論文2001-38SD-6-6

멀티미디어 확장 프로세서의 명령어 집합 구조에 관한 연구

(A Study on the Instruction Set Architecture of Multimedia Extension Processor)

吳 明 勳 *, 李 東 翊 *, 朴 性 模 **

(Myeong-Hoon Oh, Dong-Ik Lee, and Seong-Mo Park)

요 약

최근의 멀티미디어의 발달에 따라 범용 프로세서에서 멀티미디어 데이터를 효과적으로 처리하려는 연구가 계속되고 있다. 본 논문에서는 범용 프로세서 안에서 멀티미디어 데이터를 효율적으로 처리할 수 있는 명령어들과 그 프로세서의 구조를 제안하고 이를 HDL(Hardware Description Language)로 행위 레벨에서 기술하고 시뮬레이션 하였다. 제안된 멀티미디어 명령어는 특성에 따라 7개의 그룹에 총 48개의 명령어로 구성되며 64비트 데이터 안에서 각각 8비트의 8바이트, 16비트의 4하프워드, 32비트의 2워드의 subword 데이터들을 병렬 처리한다. 모델링된 프로세서는 오픈아키텍처(Open Architecture)인 SPARC V.9의 정수연산장치에 기반을 두었으며 하바드 구조를 지닌 5단 파이프라인 RISC 형태이다.

Abstract

As multimedia technology has rapidly grown recently, many researches to process multimedia data efficiently using general-purpose processors have been studied. In this paper, we proposed multimedia instructions which can process multimedia data effectively, and suggested a processor architecture for those instructions. The processor was described with Verilog-HDL in the behavioral level and simulated with CADENCE™ tool. Proposed multimedia instructions are total 48 instructions which can be classified into 7 groups. Multimedia data have 64-bit format and are processed as parallel subwords of 8-bit 8 bytes, 16-bit 4 half words or 32-bit 2 words. Modeled processor is developed based on the Integer Unit of SPARC V.9. It has five-stage pipeline RISC architecture with Harvard principle.

* 正會員, 光州科學技術院 情報通信工學科

(Dept. of Info. & Comm., Kwang-Ju Institute of Science and Technology)

** 正會員, 全南大學校 컴퓨터工學科

(Dept. of Computer Engineering, Chonnam National University)

※ 본 논문은 한국과학재단 지정 전남대학교 고품질 전기 전자 부품 및 시스템 연구센터와 IDEC 사업의 지원에 의해 연구되었음

接受日字:1999年2月5日, 수정완료일:2001年5月28日

Ⅰ. 서 론

과거에 디지털 정보들은 숫자와 문자에만 국한되었고 멀티미디어 정보들은 아날로그방식으로 처리되어 왔다. 하지만 최근의 멀티미디어 정보들은 품질, 속도, 신뢰도, 가격 등 여러 면에서 큰 이득을 볼 수 있는 디지털 방식으로 처리되고 있고 이미지, 비디오, 오디오, 삼차원 그래픽, 애니매이션 등을 포함하는 통합적인 멀티미디어 정보로 발전되고 있어 오늘날 대부분의 정보 활동은 이러한 디지털 멀티미디어 정보의 처리를 포함하게 되었다.

초기의 멀티미디어 데이터는 주로 오디오 데이터에 대하여 전용 보드 수준에서 처리되었으며 전용 프로세싱 칩과 전용 메모리를 사용하여 처리하였다. 이후에 오디오 데이터 처리 알고리즘을 구현하기 위해 프로그램 가능한 DSP 칩을 사용하는 형태로 발전하였다. 즉, 각각의 사운드 채널에 대해 DSP를 사용하고 또 거기에 하드웨어의 구조를 상세히 알아서 제어해 주는 하위레벨 프로그램이 필요한 형태였다. 그러나 멀티미디어 기술의 발달에 따라 이러한 오디오 데이터 처리용 DSP 칩들은 3차원 그래픽, 비디오, 동영상처리에서 요구되는 큰 대역폭과 빠른 처리성능을 감당하지 못하게 되었다. 이러한 단점을 극복하는 대안으로 시스템의 특성에 따라 DSP Core, 제어용 CPU, 아날로그 접속을 위한 AFE(Analog Front End), 기타 일반부품들을 집적화한 시스템 전용 ASSP(Application Specific Standard Product) 비디오 칩셋이 개발되었고 보편화되었다. 이는 특정 애플리케이션을 위하여 개발자가 특성에 맞는 전용 칩들을 사용한 형태이다.

DSP 칩이나 ASSP 비디오 칩셋은 멀티미디어 데이터 처리 시에 각각의 고유한 기능들을 하드웨어에서 처리하므로 속도 면에서 매우 우수하다. 하지만 각각 데이터 타입과 처리 방식이 다른 멀티미디어 데이터들을 하드웨어에서 처리하는데는 한계가 있다. 즉, 다양한 멀티미디어 데이터마다 고유한 기능들을 제공하려 한다면 기능의 수에 따라 하드웨어의 크기가 커지게 된다. 멀티미디어 데이터를 처리하는 하드웨어를 공유시키는 경우에는 프로세서 설계의 복잡성이 증가되고 설계 시간도 오래 걸리며 검증 또한 어렵게 된다.

전적으로 하드웨어에 의존하여 멀티미디어 데이터를 처리하지 않고 멀티미디어 데이터를 위한 명령어를 추가하여 소프트웨어로 처리하면 프로세서의 크기를 줄일 수 있고 새로운 형태의 멀티미디어를 처리하는데 flexibility를 증가시킬 수 있다[1]. 범용에 초점을 맞추면서 멀티미디어 데이터의 효율적인 처리를 위하여 미디어 프로세서와 멀티미디어 확장 프로세서 등 2가지 형태의 프로세서가 개발되었다. 미디어 프로세서는 여러 가지 멀티미디어 데이터를 처리하면서 범용 프로세서처럼 작동하여 메모리 관리, 정수 및 부동소수점 계산을 할 수 있는 프로세서 형태이고, 멀티미디어 확장 프로세서는 기존의 고성능 범용 프로세서에 멀티미디어 데이터를 전용으로 처리할 수 있는 명령어 집합을 첨가하고 이를 수행할 수 있게 약간의 구조를 변경한 형태이다.

미디어 프로세서는 처리하는 데이터가 지극히 멀티미디어 데이터에 국한된 프로세서이므로 사용범위가 제한되어 있다. 따라서 프로세서 개발 회사들은 기존의 범용 프로세서에서 멀티미디어 정보를 효과적으로 처리할 수 있는 멀티미디어 확장 프로세서의 개발에 그 관심을 돌리기 시작했다. 대표적으로 Sun사의 VIS(Visual Instruction Set)를 포함한 Ultra-Sparc, Intel사의 MMX(Multi-Media eXtension)를 포함한 Pentium 프로세서, Hewlett-Packard사의 MAX-2 (Multimedia Accelation eXtension)를 포함한 PA-RISC2.0이 좋은 예이다.

멀티미디어 데이터를 기존의 마이크로프로세서에 확장하여 처리하는 초기의 마이크로프로세서들은 압축된 저 정밀도(lower precision) 데이터를 고 정밀도(higher precision) 워드에서 병렬적으로 처리한다는 특징을 가지고 있었다. 이 개념은 최근의 멀티미디어 처리를 지원하는 마이크로프로세서의 가장 기본적인 명령어의 형태로 나타나며 정형화되었다[2].

Hewlett-Packard사에서 개발한 프로세서인 PA-7200 LC에 구현된 멀티미디어 명령어 셋인 MAX-1은 적은 숫자의 명령어들로 병렬 subword 연산을 수행할 수 있었으며 그래픽 데이터뿐만 아니라 모든 멀티미디어 데이터 처리를 목표로 하였다. 하지만 진정한 멀티미디어 명령어는 아니었으며 MAX-1의 명령어를 직접 불러다 쓰는 매크로 형식을 사용하여 고수준 언어인 C로 수행되는 MPEG-1에 사용되었다. MAX-2는 64비트 마이크로프로세서인 PA-RISC 2.0과 함께 소개되었다. MAX-2는 MAX-1에 덧붙여 subword 데이터의 정렬(alignment)과 재배열(rearrangement)을 수행하는 명령어들이 추가되었으며 정수 레지스터를 오퍼랜드로 사용하였다[3].

SUN사의 UltraSparc 마이크로프로세서에 사용되는 VIS에는 멀티미디어 명령어의 수가 많이 늘어나 병렬 연산뿐만 아니라 visual 데이터를 조작하여 메모리 latency를 줄이기 위해 특별히 고안된 명령어를 제공한다. 또한 VIS는 정수 파이프라인과 부동소수점 파이프라인이 구별되어 있는 구조를 이용하기 위해 부동소수점 레지스터 파일을 사용함으로써 정수 레지스터가 어드레싱과 분기계산에만 전념할 수 있게 하고 다중싸이클이 필요한 VIS명령어를 부동소수점 파이프라인에 mapping시켜 명령어처리를 효율적으로 할 수 있게 하

었다. 하지만 비교 VIS 명령어의 결과 값은 정수 레지스터에 저장한다[4,5].

Intel 사에서 개발한 그래픽 랜더링을 지원하는 최초의 범용 프로세서인 i860은 인접 데이터들을 병렬적으로 연산하였으며 이 i860 프로세서를 기초로 이미지 프로세싱, 텍스처 맵핑, 오디오, 비디오 신호 처리와 같은 고성능을 요구하는 멀티미디어 알고리즘 처리를 지원하고 기존의 Intel x86계열의 프로세서의 구조와 호환성을 유지하기 위한 방안으로 MMX가 고안되었다. VIS, MAX-2와는 구별되는 병렬 16비트 곱셈-덧셈 명령어와 같은 특별한 명령어를 지원한다. MMX는 실제로는 부동 소수점 레지스터에 있는 64비트의 압축 데이터를 저장하는 MM0-MM7의 MMX 레지스터를 부동 소수점 명령어처럼 스택 모델을 거치지 않고 억세스 하여 사용하며 정수 레지스터를 사용하기도 한다. 이는 레지스터 allocation을 쉽게 하고 저장해야할 데이터를 임시적으로 메모리에 저장해야하는 경우를 줄일 수가 있다[6,7].

