44
논문번호 08-02-07

Design of a Variable-Length Instruction based on a OpenGL ES 2.0 API

OpenGL ES 2.0 API 기반 가변길이 명령어 설계

Kwang-Yeob Lee

이광엽

Abstract

The Khronos group releases OpenGL ES 2.0 API specification bringing streamlined shader programming to graphics processor of embedded system. For this reason, the mobile devices have need of graphics processor for supporting a OpenGL ES 2.0 API. We need to extend instruction's length to support OpenGL ES 2.0 API, so it needs more memory size. In this paper, we propose a new instruction format that offers availability for use the instructions. This proposed instruction adopt a variable length method and unit instruction architecture. This proposed instruction architecture that support to OpenGL ES 2.0 API has consist of 32bit unit instructions up to 4 which can be combined for embellishing each other. Therefore, it can execute flexible instruction combination and reduce waste of instruction fields.

요 약

최근 Khronos에서 OpenGL ES 2.0 API 표준을 배포 하면서 임베디드 시스템의 그래픽 프로세서에서 능률적인 쉐이더 프로그램이 가능하게 되었다. 그 결과 모바일 기기에서도 OpenGL ES 2.0을 지원하는 그래픽 프로세서를 요구하게 되었다. OpenGL ES 2.0을 지원하기 위해서 명령어의 길이의 증가가 요구되고, 이는 메모리 용량의 증가를 초래한다. 본 논문에서는 효율적으로 명령어를 사용하는 새로운 명령어를 제안한다. 이 명령어는 가변 길이 방법과 유닛구조를 채택한 명령어 구조이다. 제안된 명령어 구조는 OpenGL ES 2.0 API를 지원하고 명령어 필드 낭비를 줄일 수 있도록 최대 4개의 32비트 유닛 명령어가 가변적으로 조합되어 수행된다.

Keywords : *Variable Length Instruction Words, Shader, Graphics OpenGL*

Ⅰ. 서 론

일반적인 SIMD 프로세서는 그 명령어 필드에 어떤 연산기와 레지스터들을 사용하는지 나타내고 있으며, 빠른 명령어 디코딩을 위해 대부분 각 필드 영역이 고정되어 있다.

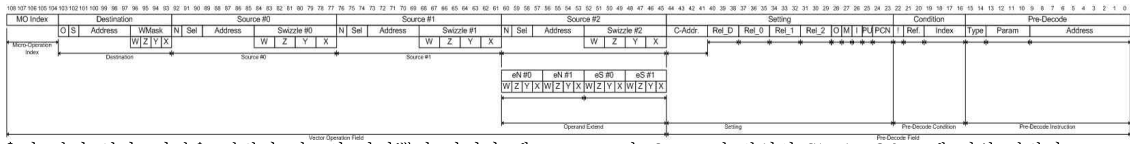
프로세서의 발전과 그래픽 기술의 발전에 따라 더욱더 현실적이고 화려한 효과를 위하여 SIMD 명령어 구조를 통하여 그래픽 과정을 처리 할 때 많은 양의 데이터를 몇 개의 동일한 명령어로 처리를 하게 된다. 더욱 발전된 그래픽 처리를 위한 Shader에서도 이와 같은 SIMD 명령어 구조를 사용한다.

OpenGL ES 2.0 이나 DirectX Shader 3.0 버전에는 기존의 기능이외에 Dynamic Branch / Looping 을 포함한 다양한 기능이 추가되었다. 따라서 명령어에 표현되어야 할 정보들이 많아지게 되고, 이를 다 표현 할 수 있는 고정 길이의 명령어를 만들기 위해선 명령어의 길이가 길어져야 한다.

명령어의 길이가 고정된 긴 명령어 구조에서 주로 사용되는 명령어에는 많은 필드들이 사용되지 않는 상태로 있는 경

* 회원, 서경대학교 컴퓨터공학과
(Dept. of Computer Engineering Seokyeong Univ.)

※ 본 논문은 지식경제부 시스템반도체 2010사업 지원으로 제작되었으며 IDEC지원장비를 활용하였습니다.



