 면적이 소폭 증가하였다. 부호화 과정에서 덧셈기의 지연시간을 줄이기 위해서 파이프라인의 실행단계 이전에 3배수 값을 연산하여 부호화 시에는 저장된 값을 불러오는 방식을 적용할 수 있다[9]. 그림 11과 같이 RA단계가 항상 존재하면 간단히 이러한 동작을 구현할 수 있으나 아닐 경우에는 제어회로와 레지스터 뱅크의 대폭적인

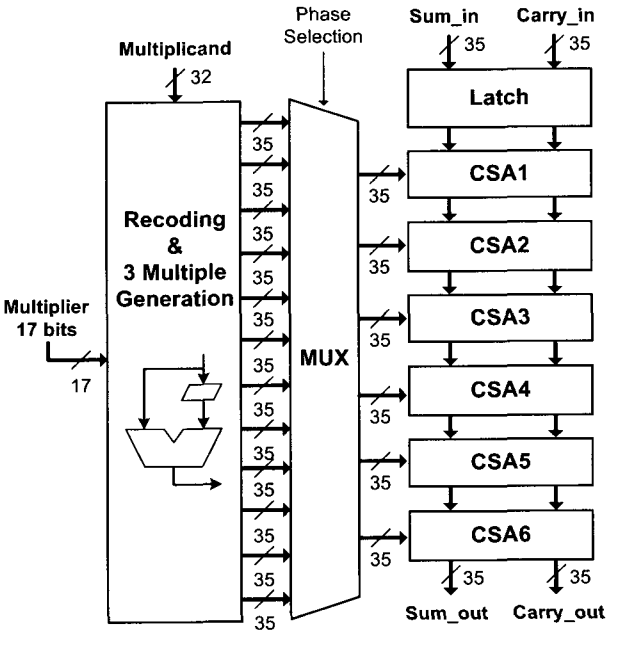

그림 12. R8-32×32 곱셈기의 구조
Fig. 12. R8-32×32 multiplier architecture.

수정이 요구된다. 그러므로 본 논문에서는 3 배수 선처리 과정은 다루지 않고 곱셈기 자체 구조를 중점으로 다루겠다. 참고적으로 R8-32×16 구조도 구현 가능하나 R4-32×16 구조와 비교하여 부스 부호화에서 시간 지연이 발생하며 동일한 사이클 수를 가지므로 비교검증에서 생략되었다.

Ⅳ. 곱셈기 설계 결과비교

4.1 곱셈기 자체 성능비교

설계된 세 가지 곱셈기 구조를 삼성의 0.5㎛ CMOS 라이브러리를 이용하여 Synopsys사의 design compiler를 통해 합성하였다. 합성할 경우 구조는 CSA의 수에 따라 달라지지만 면적의 차이는 20% 내외로 크게 차이가 나지 않았다. 부스 부호화 과정이 복잡해진 만큼 주변 중간 값 처리부와 결과 값 정렬을 위한 회로가 간소화되어 전체적으로 면적이 상쇄되는 부분이 많았다. 곱셈 연산은 다중 사이클에 수행되므로 각각의 구조에서 사이클 당 지연시간이 가장 긴 것을 선택하여 산출하였다. 곱셈기의 구조별로 면적(게이트 수), 사이클 당 시간지연, 사이클 수를 성능지수로 비교하였다. 결과는 전체 프로세서 core에 곱셈기를 내장하지 않고 곱셈기 자체를 timing simulation하여 분석하였다. 분석 결과는 표 4에 나타내었는데 성능지수 ATC의 괄호안 값은 성능비를 역수로 환산해서 백분율을 구한 것이다.

표 4. 곱셈기 자체 성능 비교
Table 4. Performance comparison of separate multipliers.

Multiply Method	Area (A) gates	Delay time/ Cycle (T)	Cycle (C)	A×T×C Ratio (%)
Radix-4 32×8	6374	10ns	7	1 (100)
Radix-4 32×16	7281	14.5ns	5	1.18 (85)
Radix-8 32×32	8114	19.7ns	4	1.43 (70)

표 5. 프로세서 코어에 장착된 곱셈기 성능비교
Table 5. Performance comparison of embedded multipliers.

Multiply Method	Area (A) gates	Delay time/ Cycle (T)	Cycle (C)	A×T×C Ratio (%)
radix-4 (32×8)	6374	20ns	7	1 (100)
radix-4 (32×16)	7281	20ns	5	0.82 (123)
radix-8 (32×32)	8114	>20ns	4	

R4-32×8 구조에 비해 나머지 두 구조는 파이프라인 상의 수행 사이클 수는 줄었으나 면적 및 사이클 당 시간지연이 큰 폭으로 증가하여 성능평가 기준인 면적과 속도측면으로 볼 때 각 85%와 70%로 성능이 떨어졌다.