앞에서 설명한 MAX-2, VIS, MMX는 모두 64비트의 데이터를 처리하며 하드웨어구조에 따라 각기 다른 특징을 가진다. 즉, 오퍼랜드 개수를 살펴보면 VIS는 3개, MMX는 2개, MAX-2는 3개이며 정수 연산 형태는 VIS가 4개의 16비트, 2개의 32비트의 데이터를 처리하고 MMX는 8개의 8비트, 4개의 16비트, 2개의 32비트 데이터를 처리한다. 반면 MAX-2는 4개의 16비트 데이터 형태 한가지만을 지원한다.

각각의 명령어 집합구조에서 지원되는 멀티미디어 데이터 형태와 멀티미디어 명령어 숫자가 다르다는 점에서 멀티미디어 데이터를 처리하는 명령어의 종류와 subword의 크기는 프로세서의 구조뿐만 아니라 target 시장의 필요에 따라 달라질 수 있다는 것을 알 수 있다.

본 논문에서는 이 범용 프로세서들을 분석한 결과를 바탕으로 범용 프로세서 안에서 멀티미디어 데이터를 효과적으로 처리하기 위한 명령어와 이를 처리할 수 있는 멀티미디어 확장 프로세서의 구조를 제안하고 모델링하여 이를 시뮬레이션 하였다. 논문의 구성은 제 2장에서 추가한 멀티미디어 명령어를 구체적으로 설명하고 제 3 장에서는 모델링된 프로세서의 구조 및 설계 방법에 관하여 기술하였다. 제 4 장에서는 멀티미디어 명령어 적용 예를 기술하였고 제 5 장에서 결론을 맺었다.

Ⅱ. 멀티미디어 명령어

본 논문에서 제안하는 멀티미디어 데이터 형태와 멀티미디어 명령어는 오픈 아키텍처인 64비트 SPARC 구조의 정수연산장치(IU : Integer Unit)[8,9]에 확장 가능하다. 추가한 멀티미디어 명령어는 멀티미디어 알고리즘의 몇 가지 처리 특성과 SPARC IU의 아키텍처를 고려하여 선택되었고 64비트 데이터 안에서 subword를 병렬로 연산함으로써 SIMD(Single Instruction Multiple Data) 연산을 수행할 수 있다.

VIS와 MMX가 부동 소수점 데이터 패스를 사용하여 멀티미디어 명령어를 처리하는 반면에 본 논문에서 제안하는 멀티미디어 명령어는 정수 데이터 패스 및 정수 레지스터를 사용한다. 그 이유는 첫째로 logical 명령어 같은 몇 가지 정수연산을 위한 블록을 정수 연산장치에 있음에도 불구하고 똑같은 연산장치를 부동 소수점 연산장치에 부가해야되는 부담을 덜기 위해서이고, 두 번째로는 멀티미디어 명령어를 처리하기 위해 정수연산장치를 변경하는 것이 부동 소수점 연산장치를 변경하는 것보다 용이하기 때문이다. 또한 본 논문의 멀티미디어 명령어 세트는 94개의 VIS 명령어나 57개의 MMX 명령어보다 가능한 적은 수의 명령어로 기존의 멀티미디어 명령어로 가능했던 application을 포함한 다양한 기능을 수행할 수 있는 확장성과 명령어를 application에 적용 시에도 보다 적은 수의 명령어로 수행 가능할 수 있는 효율성에 초점을 두었다.

1. 멀티미디어 데이터의 특징

(1) 압축 데이터 형태

대부분의 멀티미디어 알고리즘의 큰 특징은 사용되는 데이터 오퍼랜드 크기가 8비트 또는 16비트로 작다는 것이다. 또한 멀티미디어 데이터처리시 대부분 많은 수의 인접 데이터로 반복적인 같은 연산을 수행한다. 이 두 가지 특징은 SIMD의 기본 idea를 제공한다. 본 논문에서 제안하는 멀티미디어 데이터의 크기는 총 64비트이며 이 안에서 각각의 subword로써 8비트, 16비트, 32비트를 지원한다.

(2) 고정 소수점 연산

이미지 처리에서 filtering에 사용되는 계수값과 같이 멀티미디어 응용 중에는 소수점 데이터를 연산하는 경우가 있다. 대개의 경우 부동 소수점 유닛을 사용하여

이를 처리하지만, 멀티미디어 데이터를 병렬적으로 처리하는데 있어서 연산 성능, 부동 소수점 유닛과의 인터페이스 및 설계 시에 복잡성, 부동 소수점 데이터의 사용빈도 등을 고려 할 때 고정 소수점 연산 방식이 적합하다. 고정 소수점 연산에서 데이터들은 모두 정수형으로 계산되며 부동 소수점 데이터로의 변환은 소프트웨어적으로 하고 데이터의 scaling에 필요한 병렬 쉬프트 명령어가 지원된다.

(3) Saturating 연산

대부분의 멀티미디어 응용에서 연산결과 overflow 발생 시 결과값으로써 wrap-around 방식으로 저장하기보다는 최대값을 저장하는 것이 바람직하다. 예를 들어 이미지 처리에 사용되는 8비트 흑백 표현방식에서 덧셈연산 결과 최대값인 255(흰색)를 넘는 overflow가 발생하였을 때 결과 값을 wrap-around방식으로 저장하면 검은색일 확률이 높다. underflow의 경우도 마찬가지이므로 본 논문에서 제안하는 병렬 subword 덧셈과 뺄셈의 명령어는 이 saturation 방식의 연산을 지원한다.

(4) 마스크 생성에 의한 조건 실행

한 개의 명령어로 다수의 연산을 수행하는 병렬 subword끼리의 오퍼랜드의 연산에서 조건에 따라 다른 연산을 수행할 경우에 문제점이 생긴다. 보통의 경우 조건 연산은 상태 레지스터의 값을 가지고 분기할 것인지 아닌 지를 결정하여 연산을 수행한다. 병렬 subword 데이터 안에서 조건 연산을 수행할 때 앞의 경우처럼 처리한다면 subword데이터 수만큼의 상태 레지스터 정보가 필요할 뿐만 아니라 더욱이 각각의 subword마다 분기 여부가 다를 것이므로 명령어 하나로 처리하기는 불가능하며 SIMD의 구조에도 맞지 않는다. 본 논문에서는 멀티미디어 데이터 처리 시에 조건 연산을 수행하는 경우 발생하는 문제점을 해결하기 위해 마스크를 생성하는 비교 명령어를 두었다. 즉, 조건 비교 시에 각각의 subword마다 참과 거짓을 나타내는 마스크를 생성하고 참에 대한 연산과 거짓에 대한 연산에 이 마스크와 논리 연산을 수행하면 각 subword에 대한 조건 연산을 병렬적으로 처리할 수 있다.

(5) 압축 데이터 형태의 변환

멀티미디어 알고리즘이 intermediate 계산에서 고정밀도(higher precision)의 데이터 형태를 필요로 하는 경우가 있다. 예를 들어 이미지 필터링 연산에서 8비트의 픽셀 값과 계수값을 곱하는 경우 거의 대부분의 결과가 8비트로는 부족하여 overflow를 발생한다. 이 overflow를 방지하기 위해서 결과 저장 시에 더 고정밀도의 데이터 형태가 필요하다. 이를 위해 각 subword 데이터들의 정밀도를 변환하는 확장 명령어를 지원하며 나중에 다시 저정밀도 데이터 형태로 변환하기 위한 압축 명령어를 지원한다.

2. 멀티미디어 데이터 형태

그림 1은 멀티미디어 명령어의 형태를 나타내고 있다. 제안하는 멀티미디어 명령어는 SPARC V.9의 명령어 형태에 기초를 두고있으며 정수연산장치에서 멀티미디어 데이터가 처리될 수 있도록 하였다.

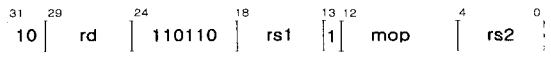

그림 1. 멀티미디어 명령어 형태
Fig. 1. Multimedia instruction format.

명령어 형태에서 [31:30]과 [24:19]까지의 필드 값은 "10110110"으로 하였는데 이는 SPARC V.9 명령어 집합에서 Implement dependant로 지정되어 있으므로 이 필드 값을 사용하였다. 오퍼랜드로 레지스터 소스만을 사용하며 immediate 소스는 사용하지 않는다. rs1은 소스 1번째 레지스터 주소 값을 나타내고 rs2는 소스 2번째 레지스터 주소 값이다. 각 멀티미디어 명령어는 mop 필드 값에 의해 구분되어진다.

그림 2는 멀티미디어 명령어가 처리하는 데이터의 형태를 나타내고 있다. 총 데이터의 길이는 64비트이며 8개의 8비트 바이트, 4개의 16비트 하프워드, 2개의 32비트 워드데이터를 처리한다. 처리하는 데이터의 종류에 따라서 각 명령어 그룹의 명령어들이 구분되며 이는 mop 필드에 나타난다. mop의 하위 2비트를 바이트는 "00"으로 하프워드는 "01"로 워드는 "11"로 할당하였다.

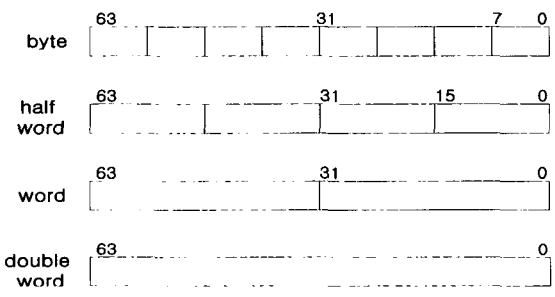

그림 2. 멀티미디어 데이터 형태
Fig. 2. Multimedia data format.