우가 많이 있다. 이것은 명령어 필드의 낭비뿐만 아니라 메모리의 낭비나 패치로 인한 속도 저하도 야기 시킬 수 있다.

본 논문에서는 위와 같은 명령어 필드 낭비를 줄일 수 있도록 4개의 32bit 유닛 명령어가 가변적으로 조합되어 수행되는 Variable Length Instruction 명령어 구조를 제안한다.

Ⅱ. 기존 방식의 OpenGL ES 2.0 명령어 구조

기존의 그래픽스 명령어는 고정된 SIMD 구조의 RISC타입의 명령어 구조가 많았다. 이러한 고정된 명령어는 단순하고 설계가 쉽다는 장점이 있으나 명령어 표현상에 제약이 많고 향후 명령어 추가에 대하여 취약하다는 문제점이 있었다.

OpenGL ES 2.0 의 Shader 3.0 모델로 발전하면[그림 1]과 같이 Dynamic-branch, Looping 과 같은 기능이 추가됨에 따라 Call Stack, Loop Counter, Predicate Register등과 같은 많은 레지스터가 추가적으로 요구된다. 이러한 구조에서 기존의 방식인 고정된 명령어 형식을 유지하고 Shader 3.0의 모든 기능을 표현하려면 [그림 2]와 같이 최소 109비트로 확장된 구조를 가지게 된다. 이러한 확장된 구조는 실제 사용되는 명령어 필드에 비하여 낭비되는 필드가 더 많아지며 그만큼 명령어 표현과 구동 효율이 떨어지게 된다.

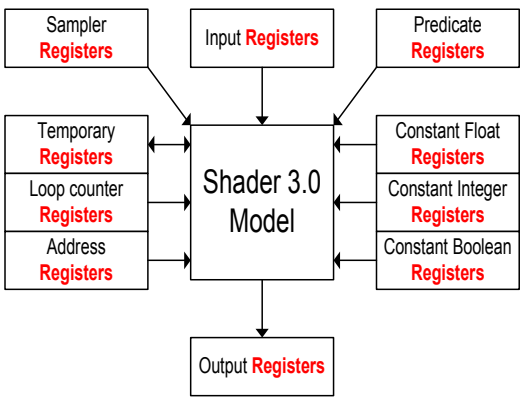


그림 1. 쉐이더 모델 3.0의 레지스터
Fig. 1. Registers of Shader Model 3.0

그림 2. 고정 방식의 Shader 3.0 모델 지원 명령어
Fig. 2. Fixed Length - Instruction format for a shader 3.0 model

이러한 고정된 길이의 명령어의 단점을 보완하기 위하여 본 논문에서는 메모리 효율을 높이고 연산에 필요한 명령어만을 해독하여 효과적으로 사용할 수 있는 새로운 명령어구조인 가변길이 명령어 VL-IW(Variable Length - Instruction Words)를 제안한다.

Ⅲ. 개선된 OpenGL ES 2.0 명령어 구조

1. 가변 길이 명령어 형식

본 논문에서 제안하는 VL-IW(Variable Length - Instruction Words)방식의 가변 길이 명령어는 [그림 3]과 같이 기존의 109비트로 되어 있는 고정된 방식의 명령어와는 다르게 32비트로 되어있는 유닛 명령어(Unit Instruction)를 최대 4개까지 조합할 수 있는 구조를 제안한다.

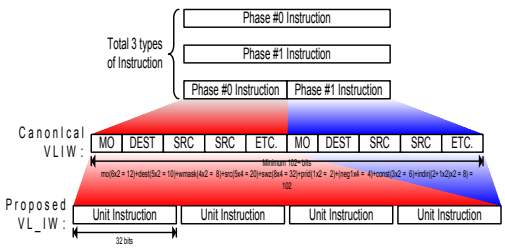


그림 3. 가변 길이 명령어
Fig. 3. VL-IW(Variable Length - Instruction Word)

[그림 3]에서 제안하는 각 하나의 유닛 명령어는 기본적으로 [그림 4]와 같이 32비트 형식으로 나타낼 수 있다.