4.2 곱셈기 장착 결과

모든 곱셈기가 그것이 내장 될 프로세서의 최대지연시간 이내에 동작하는 것으로 가정한다면 곱셈기의 사이클 당 지연시간은 프로세서의 최대 동작속도에 따른 지연시간으로 대체 될 것이다. 곱셈기가 내장될 대상 프로세서가 3단 파이프라인을 가지므로 곱셈기의 사이클 수와 프로세서 상에서 결과출력 사이클 수가 일치한다. 따라서 곱셈기의 속도측면은 연산완료까지의 사이클 수가 기준이 될 것이다. 설계된 ARM7 호환 프로세서의 최대 동작속도는 50Mhz이므로 사이클 당 최대 지연시간은 20ns이다. 제어신호 생성과 레지스터 출력 시간을 고려하면 연산처리를 14ns이내에 수행해야한다[8]. 그러나 R8 구조의 곱셈기는 19.7ns에 동작하므로 프로세서에 장착이 불가능하다. 나머지 R4 구조의 곱셈기들은 지연시간 내에 수용되므로 두 곱셈기의 사이클 당 지연시간은 프로세서 전체의 지연시간인 20ns가 된다. 이러한 측면에서 다시 성능평가를 하면 표 5와 같다.

설계된 프로세서에 내장할 곱셈기는 최종적으로 23%의 성능향상을 나타낸 R4-32×16 구조가 선택되었다. R4-32×8 구조와 대비하여 15% 정도의 면적 증가로 곱

셈연산의 최대 수행 사이클 수를 7에서 5로 줄여 연산 수행 속도는 40%가 향상되었다. 소비전력은 동일한 구조일 때 면적과 클럭 속도에 비례한다. 곱셈기의 구조가 복잡해질수록 면적과 사이클 당 연산량이 증가하므로 동일한 속도일 때 전력소비는 면적과 구조 복잡도가 큰 R4-32×16 구조가 높을 것이다. 그러나 적용된 알고리즘은 동일하므로 최종 결과의 출력에 소요되는 사이클 수를 고려하면 소비전력의 차이는 미비할 것으로 예상된다. 또한 기존에 내장된 곱셈기의 구조인 R4-32×8과 구조적 유사성으로 무리 없이 프로세서에 내장할 수 있었다. 반면에 R8 구조의 경우 프로세서의 내부구조를 일부 변경하여 선처리방식으로 3배수 처리부의 시간지연을 해결하면 우수한 성능을 가진 내장형 곱셈기가 될 것이다.

V. 설계 검증

곱셈기가 내장된 프로세서를 보다 효율적으로 검증하기위해 그림 13과 같은 명령어 무작위 추출방식(random instruction generation)으로 테스트 벡터를 만들었다[10,11]. 명령어의 기능을 정하는 opcode는 효과적인 설계오류 검출을 위해 곱셈 명령어를 중심으로 특성별로 분류하여 테이블화하였으며 opcode의 특성에 따라 필요로 하는 레지스터 번호, 상수 값, 주소 값 등은 일정한 한계 값 내에서 무작위로 생성시켰다. 이렇게 생성된 명령어 코드는 기능별 분류기준에 따라 다양한 방식으로 조합되어 곱셈 명령어사이에서 발생할 수 있는 오동작을 검출할 수 있게 하였다. 명령어들이 프로세서 내에서 실행될 때마다 연산결과 값과 상태 값들이 외부 메모리에 전달되도록 그림 13의 가운데와 같이 2개의 루틴을 결합시켜 테스트 벡터가 생성되게 하였다. 모든 명령어들의 시뮬레이션 결과는 메모리에 저장되어 상용 시뮬레이터의 결과와 비교 검증되었다[12].

응용 프로그램을 이용한 수행검증을 위해 다양한 오디오 신호처리 알고리즘을 구현하였다. 구현된 알고리즘은 ADPCM (G.726)과 SOLA(synchronized overlap and add using edge detection) 및 MP3 디코딩으로 오디오 응용분야에서 널리 이용되는 것이다. SOLA 알고리즘은 음정의 변화 없이 오디오 속도변환기능을 수행하는 것으로 어학학습에 유용하다. ADPCM과 SOLA 알고리즘은 컴파일러를 거치지 않고 직접 어셈블리어로 구현했기 때문에 컴파일러에서 사용되지 않는 명령어와 의도된 명령어간의 조합을 적용할 수 있었다. MP3 디코딩 알고리즘의 경우 일차적으로 컴파일러를 통해 변환하였고 부분적으로 인위적 수정을 하였다[8]. 이러한 응용 프로그램의 시뮬레이션 수행에는 긴 시간과 노력을 요하므로 오디오 프레임 단위로 C 프로그램의 결과값과 로직 시뮬레이션의 결과 값을 비교하여 명령어 동작이 정확히 일치함을 확인하였다.

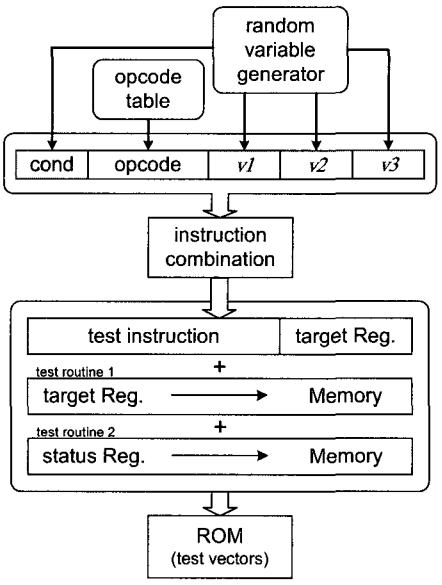

그림 13. 테스트 벡터 생성 과정
Fig. 13. Flow chart of test vectors generation.