3. 제안하는 멀티미디어 명령어

SPARC V.9의 명령어 집합에 추가한 멀티미디어 명령어는 총 48개의 명령어로 7개의 그룹으로 구분된다. 7개의 그룹은 각 subword끼리 덧셈과 뺄셈을 수행하는 add/sub 명령어 그룹, 비교하여 마스크를 생성하는 compare 명령어 그룹, 쉬프트 연산을 수행하는 shift 명령어 그룹, subword 데이터를 압축하고 확장하는 conversion 명령어 그룹, 데이터의 배열을 바꾸는 rearrange 명령어 그룹, 곱셈을 수행하는 multiply 명령어 그룹, 마지막으로 8개의 subword의 절대값을 계산하는 pixel distance 명령어 그룹 등이다. 표 1에는 멀티미디어 명령어의 mnemonic과 그룹에 따른 mop[7:4]의 비트열이 할당되어 있다. 다음에서 각각의 명령어 그룹을 상세히 설명한다.

(1) add/sub 명령어 그룹

add/sub 명령어 그룹은 총 12개의 명령어로 이루어져있다. madd와 msub 명령어는 각각의 바이트, 하프워드, 워드의 데이터를 덧셈과 뺄셈 연산을 한다. overflow, underflow를 무시하는 wrap-around방식의 연산과 overflow, underflow 발생 시에 특정 값으로 대치시키는 saturation방식의 연산을 구분하여 수행한다. 표 1에서 mnemonic 내의 'b'는 바이트, 'h'는 하프워드, 'w'는 워드를 나타내고 's'가 없는 것은 wrap-around방식의 연산, 's'가 있는 것은 saturation 방식의 연산을 수행하는 명령어이다. 그림 3은 maddb 명령어의 수행방식을 보여주고 있다.

표 1. 멀티미디어 명령어의 mnemonic
Table 1. Mnemonic of multimedia instruction.

명령어 그룹	mop[7:4]	mnemonic
add/sub 명령어	0101	madd(b/h/w/sb/sh/sw), msub(b/h/w/sb/sh/sw)
compare 명령어	0010	mcmgt(b/h/w),mcmple(b/h/w), mcmpne(b/h/w),mcmeq(b/h/w)
shift 명령어	0111	msll(h/w), msrl(h/w), msra(h/w)
conversion 명령어	0011	mpack(hb/wb/wh), mexpand, mmerge
rearrange 명령어	1000	mmix(lb/rb/lh/rh/lw/rw)
multiply 명령어	1001	mmul(bh/bhu/bhl/dbhu/dbhl), mmuladdh
pixel distance 명령어	0011	mpdist

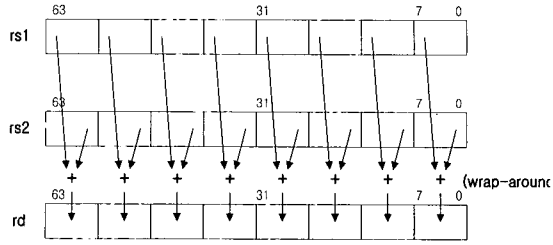

그림 3. maddb 명령어 수행도
Fig. 3. Diagram of maddb instruction.

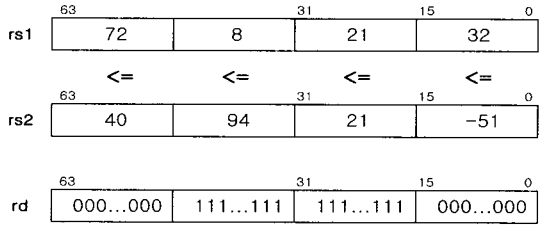

그림 4. mcmpleh 명령어의 수행도
Fig. 4. Diagram of mcmpleh instruction.

(2) compare 명령어 그룹

compare 명령어 그룹은 "greater than", "less than", "not equal", "equal"의 4개의 비교 연산을 수행한다. 소스 1 레지스터 값과 소스 2레지스터 값을 비교하여 결과 값이 참이 되면 목적지 레지스터의 각 subword의 비트들에 '1'을 저장하고 결과 값이 거짓인 경우는 '0'으로 세팅한다. 이렇게 저장된 비트 마스크는 주로 논리연산 명령어의 소스 레지스터 값으로 사용된다. 그림 4는 mcmpleh의 수행을 보여주고 있다.

(3) shift 명령어 그룹

쉬프트 명령어는 다른 일반 쉬프트 명령어와 같이 오른쪽, 왼쪽의 쉬프트와 logical, arithmetic 방식의 명령어로 구분된다. 처리 데이터 크기는 하프워드, 워드로 구분된다. 표 1의 shift 명령어 그룹에서 mnemonic의 처음 'l'은 "left", 'r'은 "right", 두 번째 'l'은 "logical", 'a'는 "arithmetic"을 의미한다. 쉬프트 명령어는 데이터의 변환과 정렬에 사용될 수 있다. 수행방식은 소스 1 레지스터의 각 subword를 소스 2 레지스터의 하위 4비트, 5비트, 6비트의 값에 따라서 쉬프트하게 된다.

(4) conversion 명령어 그룹

멀티미디어 데이터 처리 시에 데이터의 크기가 변하는 intermediate form을 요구하는 경우가 종종 발생한다. 이 때에 사용되는 명령어 그룹이다. 표 1에서 mpackhb는 16비트의 하프워드의 subword를 8비트의 바이트 단위로 압축하는 명령어이며 mpackwb는 32비

트의 워드를 8비트의 바이트로, mpackwh는 32비트의 워드를 16비트의 하프워드로 변환한다. 반대로 mexpand는 8비트의 바이트 단위의 subword를 16비트의 하프워드로 변환하며 mmerge는 64비트의 더블워드를 만들어 낸다. 각 명령어들을 좀더 자세히 설명하면 mpackhb는 소스 1 레지스터의 각 subword를 소스 2 레지스터의 하위 4비트값에 따라 '0'을 왼쪽 쉬프트시킨 후에 [14:7]까지의 값을 잘라내어 목적지 레지스터에 저장한다. 그림 5는 mpackhb의 수행을 보여준다. 그림 5에서 왼쪽의 예는 소스 2 레지스터 하위 4비트가 "1010"이므로 소스 1 레지스터의 각 subword를 왼쪽으로 10만큼 쉬프트한 후 [14:7]의 값을 취하게 되고 오른쪽의 예는 소스 2 레지스터 하위 4비트가 "0100"이므로 '0'을 4개 쉬프트한 후 역시 [14:7]의 값을 취하게 된다.

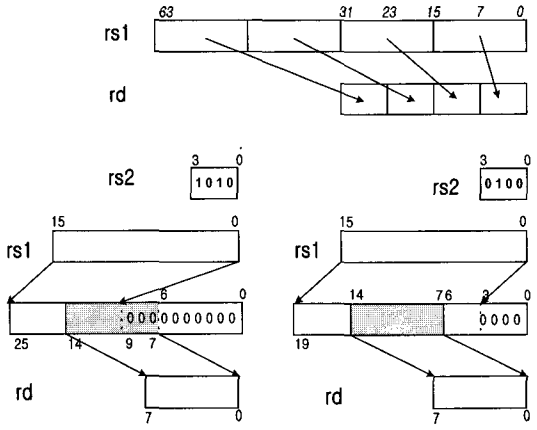

그림 5. mpackhb 명령어 수행도
Fig. 5. Diagram of mpackhb instruction.

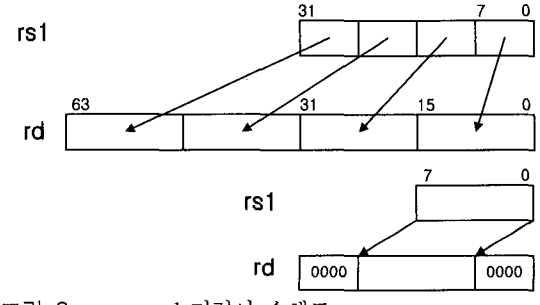

그림 6. mexpand 명령어 수행도
Fig. 6. Diagram of mexpand instruction.

mpackwb 명령어도 소스 2 레지스터의 하위 4비트값에 따라 32비트의 워드값을 쉬프트한 후 이번에는 [30:23]의 데이터 값을 취하게 된다. mpackwh는 소스 1 레지스터의 32비트 2개의 워드값을 16비트의 하프워드로 변환하여 목적지 레지스터에 저장한다. 마찬가지로 소스 2 레지스터의 하위 4비트 값에 따라 소스 1 레지스터의 각 subword를 '0'을 삽입하여 왼쪽으로 쉬프트 한 후에 [31:16]의 값을 취하게 된다. mexpand 명령어는 8비트의 subword를 16비트로 확장시킨다. 소스 1 레지스터의 하위 32비트에 저장된 4개의 8비트 subword를 확장하여 목적지 레지스터에 저장한다. 확장 시에는 8비트 값이 16비트의 subword에 그대로 [11:4]에 저장되고 상위 4비트와 하위 4비트는 '0'으로 대치된다. 그림 6은 mexpand의 수행과정을 보여준다. mmerge명령어는 2개의 32비트 소스 레지스터의 값으로부터 64비트의 값을 만들어낸다. 확장 시에는 소스 1 레지스터의 하위 32비트와 소스 2 레지스터의 하위 32비트에서 4개의 8비트 subword를 interleaved방식으로 목적지 레지스터에 저장한다. 이 명령어는 8비트 subword의 배열을 바꾸는데도 사용할 수 있다.

(5) rearrange 명령어 그룹

rearrange 명령어 그룹은 표 1에서와 같이 모두 6개로 구성되어 있다. 모두 64비트의 소스레지스터 데이터 안에서 8개의 8비트 subword, 4개의 16비트 subword, 2개의 32비트 subword의 자리를 재배치한다. 표 1에서 rearrange 명령어 그룹의 mnemonic 중 'l'이 나타내는 의미는 각 소스 레지스터의 왼쪽에 있는 값을 배열시킴을 가리키며 'r'은 오른쪽에 있는 값들을 배열시킴을 나타낸다. 'b', 'h', 'w'는 각각 바이트와 하프워드, 워드를 나타내며 각각 8비트, 16비트, 32비트 연산을 수행한다. mmixlb 명령어는 소스 1 레지스터와 소스 2 레지스터의 1번째, 3번째, 5번째, 7번째 subword 바이트의 값을 재배열하여 목적지 레지스터에 저장한다. 그림 7은 이를 보여주고 있다. mmixlh 명령어는 소스 1 레지스터와 소스 2 레지스터의 상위 1번째, 3번째 subword 하프워드 데이터들을 재배열시킨다. mmixrb 명령어는 소스 1 레지스터와 소스 2 레지스터의 2번째, 4번째, 6번째, 8번째 바이트의 값을 재배열하여 목적지 레지스터에 저장한다. mmixrh 명령어는 소스 1 레지스터와 소스 2 레지스터의 상위 2번째, 4번째 subword 하프워드 데이터들을 재배열시킨다. 마찬가지로 mmixlw와 mmixrw는 32bit 워드 데이터들을 재배열한다. 그림 8은 이 mmixrh 명령어의 수행을 보여준다.