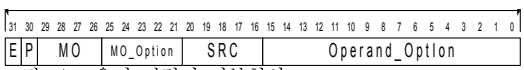


그림 4. 유닛 명령어 기본형식
Fig. 4. Unit Instruction basic format

2. 제안하는 명령어 셋

제안하는 명령어는 [그림 5]와 같은 명령어 구성을 가지고 있으며, 명령에는 크게 연산(Arithmetic) / 수식(Coordinate) / 프로세스(Process) 명령어그룹으로 나누어진다. 이러한 명령어들은 OpenGL ES 2.0과 DirectX Shader 3.0을 지원하게끔 만들어진 명령어이다. 연산 명령어의 경우 특별히 기존에 있던 기본적인 명령어 외에 유닛 구조를 효과적으로 지원하는 Additional Istruction을 명령어를 만들어 사용한다. MOV, MVS 명령을 제외한 나머지 연산 명령어에 대하여 동일한 scalar 위치의 연산을 사용하지 않는 범위에서 중복 선언이 허용된다.

Instruction Type		Phase #0		Phase #1	
		Caronical	Additional	Caronical	Additional
Arithmetic	Move	MOV (move)	MVS (move status registers)	MOV (move)	MVS (move status registers)
	Base Operation	ADD (add) MUL (multiply)	reserved reserved	ADD (add) MUL (multiply)	reserved reserved
	Compare	CMP (compare)	reserved	CMP (compare)	reserved
	Special Function	RCP (Reciprocal) MAN (Mantissa)	RSQ (Reciprocal square root) EXP (Exponent)	RCP (Reciprocal) MAN (Mantissa)	RSQ (Reciprocal square root) EXP (Exponent)
	Setting	FLR (Floor) CONV (data type conversion)	FRC (Fraction) reserved	FLR (Floor) CONV (data type conversion)	FRC (Fraction) reserved
	Logical	AND (bit logic and) XOR (bit logic exclusive or)	OR (bit logic or) SHF (logical or arithmetic shift)	AND (bit logic and) XOR (bit logic exclusive or)	OR (bit logic or) SHF (logical or arithmetic shift)
	reserved	reserved reserved		reserved reserved	
Coordinate		PRED (predicate coordinate) ADDR (address coordinate)		reserved reserved	
Process		reserved reserved		BRC (branch) MEM (memory operation)	

그림 5. 명령어 구성
Fig. 5. Instruction Set

명령어 형식은 [그림 6]에서와 같이 Arithmetic, Coordinate, Process 명령어 타입에 따라 다른 구조로 설계하였다. [그림 6]에서 보듯이 Arithmetic Instruction 구조는 Main 과 Sub로 나뉘는 것을 알 수 있다. 이는 유닛 명령어를 정렬하는 순서에 따라 Main인지 Sub인지 구분을 한다.

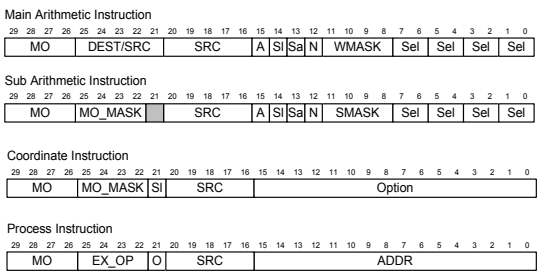


그림 6. 유닛 명령어 3가지 형식
Fig. 6. 3 Types of Unit Instruction Format

연산 명령어에 이와 같은 방식으로 Main과 Sub를 구분함으로써 각 유닛 명령어는 operand가 2개 이지만 sub를 수식하여 3 operand연산을 할 수 있고, 기존의 고정된 명령어에서는 할 수 없었던 하나의 명령어에 각 유닛마다 다른 연산을 하는 형식의 다양한 연산을 수행하는 것도 Main과 Sub를 나누어서 가능하게 하였다. 각 유닛마다 다른 연산으로 여러 조합을 하게 되면 제안하는 명령어는 수천가지의 명령어를 조합할 수 있다. 또한 연산을 할 때 제안하는 명령어는 한 유닛 명령어로 2 operand 연산을 할수 있다. 그 결과 단순한 하나의 덧셈과 곱셈같은 명령어는 32비트의 유닛 명령어 하나만 사용가능하다. 이것은 고정된 명령어 필드를 다 사용해야하는 기존의 명령어와는 다르게 메모리를 효율적으로 사용할 수 있다.