VI. 결 론

본 논문에서는 ARM7과 호환되는 32비트 RISC 프로세서에 내장할 수 있는 R4-32×8, R4-32×16, R8-32×16과 같은 3가지 구조의 곱셈기를 설계하고 비교분석하였다. 면적과 수행속도의 2가지 측면으로 프로세서에 최적화된 구조를 연구하였다. 정확한 소비전력의 측정은 시행하지 못했으나 면적 및 사이클 속도를 기준으로 간접적인 비교는 가능하였다. 기능블럭의 최적화가 아닌 구조적 변경으로 연산속도 측면에서 40% 개선시켰으며 증가된 면적은 15%대에 머물렀다. 이러한 성능으로 최종 설계된 곱셈기는 R4-32×16방식의 구조이다. 설계된 곱셈기는 50Mhz로 동작하는 ARM7 호환 프로세서에 내장되어 명령어 테스트 및 다양한 오디오 응용 알고리즘 시뮬레이션을 통해 정확한 연산동작이 검증되었다. 향후 계획으로 5단 이상의 파이프라인을 가진 프로세서에 내장되는 곱셈기의 연구를 수행할 것이다.

참 고 문 헌

[1] Steve Furber, *ARM System Architecture,* Addison-Wesley, 1996.

[2] Dave Jagger, *ARM Architectural Reference Manual,* Prentice Hall, London, 1996.

[3] M. K. Ibrahim, "Radix-2n multiplier structure: a structured design methodology", IEE Proc. Computers and Digital Techniques Vol. 140, pp. 185-190, July 1993.

[4] Brian S. Cherkauer and Eby G. Friedman, "A Hybrid Radix-4/Radix-8 Low Power Signed Multiplier Architecture", IEEE Trans. on Circuits and Systems Vol. 44 No. 8, pp. 656-659, Aug. 1997.

[5] C.S. Wallace, "High speed arithmetic in binary computers," *IEEE Trans. Electron. Comput.,* Feb. 1964.

[6] G. Y. Jeong and J. S. Park, "Implementation of 32-bit RISC Processor", IDEC Conference, pp. 200-201, KAIST korea, Aug. 2002.

[7] G. Y. Jeong and J. S. Park, "Design and Verification of 32-bit RISC Processor", *AP-SOC 2002 (Asia Pacific-System on a chip 2002),* pp. 228-231, COEX, Korea, Nov. 2002.

[8] G. Y. Jeong and J. S. Park, "Implementation of 32-bit RISC Processor and Efficient Verification", *8th Korea-Russia International Symposium on Science and Technology,* Tomsk Univ., Russia, June 2004.

[9] Michael J. Flynn and Stuart Oberman, *Advanced Computer Arithmetic Design,* Wiely Interscience, 2001.

[10] Ta-Chung Chang, "A Biased Random Instruction Gerneration Environment for Architectural Verification of Pipelined Processor," *in Journal of Electronic Testing : Theory and Applications* 16, pp. 13-27, 2000.

[11] M. Bose, E. M. Rudnick, M. Abadir, "Automatic bias generation using pipeline instruction state coverage for biased random instruction genera-tion", *On-Line Testing Workshop 2001 Procee-dings Seventh International,* pp. 65-71, July 2001.

[12] C. Pixley, and et al., "Commercial Design Verification: Methodology and Tools," *Proc. IEEE Int. Test Conf.,* pp. 839-848, 1996.

저 자 소 개

정 근 영(학생회원)

1997년 부산대학교 전자공학과 학사 졸업.

1999년 부산대학교 전자공학과 석사 졸업.

2004년 부산대학교 전자공학과 박사과정.

<주관심분야: 마이크로 프로세서 설계, SoC 설계, 오디오 신호처리>

김 석 찬(정회원)

1993년 부산대학교 전자공학과 학사 졸업.

1995년 한국과학기술원 전기 및 전자공학과 석사 졸업.

2000년 한국과학기술원 전기 및 전자공학과 박사 졸업.

2002년~현재 부산대학교 전자공학과 조교수

<주관심분야: 이동통신, 통신신호처리, 무선전송, WLAN, WPAN, DSP 설계>

박 주 성(평생회원)

1976년 부산대학교 전자공학과 학사 졸업.

1978년 한국과학기술원 전기 및 전자공학과 석사 졸업.

1989년 Univ. of Florida 전자공학과 박사 졸업.

1978년~1985년 ETRI 실장.

1991년~현재 부산대학교 전자공학과 교수.

1998년~현재 부산대 IDEC 센터장.

<주관심분야: DSP 설계, ASIC 설계, SoC 설계, 반도체 소자 모델링, 음성/사운드 신호처리 및 구현>