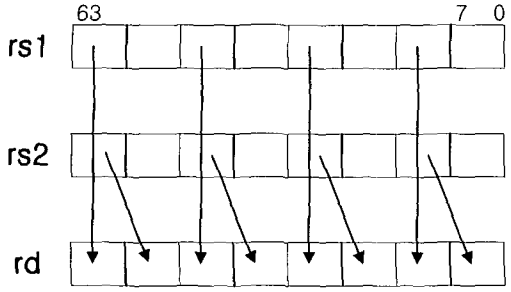

그림 7. mmixlb 명령어 수행도
Fig. 7. Diagram of mmixlb instruction.

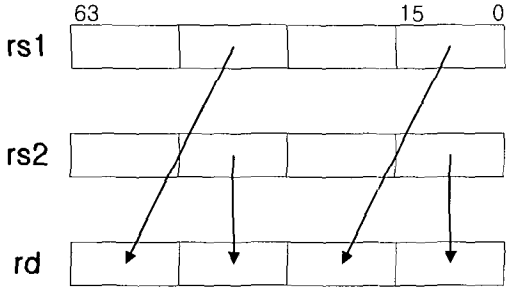

그림 8. mmixrh 명령어 수행도
Fig. 8. Diagram of mmixrh instruction.

(6) multiply 명령어 그룹

multiply 명령어 그룹은 모두 6개의 명령어로 구성되어 있다. mmulbh 명령어는 4개의 8비트 unsigned 데이터와 16비트 signed 데이터와의 곱셈을 하여 상위 16비트 값을 취하게 된다. 그림 9는 이 명령어의 수행을 보여준다.

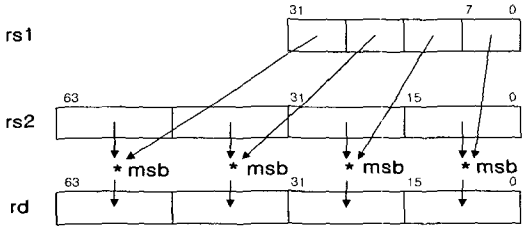

그림 9. mmulbh 명령어 수행도
Fig. 9. Diagram of mmulbh instruction.

mmulbhu 명령어는 두 소스 레지스터의 4개의 16비트 subword끼리 곱셈을 하는데 각 subword에서 소스 1 레지스터의 상위 8비트만 곱셈을 한다. 소스 1 레지스터는 signed 데이터로 처리된다. 마찬가지로 결과는 상위 16비트만을 취한다. mmulbhl 명령어는 앞의 mmulbhu 명령어와 같은 연산을 수행하는데 다른 점은 소스 1 레지스터의 각 subword내의 하위 unsigned 8비트 값을 소스 2 레지스터의 subword와 곱하게 된다. 저장 시에는 sign-extended 데이터를 취하게 된다. mulbhu 명령어와 mmulbhl 명령어는 16비트와 16비트의 곱셈을 하여 16비트 데이터 값을 생성하는데 사용된다. mmuldbhu 명령어는 소스 1 레지스터의 하위 32비트 내의 2개의 16비트 signed 데이터의 상위 8비트를 소스 2 레지스터의 하위 32비트 내의 2개의 16비트 signed 데이터와 곱해진다. 결과로 생긴 24비트의 데이터는 그대로 목적지 레지스터에 저장된다. 목적지 레지스터에 저장시 각 subword끼리 곱해진 24비트의 결과 값은 목적지 레지스터의 상위에 저장되고 나머지 하위 즉 목적지 레지스터의 [39:32]과 [7:0] 필드는 '0'으로 채워진다. mmuldbhl 명령어는 소스 1 레지스터의 하위 32비트 내의 2개의 16비트 데이터의 하위 8비트를 소스 2 레지스터의 하위 32비트 내의 2개의 16비트 signed 데이터와 곱한다. 이때 소스 1 레지스터의 데이터는 unsigned 값을 처리한다. 결과 값을 목적지 레지스터에 저장할 때는 sign-extended 한 값을 저장한다. mmuldbhu 명령어와 mmuldbhl 명령어를 같이 사용하면 16비트 데이터끼리의 곱셈을 수행하여 32비트 결과 값을 생성할 수 있다. multiply 명령어 그룹의 마지막 명령어로 mmuladdh 명령어는 곱셈 후에 덧셈을 연산한다. 그림 10과 같이 mmuladdh 명령어는 소스 1 레지스터와 소스 2 레지스터의 각 16비트 subword끼리 곱셈을 하여 생긴 32비트의 결과 값을 서로 더하여 연산한다.

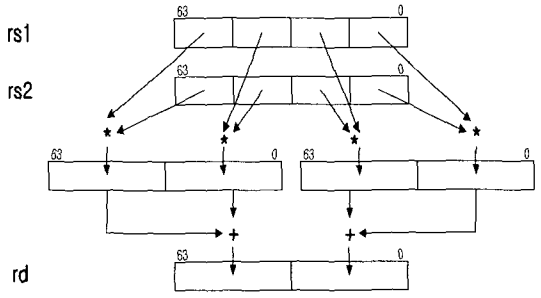

그림 10. mmuladdh 명령어 수행도
Fig. 10. Diagram of mmuladdh instruction.

(7) pixel distance 명령어 그룹

이 명령어 그룹의 명령어인 mpdist는 64비트 데이터 안의 8개의 8비트 subword의 절대값을 계산하여 더하게 된다. 즉 소스 1 레지스터의 각 subword와 소스 2

레지스터의 subword끼리 절대값을 취한 후 목적지 레지스터 값과 계산된 절대값들을 더하게 된다. 이 명령어는 MPEG 2의 motion estimation에 사용된다. 그림 11은 이 명령어의 수행을 보여준다.

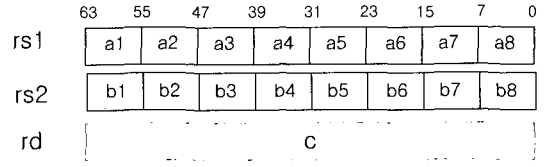

그림 11. mpdist 명령어 수행도
Fig. 11. Diagram of mpdist instruction.

Ⅲ. 프로세서 구조 및 모델링

1. 프로세서 구조

제안한 멀티미디어 명령어를 처리하기 위해 모델링된 프로세서는 SPARC V.9의 IU 명령어와 64비트 데이터를 처리하기 위한 구조를 기본으로 하였으며 F(Fetch), D(Decode), E(Execute), M(Memory), W(Writeback)의 5단 파이프라인 구조에 명령어와 데이터 캐시를 따로 지녀 성능 향상을 꾀할 수 있는 하바드 구조를 채택하였다. 또한 명령어 해석 시에 필요한 제어신호가 단계마다 분산되어 하드웨어 량이 줄어들고 면적과 시간지연의 최적화가 가능한 시간 정적 제어 방식에 의한 파이프라인 구조제어 방법을 채택하였다.

그림 12는 모델링된 프로세서의 데이터 패스이다. 명령어 캐시에서 패치된 명령어는 id[31:0]로 오며 oprd 레지스터에 저장되고 시간 정적 제어 방식에 따라 패치된 명령어가 opex, opm, opw 레지스터로 매 클럭마다 이동하고 각 단에서 제어 신호를 발생한다.

특정시간에서 각 단계의 제어 신호는 겹치지 않으며 이 제어 신호는 데이터 패스를 제어하게 된다. REG GROUP에는 32개의 범용레지스터를 포함한 각종 특별 목적의 레지스터들이 있다. 그리고 data_con block으로부터 오는 제어 신호를 사용하여 읽기 포트에 값을 내보내는 제어 로직회로가 포함되어 있다.

oprd 레지스터에서 REG GROUP로 연결되는 5비트의 rsa와 rsb 신호는 소스 레지스터의 주소를 보내며 주소에 맞는 레지스터 값을 tempA와 tempB로 래치시킨다. 또한 멀티미디어 명령어 중 그 전 목적지 레지스터의 값을 소스 레지스터로 사용하는 명령어를 위해 목적지 레지스터의 주소 값인 rdest 신호 의해 tempC에 레지스터 값이 래치된다. 래치된 레지스터 값들은 opex 레지스터의 제어 신호에 의해 ALU와 Shifter로 구성된 arithmetic block에서 계산된다. 멀티미디어 명령어인 경우는 multi-arith block에서 처리되며 연산 결과 값은 파이프라인 되어 마지막의 w_alu 레지스터 값이 wdata 레지스터에 저장되고 그 때 opw 레지스터에서 목적지 레지스터의 주소인 w_rdest를 보내 원하는 레지스터에 값이 저장된다.

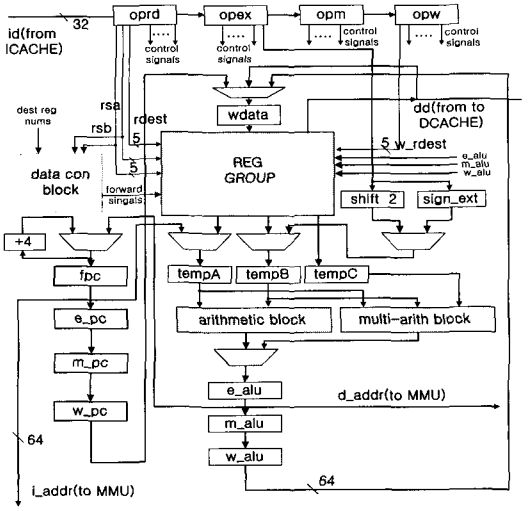

그림 12. 프로세서 데이터 패스
Fig. 12. Data path of processor.