OpenGL ES 2.0에서는 Dynamic-branch, Looping 같은 기능을 추가함으로 흐름 제어에 관한 명령어를 필요로 하게 되었다. 따라서 유닛 명령어에 [그림 6]의 Coordinate 와 Process operation 형식을 만들어 사용한다. Coordinate 와 Process 연산 명령어는 [그림 5]에서 보듯이 각 Phase #0과 Phase #1에 구현되어있다. Phase #1의 Branch나 Memory 명령어가 Phase #0의 Predicate와 Addr명령어의 결과를 바로 수식받아서 사용하기 때문에 기존의 명령어로 할 수 없었던Dynamic/nested Branch를 단일 명령어로 사용할 수 있다. 그래픽 프로세서에서는 여러 데이터를 몇 가지의 명령어로 동일한 연산을 반복해서 처리하기 때문에 위와 같은 방법으로 제안하는 명령어를 사용하여 Loop의 명령어를 줄여 성능을 향상 시킬 수 있다.

3. 유닛 명령어 기반 조합 명령어 구조 설계

[그림 7]에서 보듯이 한 명령어는 1개에서 최대 4개까지의 유닛 명령어로 구성된다. 유닛 명령어의 최상위 비트 'E'(End of Instruction) 필드는 한 명령어가 이루는 유닛 명령어의 개수를 정한다.'P'(Phase) 필드는 유닛 명령어가 해석될 Phase를 의미한다. 명령어 조합은 [그림 7]에서와 같이 최대 8가지 종류의 유닛 명령어 조합이 이루어 질 수 있다. 또한 반드시 Phase 1의 유닛 명령어는 Phase 0의 명령어 뒤에 존재하여야 하며 각 Phase는 최대 2개의 유닛 명령어를 선언할 수 있다.

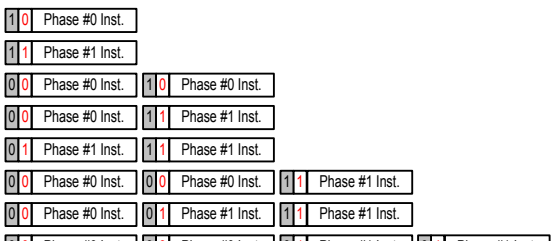


그림 7. 유닛 명령어의 기본 조합
Fig. 7. Basic combinations of unit instructions

iv. 제안하는 명령어 정렬 방법

명령어가 명령어 저장 공간에서 프로세서로 전달되기 위해서 PC(Program Counter)를 어드레스로 사용하여 명령어 저장 공간에 적절히 입력되어야 한다. [그림 8]은 가변적으로 저장되어 있는 명령어가 어떠한 방식으로 정렬이 되는지를 예를 들어 나타내고 있다. [그림 8]과 같이 명령어 저장 공간은 4개의 Bank로 나누어 유닛 명령어들이 순차적으로 저장되어 있다. 이것을 Instruction Fetch 단계에서 순차적으로 4개를 읽어 와서 프로세서로 전달하는 것이다. 예에서는 현재 PC를 6으로 가정하고 Fetch 하는 과정을 보이고 있다. 우선 현재 PC가 6 이므로 그에 해당하는 유닛 명령어부터 4번째 유닛 명령어인 9번째까지의 명령어를 읽어온다. 그리고 읽어온 유닛 명령어들의 END 비트를 확인하여 비트가 1인 8번째 유닛 명령어 까지를 조합하여 사용하고, 다음 PC를 9로 세팅하여 다음 수행 명령어를 읽을 때에 9번째 명령어부터 읽어 들이도록 한다. 이 방법으로 필요한 유닛 명령어만 읽어서 효율적이 사용이 가능하게 되었다. 이 방법으로 정렬된 상태에서 연산 명령어는 순서적으로 Main과 Sub를 구분 지어서 처리한다.

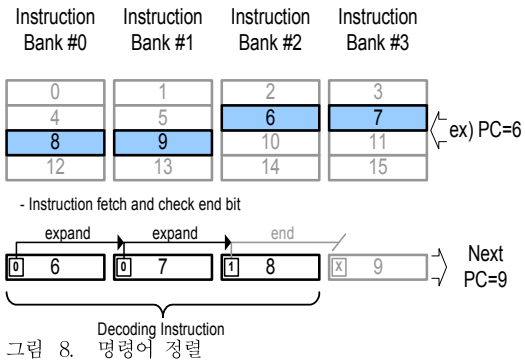


그림 8. 명령어 정렬
Fig. 8. Instruction Alignment

Ⅳ. 가변 길이 명령어 처리 데이터 패스 설계

1. 데이터 패스

본 논문에서 제안하는 가변 길이 명령어를 처리하기 위한 쉐이더 프로세서의 데이터 패스 구조를 [그림 9]과 같이 나타낼 수 있다.

[그림 9]에서 보이는 바와 같이 쉐이더의 데이터 패스 구조는 2 Phase 구조로 되어있다. 각 Phase마다 유닛명령어를 지정할 수 있으며, 하나의 Phase에 Main 유닛 명령어와 Sub 유닛 명령어로 2개의 유닛 명령어를 처리가 가능하며, 이로 인하여 하나의 명령어 조합에 최대 4개까지의 유닛 명령어 조합이 가능하다.

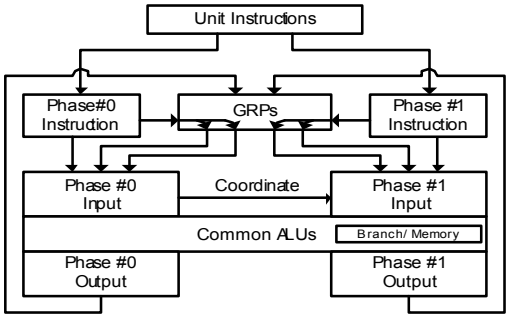


그림 9. 쉐이더 데이터 패스 구조
Fig. 9. Architecture of a Shader data path

그러나 모바일 기기 기반의 쉐이더 설계를 위하여 면적을 고려하여 데이터 패스에서 가장 큰 면적을 차지하는 ALU는 하나의 ALU만을 가지도록 설계하여 두 개의 Phase가 공용으로 사용하도록 설계 되어있다. 그러므로 각 Phase에서는 다른 Phase와 연산이 중복되어서는 안 된다. 또한 GPRs(General Purpose Registers) 하나를 사용하여 각 Phase에서 연산에 필요한 다양한 레지스터나 연산 결과가 저장되는 레지스터 역시 공용으로 사용한다.

Phase #0과 Phase #1은 모두 연산 명령어 수행이 가능하고, Branch와 같은 흐름제어 명령어와 Memory 명령어는 Phase #1에서 전담하도록 되어있다. 이러한 Phase #1의 흐름제어 명령어나 메모리 명령어의 보다 다양하고 복잡한 표현을 위하여 Phase #0에서는 Phase #1의 명령어를 수식 가능하도록 되어있다. 이로써 Indirect Address나 Dynamic Branch, Looping과 같은 복잡한 명령어도 사용이 가능하도록 되어있다.

Ⅴ. 기능 검증 및 결과

설계된 유닛 명령어의 기능을 검증하기 위해서 OpenGL ES 2.0 API에서 요구되는 명령어를 사용하였다. [그림 10]에서는 유닛 명령어를 조합해서 다양한 연산을 수행하는 과정을 보인다. 이 중 다섯 번째 조합의 경우에는 MUL A, MOV, ADD, RCP 네 개의 명령어가 사용이 되고 있는데 기존 명령어 구조에서는 각각 4개의 명령어를 통해서 수행이

이루어져야 하나 가변길이 명령어 구조에서는 각 Phase에 Main 과 Sub로 나뉘어서 하나의 명령어로 조합이 가능함을 보여주고 있다.