그림 13은 multi-arith block의 내부 데이터 패스이다. 연산 블록은 크게 4 그룹으로 구성되어 있고 Arithmetic 블록에서는 add/sub, pixel distance, compare 명령어 그룹을 처리하며 Shifter 블록에서는 shift 명령어 그룹과 mmerge 명령어를 제외한 conversion 명령어를 처리한다. Multiplier 블록은 multiply 명령어 그룹을 처리하고 mmuladdh 명령어를 위한 Rearranger 그룹에서는 rearrange 명령어 그룹과 mmerge 명령어를 처리한다.

data con block에는 forwarding과 interlock 체크 logic이 있다. 각 단에서 파이프라인된 목적지 레지스터의 주소 값과 현재 D 단계의 명령어의 소스 레지스터 주소값인 rsa와 rsb신호를 입력으로 하여 발생시킨 forwarding 제어 신호를 REG GROUP으로 보낸다. 여

기에는 E 단계에서 계산된 값을 각 단에서 저장한 e_alu, m_alu, w_alu값들이 들어오고 있으므로 data con block에서의 제어 신호에 의해 forwarding할 것인지 rsa와 rsb에 의해 디코드된 레지스터 값을 읽을 것인지를 결정하여 tempA와 tempB에 저장한다. 메모리 읽기 명령어의 목적지 레지스터를 다음 명령어가 소스 레지스터로 사용하는 경우에 forwarding으로 해결이 안되므로 한 싸이클을 stall하기 위한 제어신호도 발생한다.

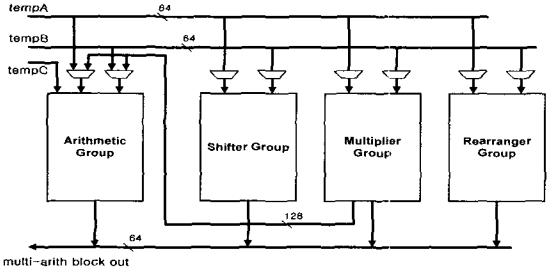

그림 13. multi-arith block 데이터 패스
Fig. 13. Data path of multi-arith block.

제어 전달 명령어가 패치된 PC 값을 저장하는 경우를 위해 PC값도 파이프라인된다. e_alu에서 fpc로 가는 값은 제어 전달 명령어의 목적지 주소를 나타낸다.

또한 d_addr은 데이터 주소 버스이므로 메모리 접근 명령어의 메모리 주소를 담고 있다. SAPRC 구조에 사용되는 SETHI 명령어와 제어 전달 명령어의 목적지 주소를 계산하기 위한 left shift by 2와 sign_ext의 모듈은 opex에서 발생시킨 제어신호에 의해 동작한다.

fpc에서 i_addr로 나가는 값은 제어 전달 명령어가 새로운 메모리 영역에서 명령어를 패치하기 위해서 주소 값이다. wdata 레지스터에는 메모리 읽기 명령어에 의한 데이터 값 dd, W 단계의 결과 값인 w_alu, PC값 중 하나를 선택하여 저장한다. dd는 입출력포트로 메모리 읽기 명령어처리 시는 입력으로 메모리 저장 명령어 처리 시는 출력포트로 사용된다.

2. 프로세서 모델링

제안하는 프로세서 구조는 Verilog-HDL로 행위 레벨에서 기술되었다. 총 8개의 모듈로 구성하였으며 각 모듈은 각 단계의 수행을 기술한 stage_fe, stage_de, stage_ex, stage_m, stage_w와 레지스터의 입출력을 기술한 regs, forwarding과 interlock을 체크하는 제어 신호 및 파이프라인 제어 신호를 발생하는 data_con, 그리고 stage_ex의 부 모듈로써 멀티미디어 명령어를 처리하는 multi_inst로 이루어져 있다. 그림 14에는 모듈 블록도를 나타내고 있으며 다음에서 각 모듈의 기능을 설명하였다.

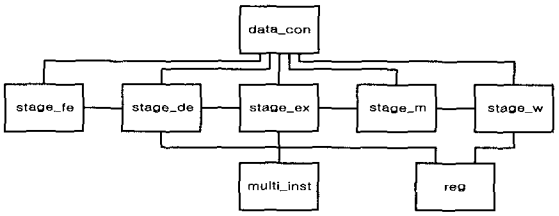

그림 14. 프로세서 모델링 모듈 블록도
Fig. 14. Module block diagram of processor modeling.

(1) stage_fe 모듈

명령어 캐시에서 주소 값에 의해 지정된 명령어를 패치하여 래지스터에 저장한다. 또한 다음 명령어를 패치하기 위한 주소 값을 생성하는데 분기명령어에 의한 전달인 경우는 계산된 주소 값을 명령어 주소 버스로 보내고 그렇지 않은 경우는 PC(Program Counter)값에 4를 더한다.

(2) stage_de 모듈

실행중인 명령어의 소스 레지스터와 목적지 레지스터의 주소 값을 디코드하며 특히 목적지 레지스터 값이 소스 레지스터로 사용되는 mpdist 명령어를 위한 제어 신호를 발생시킨다. stage_fe에서 받은 명령어 스트림을 stage_ex에 보내고 마찬가지로 디코드된 레지스터 주소 값 또한 계속 파이프라인되기 위해 E 단계인 stage_ex로 보낸다.

(3) stage_ex 모듈

stage_de에서 디코드되어 레지스터 파일인 regs에서 읽은 값은 그림 13의 tempa와 tempb 레지스터에 저장되어 있으므로 이 값을 사용하여 연산을 수행한다. 멀티미디어 명령어를 처리하는 경우는 부 모듈인 multi_inst에서 계산된 값을 stage_m으로 넘긴다. 마찬가지로 레지스터 주소, PC 값도 파이프라인을 위해 stage_m 모듈로 보낸다.

(4) stage_m 모듈

이 단계는 메모리에서 주소를 참조해서 얻은 데이터를 W 단계에서 받기 위한 단계로 볼 수 있다. 프로세서 안에서는 stage_ex 모듈에서 계산된 결과 값을 받아

서 다음 stage_w 모듈로 넘긴다. 또한 명령어 파이프라인을 위해서 E 단계에서 사용한 명령어 스트림 값을 넘긴다.

(5) stage_w 모듈

이것은 파이프라인의 맨 끝 단계의 모듈이며 주로 W 단계에서 레지스터에 저장할 제어 신호들을 발생시키고 목적지 레지스터의 주소를 계산한다.

(6) regs 모듈

이 모듈에서 범용레지스터를 규정하고 특별레지스터를 담고 있다. 범용레지스터는 총32개로 각각 64비트의 array형식으로 구성하였다. stage_w 모듈에서의 각종 쓰기 가능 신호에 따라 메모리 읽기 명령어에 의한 데이터, 리턴 주소 값, stage_ex에서 계산되어 파이프라인된 연산결과 값을 선택하여 저장한다. 레지스터에 값을 저장하는 시점은 W 단계의 중간에서 안정된 값을 가지므로 클럭의 falling시점이다. 또한 이 모듈에서는 stage_ex 모듈에서 사용될 각종 소스 오퍼랜드 값을 data_con 모듈에서의 제어신호에 의해 출력한다.

(7) data_con 모듈

현재 stage_de 모듈에서의 소스 및 목적지 레지스터의 주소 값과 그 전에 stage_de에서 디코드되어 각 단으로 파이프라인된 레지스터 주소 값을 입력으로 사용하여 E forwarding, M forwarding, W forwarding 제어 신호를 regs 모듈로 출력한다. 또한 interlock을 위한 hold 신호를 발생시킨다.

(8) multi_inst 모듈

stage_ex의 부 모듈로 멀티미디어 명령어를 처리하는 제어 신호를 발생하고 해당되는 명령어를 수행하여 결과 값을 stage_ex 모듈에 리턴한다.

Ⅳ. 멀티미디어 명령어 적용 및 시뮬레이션

1. 멀티미디어 명령어 적용

본 장에서는 제안한 멀티미디어 명령어를 멀티미디어 데이터 처리 시에 사용되는 몇 가지 알고리즘에 적용하여 멀티미디어 명령어를 사용하지 않은 경우 및 기존에 제시되었던 MAX-2, VIS, MMX의 멀티미디어 명령어를 사용한 경우에 명령어 수적인 측면에서 비교하였다.

프로그램 1의 코드는 1*N 행렬과 N*N 행렬의 내적을 수행하는 프로그램의 일부이다. intmat_a는 첫 행렬의 배열이름이며 inmat_b는 두 번째 행렬의 배열이다. outmat은 저장할 배열 값의 이름이다. 'N'은 행렬의 크기를 나타낸다. 만약 N이 4일 때 대부분의 수행시간은 ①에서 소비되며 이 부분은 멀티미디어 명령어를 사용하지 않을 때 42 = 16개의 곱셈명령어와 4*(4-1) = 12개의 덧셈 명령어가 필요하다. 이를 제안한 멀티미디어 명령어 집합에 적용하면 4개의 mmuladdh 명령어와 2개의 maddw 명령어로 구현가능하다.

```
short inmat_a[N], inmat_b[N][N], outmat[N], i, j;
for(i=0; i<N; i++){
    for(j=0; j<N; j++){
        outmat[i] += inmat_a[j] * inmat_b[j][i];--①
    }
}
```

프로그램 1. 벡터 내적
Program 1. Vector inner product.

그림 15는 N이 4이고 각 subword값들이 하프워드일 때에 제안된 멀티미디어 명령어를 적용한 수행을 보여준다. 두 번째 행렬의 1, 2열에 대한 곱셈을 수행하기 위해서 각 행렬의 값들을 조정하여 레지스터에 값을 적절히 배열한 후에 mmuladdh 명령어를 수행하면 첫째, 둘째 그리고 셋째, 넷째 subword들을 곱하여 더하게 되고 maddw 명령어로 계산 값들을 더하면 결과 행렬의 첫 번째와 두 번째 값을 구한다.

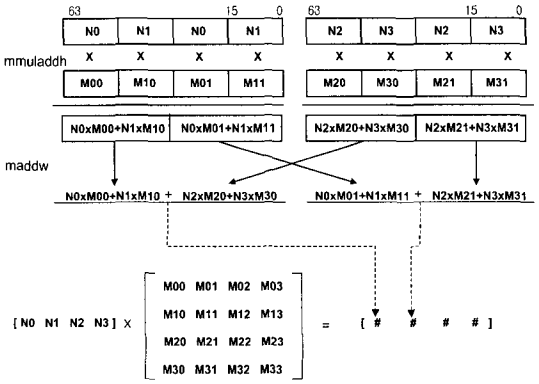

그림 15. 제안된 명령어 적용 예 - 벡터내적
Fig. 15. Application of proposed instructions - Vector inner product.