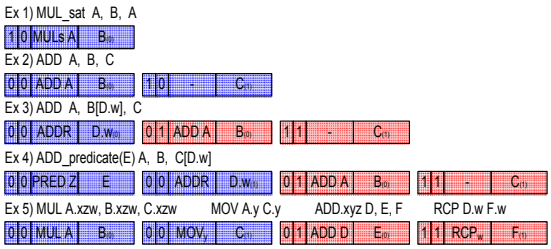


그림 10. 유닛 명령어의 조합
Fig. 10. Combinations of Unit Instructions

또한 [그림 9]에서 본 데이터 패스 구조에서 하나의 General Purpose Register(GPRs)를 사용함에도 불구하고 Phase#0에서 Phase #1을 수식할 수 있는 처리방식 때문에 Dynamic/nested Branch 명령어를 단일 명령어로 사용하는 방법을 쓸 수 있다는 것을 [그림 11]에서 나타내고 있다.

두 번째 조합의 경우 Phase #0에서 ADDR명령을 통하여 D.w 값을 이용한 Indirect Addressing 연산이 가능함을 보여주며, 네 번째 조합은 'for (i=0; i<100; i++)' 를 예로 들어, Loop연산 과정에서 Count값의 비교와 증가, 이를 이용한 Loop Routine의 수행여부, Loop Routine의 수행 후 반복 수행을 위한 BRC 명령어 등의 조합을 예로 나타내고 있다.

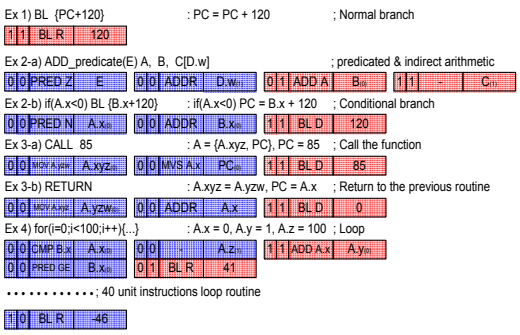


그림 11. 흐름제어 명령어의 조합
Fig. 11. Combinations of data flow control instructions

그래픽 프로세서는 메트릭스 연산을 기본적으로 수행 하고, 데이터의 메트릭스 연산이 연산에서 가장 많은 비용을 차지하고 있다.

3D 그래픽에서 쓰이는 4x4 매트릭스 연산 수행 과정을 보면 {x, y, z, w} 각 컴포넌트당 4번의 곱셈 과정과 세 번의 덧셈 과정이 필요하다. 이러한 과정을 기존의 고정 명령어를 사용할 경우 [그림 12]의 왼쪽과 같이 10개의 명령어가 필요하다. 이것을 제안하는 명령어를 이용할 경우 6개의 명령어로 가능하는 것을 [그림 12]의 오른쪽에서 나타낸다. 명령어를 4개를 줄인 것은 반복된 연산을 계속해서 반복하는 그래픽 연산에서 효율을 높일 수 있다.

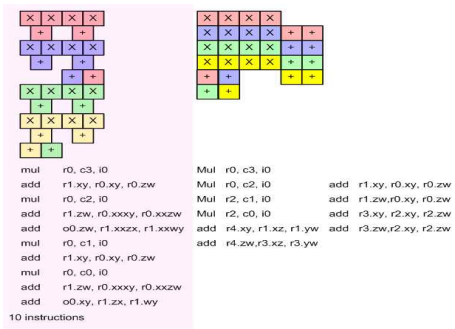
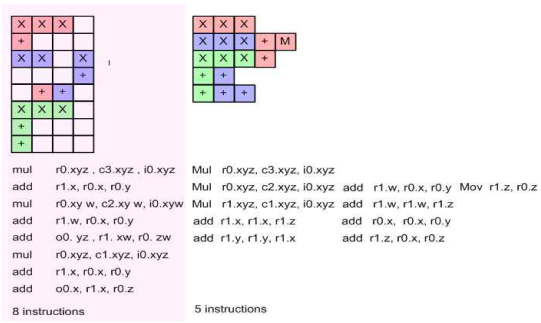