MAX-2에는 벙렬 곱셈 명령어가 존재하지 않으므로 일반 곱셈 명령어를 사용해야 되는데 이때에는 멀티미디어 명령어를 적용하지 않은 경우와 같은 수의 연산

을 수행하게 된다. 그러나 첫 번째 행렬의 값들이 2^k 인 경우에는 쉬프트 연산으로 곱셈을 수행할 수 있으므로 그림 16처럼 7개의 명령어로 수행이 가능하다.

MAX-2	VIS	MMX
n=2^sa일 경우, Ra=m00 m01 m02 m03 Rb=m10 m11 m12 m13 Rc=m20 m21 m22 m23 Rd=m30 m31 m32 m33 n0=(2^sa0) n1=(2^sa1) n2=(2^sa2) n3=(2^sa3) HSHL Ra,sa0,R1 HSHL Rb,sa1,R2 HSHL Rc,sa2,R3 HSHL Rd,sa3,R4 HADD R1,R2,Rx HADD R3,R4,Ry HADD Rx,Ry,Rz	%d0=m00 m01 m02 m03 %d1=m10 m11 m12 m13 %d2=m20 m21 m22 m23 %d3=m30 m31 m32 m33 %d4=n0 n0 n0 n0 %d5=n1 n1 n1 n1 %d6=n2 n2 n2 n2 %d7=n3 n3 n3 n3 fmul8sux16 %d0, %d4, %d8 fmul8ulx16 %d0, %d4, %d9 fpadd16 %d8, %d9, %d10 fmul8sux16 %d1, %d5, %d11 fmul8ulx16 %d1, %d5, %d12 fpadd16 %d11, %d12, %d13 fmul8sux16 %d2, %d6, %d14 fmul8ulx16 %d2, %d6, %d15 fpadd16 %d14, %d15, %d16 fmul8sux16 %d3, %d7, %d17 fmul8ulx16 %d3, %d7, %d18 fpadd16 %d17, %d18, %d19 fpadd16 %d10, %d13, %d20 fpadd16 %d16, %d19, %d21 fpadd16 %d20, %d21, %d22	mm1=n0 n1 n0 n1 mm2=m00 m10 m01 m11 mm3=n2 n3 n2 n3 mm4=m20 m30 m21 m31 mm5=m02 m12 m03 m13 mm6=m22 m32 m23 m33 Pmaddwd mm1, mm2 Pmaddwd mm3, mm4 Paddd mm1, mm3 Pmaddwd mm5, mm1 Pmaddwd mm6, mm3 Paddd mm5, mm6

그림 16. 다른 멀티미디어 명령어 적용 예 - 백터내적
Fig. 16. Application of other multimedia instructions - Vector inner product

VIS의 경우에는 fmul8sux, fmul8ulx, fpadd16의 3개 명령어를 사용하여 병렬적으로 16비트와 16비트의 곱셈 결과를 16비트로 저장할 수 있으므로 모두 15개의 명령어로 수행가능하다. MMX의 경우는 본 논문에서 제안한 결과와 같은 수의 명령어로 수행 가능하다. 표 2는 벡터 내적 수행 시에 행렬 크기에 변화함에 따라 각 경우에 따라 필요한 명령어 수를 추정한 결과이다.

표 2. 벡터 내적 명령어 수 비교
Table 2. The number of instructions used in vector inner product.

N	4	8	16	32
멀티미디어 명령어 적용안함	28	120	496	2016
멀티미디어 명령어 적용함	6	28	120	496
MAX-2(2^k인 경우)	28 (7)	120 (30)	496 (124)	2016 (504)
VIS	15	62	252	1016
MMX	6	28	120	496

프로그램 2는 크기가 N*N인 정방행렬의 전치행렬을 구현한 C 코드이다. mat는 입력 행렬이며 temp는 스왑(swap)에 필요한 임시 행렬이다. temp 배열에는 입력 행렬의 값이 똑같이 들어있어야 한다. 수행시 대부분의 시간은 ②의 연산부분에서 소요된다. N이 4일 때 멀티미디어 명령어를 사용하지 않으면 2*2+2 = 6개의 스왑 연산을 하므로 3*(2*2+2) = 18개의 데이터 이동 명령어가 필요하다. 제안한 멀티미디어 명령어 집합에 적용시키면 6개의 mmix 명령어로 수행할 수 있다.

```
int mat[N][N], temp[N][N], i, j;
for(i=0; i<N; i++){
    for(j=0; j<N; j++){
        mat[i][j] = temp[j][i]; ----②
    }
}
```

프로그램 2. 전치 행렬 변환
Program 2. Transpose matrix.

그림 17은 6개의 mmix명령어를 사용하여 8비트의 subword를 갖는 4*4의 행렬에서 전치 행렬을 구하는 수행 내용을 보여준다. 64비트의 r1, r2는 입력 레지스터이며 행렬의 1번째와 3번째 행은 r1에 저장되고 2번째의 4번째 행은 r2에 저장되어 연산을 수행한다. d1, d2, t1, t2의 임시 레지스터를 사용하여 연산 수행 후에는 r1과 r2에 subword의 위치가 바뀌게 된다.

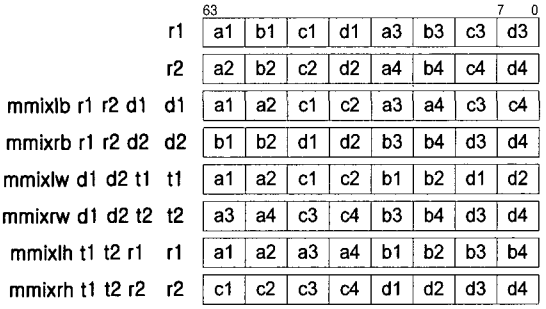

그림 17. 제안된 명령어 적용 예 - 전치행렬 변환
Fig. 17. Application of proposed instructions - Transpose matrix.

이 전치 행렬 변환 알고리즘을 다른 멀티미디어 명령어에 적용하면 그림 18과 같다. MAX-2의 경우는 8비트의 데이터연산 시에 병렬적으로 4개를 연산할 수 있으므로 각 행들을 4개의 레지스터에 저장하여 8개의 명령어로 수행한다. VIS의 경우는 32비트 레지스터에 8비트 데이터 4개씩을 저장하여 fpmerge 명령어로 64비트 레지스터에 8개의 8비트 데이터를 만들어서 사용한다. MMX의 경우에는 6개의 명령어로 적용 가능하다. 표 3은 다양한 행렬 크기에 따른 전치 행렬 변환의 명령어 수를 각 경우에 대해서 비교하고 있다.

MAX-2	VIS	MMX
r1=a1 b1 c1 d1 r2=a2 b2 c2 d2 r3=a3 b3 c3 d3 r4=a4 b4 c4 d4	s1=a1 b1 c1 d1 s2=a2 b2 c2 d2 s3=a3 b3 c3 d3 s4=a4 b4 c4 d4	mm1=a1 b1 c1 d1 a3 b3 c3 d mm2=a2 b2 c2 d2 a4 b4 c4 d4

MIXH,L r1, r2, d1
MIXH,R r1, r2, d2
MIXH,L r3, r4, d3
MIXH,R r3, r4, d4
MIXW,L d1, d3, r1
MIXW,L d2, d4, r2
MIXW,R d1, d3, r3
MIXW,R d2, d4, r4
	fpmerge %s1, %s3, %d20
	fpmerge %s2, %s4, %d21
	read_hi %d20, %s5
	read_hi %d21, %s6
	fpmerge %s5, %s6, %d22
	read_lo %d20, %s7
	read_lo %d21, %s8
	fpmerge %s7, %s8, %d23
		Movq mm0, mm1
		Punpckhbw mm1, mm2
		Punpcklbw mm0, mm2
		Movq mm3, mm1
		Punpckhwd mm1, mm0
		Punpcklwd mm3, mm0

그림 18. 다른 멀티미디어 명령어 적용 예 - 전치행렬 변환

Fig. 18. Application of other multimedia instructions - Transpose matrix.

표 3. 전치 행렬 수행 시 명령어 수 비교

Table 3. The number of instructions used in transpose matrix.

N	4	8	16	32
멀티미디어 명령어 적용안함	18	84	360	1488
멀티미디어 명령어 적용함	6	24	96	384
MAX-2	8	44	200	496
VIS	8	72	288	1152
MMX	6	36	144	576

프로그램 3은 chroma keying 기법을 기술한 프로그램의 일부이다. 배열 x는 입력 이미지 값이고 여기에서 파란색인 영역을 배경 정보를 갖는 배열 y 이미지로 대치하여 배열 new_image에 저장한다. 여기에서의 각 픽셀 값은 8 비트 흑백이미지이며 i_size는 전체 이미지 내의 픽셀 수를 나타낸다. ③ 라인에서 입력 이미지인 x에서 파란색 영역을 추출하여 해당된 픽셀들은 배경 이미지인 y 값을 대치시킨다. 파란색 영역이 아닌 경우는 원래의 이미지 x 값을 대치시킨다. 멀티미디어 명령어를 사용하지 않고 실행하는 경우에는 한 개의 픽셀을 계산하는데 compare 명령어, branch 명령어, mov 명령어를 가지고 2, 3개의 명령어가 필요하므로

```
int i_size, i;
int new_image[i_size], x[i_size], y[i_size];
for(i=0; i<i_size; i++){
    if(x[i]==BLUE) new_image[i] = y[i];   ③
    else new_image[i] = x[i];
}
```

프로그램 3. chroma keying

Program 3. Chroma keying.

평균 2.5/(1픽셀)개의 명령어가 필요하다. 멀티미디어 명령어를 적용하는 경우에는 8개의 픽셀 당 5개의 명령어로 구현 가능하므로 0.625/(1픽셀)개의 명령어가 필요하다.