그림 12. 4x4 매트릭스 연산의 명령어 차이
Fig. 12. Difference instructions of 4x4Matrix operation

2D 그래픽에서는 3x3 매트릭스연산 수행을 하는데 4x4연산 보다 각 컴포넌트가 하나씩 적다. 그러나 이 연산의 문제는 3번의 곱셈과정을 거친 후 3개의 값을 더해주는데서 약간의 문제가 있다. Dest <= X+Y+Z 3개의 값을 더할 때 하나 add를 한 후 결과 값을 이용하여 나머지 하나를 더하기 때문에 특히 쉐이더 3.0을 지원하기 위해 긴 명령어가 필요한 상황에서 많은 명령어 필드의 낭비가 있다. 고정명령어를 쓰면 [그림 13]왼쪽 그림과 같이 8개의 명령어를 쓰지만 간단한 단 하나의 Add r1.x, r0.x, r0.y를 위해서 긴 명령어 전체를 만들었음을 볼 수 있는데 이는 비효율적이다. 이것을 제안하는 가변길이 명령어는 [그림 13] 오른쪽 그림에서 보듯이 명령어도 5개로 줄일뿐더러 간단한 명령어는 유닛 하나만 써서도 가능한 구조로 만들었기 때문에 하나의 유닛 명령어만 있어도 되고 필요 없는 공간을 잘 활용이 가능한 구조기 때문에 메모리를 효율적으로 쓸 수 있음을 나타낸다.

그림 13. 3x3 매트릭스 연산의 명령어 차이
Fig. 13. Difference instructions of 3x3Matrix operation



VI. 결 론

제안하는 가변 길이 명령어는 잘 알려진 SIMD와 VLIW(Very Long Instruction Words)의 문제점을 해결하고 수식 가능한 명령어 구조를 통하여 개별적으로는 단순한 구조를 제시하며 전체적으로 진보된 프로세서 구조를 제시한다.

기존의 고정된 명령어를 사용하면 한 명령어에 하나의 연산만을 수행할 수 있고, 또한 Dynamic/nested Branch 수행도 하나 이상의 명령어가 필요하다. 이러한 문제점을 검증에서 보듯이 4개의 유닛 명령어의 조합을 통해 하나의 명령어로 수행할 수 있음을 보여준다. 또한 유닛 명령어들을 최대 4개까지를 어떤한 방식으로 조합하느냐에 따라 많은 수의 다양한 명령어를 만들어 낼 수 있다. 그리고 32비트 유닛 명령어 하나로도 표현할 수 있는 비교적 간단한 명령어는 유닛 명령어 하나만 사용함으로써 고정된 긴 명령어의 필드 낭비의 문제점을 해결할 수 있다.

참 고 문 헌

[1] Microsoft Shader3.0, http://msdn.microsoft.com
[2] H.K. Jeong, "Design of 3D Graphics Geometry Accelerator using the Programmable Vertex Shader" ITC-CSCC 2006.
[3] Mauricio Breternitz, Jr., "Compilation, Architectural Support, and Evaluation of SIMD Graphics Pipeline Programs on a General-Purpose CPU" Proceedings of the 12th international conference on parallel architectures and compilation techniques.
[4] James C. Lelterman, "Learn Vertex and Pixel Shader Programming with DirectX9" Wordware Publishing, Inc. 2004.
[5] Liza Fireman, "The Complexity of SIMD Alignment" Technion -Computer Science Department - M.Sc. Thesis MSC - 2006.

저 자 소 개

이 광 엽 (정회원)

1985년 8월 서강대학교전자공학과 학사.
1987년 8월 연세대학교전자공학과 석사.
1994년 2월 연세대학교전자공학과 박사.
1989~1995년 현대전자 선임연구원.
1995년~현재 서경대학교 컴퓨터공학과 부교수.
<주관심분야 : 마이크로 프로세서, Embedded System, 3D Graphics System>