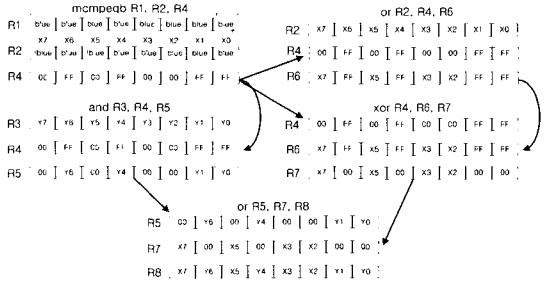

그림 19. 제안된 명령어 적용 예 - chroma keying

Fig. 19. Application of proposed instructions - chroma keying.

그림 19는 앞에서 설명한 프로그램에서 i_size가 8, 즉 이미지 내에서 처리할 픽셀 수가 8개인 경우를 제안된 멀티미디어 명령어로 수행할 때의 과정을 나타내고 있다. 입력 이미지는 레지스터 R2에 저장되어 있고 mcmpeqb 명령어에 의해서 파란색부분만 추출하는 마스크를 생성하여 R4에 저장한다. 생성한 마스크와 배경 이미지 값을 갖는 레지스터 R3과 일반 명령어인 AND에 의해서 입력 이미지의 파란색 부분을 배경으로 대치시킬 값을 레지스터 R5에 저장한다. 마스크 레지스터 R4를 사용하여 or, xor 명령어를 거치게 되면 파란색 이외의 부분만을 추출한 이미지 정보가 레지스터 R7에 저장된다. 마지막으로 or 명령어로써 입력 이미지에 배경 이미지를 삽입하게 된다.

MAX-2	VIS	MMX
1픽셀에 대하여 r1=blue r2=x0 r3=y0	1픽셀에 대하여 %d1=blue %d2=x0 %d3=y0	8픽셀에 대하여 mm1=blue mm3=x7 x6 x5 x4 x3 x2 x1 mm4=y7 y6 y5 y4 y3 y2 y1
loop1: CMPB,<> r1, r2, loop2 loop2: MOVB,TR r3, r2, loop1	subcc %d1 %d2 %d4 movrnz %d4 %d3 %d2	Pcmpeqb mm1, mm3 Pand mm4, mm1 Pandn mm1, mm3 Por mm4, mm1

그림 20. 다른 멀티미디어 명령어 적용 예 - chroma keying

Fig. 20. Application of other multimedia instructions - chroma keying.

MAX-2 경우를 chroma keying 기법에 적용 시에 병렬적으로 검사할 수 있는 명령어가 없으므로 그림 20과 같이 픽셀 1개당 1개 혹은 2개의 일반 명령어로 구

현할 수밖에 없다. VIS의 경우에는 subword의 조건을 병렬적으로 검사할 수 있는 명령어는 존재하나 결과를 1비트로 저장하므로 마스크 데이터를 생성할 수 없으므로 조건을 검사하는 명령어와 그 결과로 레지스터의 데이터를 이동하는 2개의 명령어로 구현 가능하다. MMX의 경우에는 그림 19에서 not 명령어와 and 명령어를 동시에 수행할 수 있는 Pandn 명령어가 존재하므로 제안된 명령어보다 8개의 픽셀 계산 시에 1개 더 적은 명령어 수로 구현한다. 표 4에는 N*N 8비트 흑백 이미지에 대하여 chroma keying을 수행할 때 각 경우에 대하여 명령어의 수적인 측면에서 비교한 결과가 나타나 있다.

표 4. chroma keying 수행 시 명령어 수 비교

Table 4. The number of instructions used in chroma keying.

N	8	16	64	256
멀티미디어 명령어 적용안함	160	640	10240	163840
멀티미디어 명령어 적용함	40	160	2560	40960
MAX-2	96	384	6144	98304
VIS	128	512	8192	131072
MMX	32	128	2048	32768

프로그램 4는 motion estimation을 수행하는 C 코드의 일부분인데 이미지의 픽셀 값은 8비트의 흑백으로 가정한 것이다. src1은 현재 이미지의 픽셀의 포인터이고 src2는 이전 이미지의 픽셀 포인터이다. 대부분의 수행시간은 for 루프 부분의 ④에서 절대값을 구하는 부분과 각 포인터들을 증가시키는 부분에서 소비되며

```
accum = 0;
for(i=0; i<16; i++){
    for(j=0; j<16; j++){
        accum += abs(*src1 - *src2);        ④
        src1++; src2++;
    }
}
src1Ptr = src1 = src1Ptr + stride1;
src2Ptr = src2 = src2Ptr + stride2;
```

프로그램 4. motion estimation

Program 4. Motion estimation.

한 개의 픽셀을 처리하는 데 사용되는 명령어는 멀티미디어 명령어를 사용하지 않은 경우 sub 명령어, compare 명령어, branch 명령어, negate 명령어, add 명령어와 2개의 inc 명령어를 가지고 평균 6개의 명령어가 필요하다. 즉, 6/(1픽셀)개의 명령어로써 처리한다. 멀티미디어 명령어를 적용할 경우는 8개의 픽셀을 처리하는데 한 개의 명령어 가능하므로 0.125/(1픽셀)개의 명령어가 필요하다.

그림 21과 같이 MAX-2를 사용하면 4개의 8비트 데이터들을 총 32개의 레지스터에 저장한 후 각각의 subword의 절대값을 루프에서 누적하여 계산되며 이 4개의 subword를 마지막에 각각 더해주게 된다. MMX의 경우에는 16개의 명령어로 가능하며 VIS는 본 논문에서 제안하는 명령어로 구현했을 때와 같은 수의 명령어로 수행 가능하다. 표 5에는 motion estimation 처리 시에 절대값 계산 루틴에서 사용되는 명령어 수를 멀티미디어 명령어를 사용하지 않은 경우와 사용한 경우를 비교한 것이다.

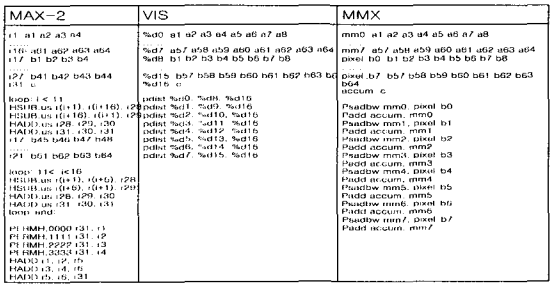

그림 21. 다른 멀티미디어 명령어 적용 예 - motion estimation

Fig. 21. Application of other multimedia instructions - motion estimation.

표 5. motion estimation 수행 시 명령어 수 비교

Table 5. The number of instructions used in motion estimation.

N	8	16
멀티미디어 명령어 적용안함	384	1536
멀티미디어 명령어 적용함	8	32
MAX-2	71	263
VIS	8	32
MMX	16	64

2. 명령어 시뮬레이션

앞에서 적용한 멀티미디어 명령어를 검증하기 위해

서 어셈블리 라인에서 명령어들의 스티뮬러스 벡터 (stimulus vector) 값을 생성하였다. 스티뮬러스 벡터 값들은 벡터 내적과 전치 행렬 연산의 알고리즘을 위한 모듈과 chroma keying 및 motion estimation을 위한 모듈의 2개의 부 프로그램으로 구성하였다.

그림 22는 시뮬레이션 환경을 나타낸다. 제 3장에서 설명한 프로세서 모듈이 processor core에 해당되고 멀티미디어 명령어를 포함한 2개의 부 프로그램이 각각 multimedia inst cache1과 multimedia inst cache2가 된다. 멀티미디어 명령어가 아닌 일반 명령어의 스티뮬러스 벡터 값들은 general inst cache에 저장되어 있다. 각각의 명령어 캐시와 데이터 캐시, 클럭 및 리셋 발생 부분은 역시 Verilog-HDL로 기술하였으며 CADENCETM 툴을 사용하여 시뮬레이션하였다.

그림 23은 전치 행렬과 벡터 내적의 알고리즘을 수행한 결과로 시간 3960에서 전치 행렬의 처음 명령어의 W 단계인 opw가 시작되어 시간 4200에서 마지막 명령어의 W 단계가 종료되어(① 참조) 8개의 명령어 실행 후 reg1, reg2, reg3, reg4의 값들이 변화됨을 알 수 있다(② 참조). 벡터 내적의 경우 시간 4420에서 첫 번째 명령어의 W 단계가 시작되어 시간 4560에서 마지막 명령어의 W 단계가 종료되면서(③ 참조) 6개의 명령어 수행 후 reg14에 원하는 결과 값을 저장함을 확인하였다(④참조).

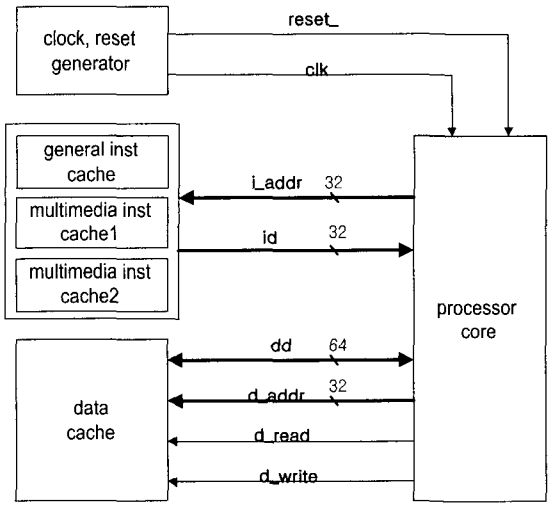

그림 22.시뮬레이션 환경
Fig. 22. Simulation environment.

그림 24는 chroma keying과 motion estimation의 경우를 시뮬레이션한 결과이다. 시간 600에서 시작한 chroma keying 적용 명령어들은 시간 1400에서 끝나면서(⑤ 참조) 40개의 명령어 실행 후 reg8에 값이 변화하면서 reg1부터 reg8까지에 정확한 값들이 저장됨을 확인하였고(⑥ 참조) reg1에서 reg16까지의 입력 값들로 절대값들의 합을 계산하는 motion estimation의 경우는 시간 1740에서 시작하여 시간 1900에 끝나(⑦ 참조) 8개의 명령어만에 결과 값인 39를 reg31에 저장하였다(⑧ 참조).

V. 결론

본 논문에서는 범용 프로세서 안에서 멀티미디어 정보를 효율적으로 처리하기 위해 필요한 멀티미디어 명령어와 이를 처리하기 위한 프로세서의 구조를 제안하였다. 제안한 멀티미디어 명령어는 연산 종류에 따라 8개의 그룹에 총 56개의 명령어로 구성되며, 처리하는 데이터는 64비트 데이터 안에서 8비트의 8개 바이트, 16비트의 4개 하프워드, 32비트의 2개 워드를 처리한다. 모두 각각의 subword를 병렬적으로 처리하는 특성을 지닌다.

프로세서는 SPARC V.9를 기반으로 하여 명령어 캐시와 데이터 캐시를 따로 사용하는 하바드 구조를 채택하고 시간 정적 제어 방식의 5단 파이프라인 구조를 갖는다. 이를 Verilog-HDL로 모델링하고 CADENCETM 툴을 사용하여 시뮬레이션을 수행하였다. 명령어 시뮬레이션을 통해 명령어들의 행위를 검증한 뒤 간단한 멀티미디어 응용 프로그램을 C 코드로 작성하여 제안된 프로세서의 명령어로 어셈블 한 뒤 시뮬레이션하여 멀티미디어 명령어를 사용하지 않은 경우와 비교하였다.

본 논문에서 기술된 프로세서 모듈은 PDA, 팜탑 컴퓨터 등의 각종 소형 멀티미디어 단말기 내의 코어로 사용될 수 있을 뿐만 아니라 멀티미디어 PC, 그래픽 워크스테이션 같은 멀티미디어 시스템의 코어 프로세서 설계나 개발의 기초 자료로 활용될 수 있을 것이다.

참 고 문 헌

[1] Paul Kalapathy, "Hardware-Software Interactions on Mpact", *IEEE Micro,* pp.20~26,

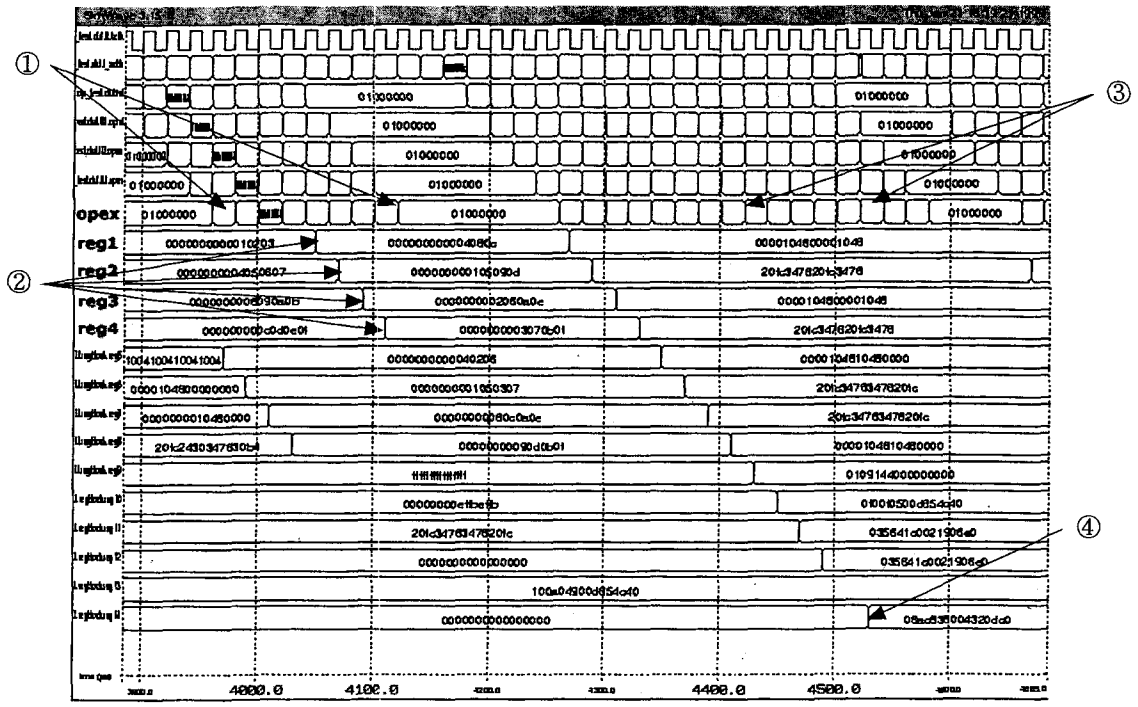

그림 23. 전치 행렬 및 벡터 내적 시뮬레이션 결과
Fig. 23. Simulation waveform of transpose matrix and vector inner product.

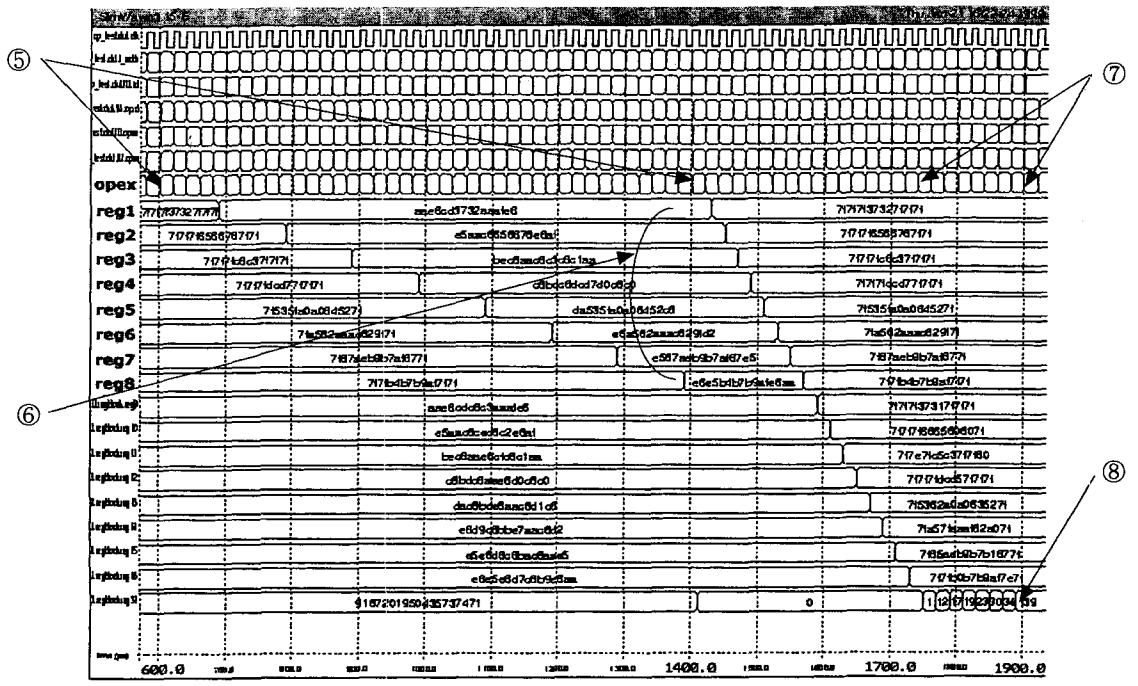

그림 24. chroma keying 및 motion estimation 시뮬레이션 결과
Fig. 24. Simulation waveform of chroma keying and motion estimation.

March/April, 1997.

[2] Ruby B. Lee and Michael D. Smith, "Media Processing : A New Design Targe", *IEEE Micro*, pp.6~9, Aug, 1996.

[3] Ruby B. Lee, "Subword Parallelism MAX-2", *IEEE Micro*, pp.51~59, Aug, 1996.

[4] Lavi A. Lev, et al., "A 64-b Microprocessor with Multimedia Support", *IEEE J. Solid-State Circuits*, Vol. 30, No. 11, pp.1227~1238, Nov, 1995.

[5] *UltraSPARC User's Manual*, SUN microsystems, July, 1997.

[6] *Intel Architecture Software Developer's Manual Volume 1 : Basic Architecture*, Intel Corporation, 1997.

[7] Alex Peleg and Uri Weiser, "MMX Technology Extension to the Intel Architecture", *IEEE Micro*, pp.42~50, Aug, 1996.

[8] *The SPARC Architecture Manual Version 8*, Prentice Hall, 1993.

[9] *The SPARC Architecture Manual Version 9*, Prentice Hall, 1994.

저 자 소 개

吳 明 勳(正會員)

1974년 8월 19일생. 1997년 전남대학교 컴퓨터공학과 학사. 1999년 전남대학교 컴퓨터공학과 석사. 1999년 ~ 현재 광주과학기술원 정보통신공학과 박사과정. 관심분야는 고성능 프로세서 구조, 비동기회로 및 시스템 레벨 설계, GALS 시스템, 저전력 회로 설계 등

朴 性 模(正會員)

1956년 3월 7일생. 1977년 서울대학교 전자공학과 학사. 1979년 한국과학원 전기 및 전자공학과 석사. 1979년 ~ 1984년 한국전자기술연구소 설계개발부 연구원, 선임연구원. 1988년 미국 노스캐롤라이나주립대 전기 및 컴퓨터공학과 박사. 1988년 ~ 1992년 미국 올드도미니언대 전기 및 컴퓨터공학과 조교수. 1992년 ~ 현재 전남대학교 컴퓨터공학과 조교수, 부교수, 교수. IEEE senior member. 관심분야는 컴퓨터 구조, 마이크로프로세서, DSP, 영상압축/처리, VLSI 시스템 설계 등

李 東 翊(正會員)

1958년 12월 12일생. 1985년 영남대학교 전기공학과 학사. 1989년 일본 Osaka 대학 전자공학과 석사. 1993년 일본 Osaka 대학 전자공학과 박사. 1993년 ~ 1994년 University of Illinois 객원 연구원. 1990년 ~ 1995년 일본 Osaka 대학 문부교관. 1995년 ~ 현재 광주과학기술원 정보통신공학과 조교수, 부교수. 관심분야는 비동기 시스템 CAD/설계, 자율 분산 시스템, 프로토콜 공학, 페트리넷 이론, 정형 기법, 병행시스템 설계 및 분석 등