논문 2011-48SD-7-6

SIMD 프로그래머블 통합 셰이더를 위한 제어 유닛 설계 및 구현

(Control Unit Design and Implementation for SIMD Programmable Unified Shader)

김 경 섭*, 이 윤 섭*, 유 병 철*, 정 진 하**, 최 상 방***

(Kyeong-Seob Kim, Yun-Sub Lee, Byung-Cheol Yu, Jin-Ha Jung, and Sang-Bang Choi)

요 약

그래픽 프로세서의 발달로 실사 수준의 고품질 컴퓨터 그래픽은 여러 분야에 다양한 용도로 사용되고 있으며, 그래픽 프로세서의 핵심 중 하나인 셰이더 프로세서는 프로그램 가능한 통합 셰이더로 발전하였다. 그러나 현재의 상용 그래픽 프로세서들은 특정한 알고리즘에 최적화되어 있어 다양한 알고리즘의 개발을 위해서는 독립적인 셰이더 프로세서가 필요하다. 본 논문에서는 프로그래머블 통합 셰이더 프로세서에서 DirectX 셰이더 어셈블리 명령어를 수행할 수 있는 고성능 3차원 컴퓨터 그래픽 영상을 지원하기 위한 제어 유닛을 설계하고 구현하였다. 설계한 제어 유닛은 기능적 레벨에서 시뮬레이션을 통하여 그 성능을 검증 하였으며, FPGA Virtex-4에 구현하여 하드웨어 리소스 사용율을 확인하고 ASIC 라이브러리를 적용하여 동작속도를 확인 하였다. 또한 비슷한 기능을 하는 셰이더 프로세서에 비해 약 1.5배 정도 많은 수의 명령어를 지원하며, 사용하는 연산 유닛 수에 비해 전체적인 성능은 약 3.1GFLOPS 향상된 결과를 보였다.

Abstract

Real picture like high quality computer graphic is widely used in various fields and shader processor, a key part of a graphic processor, has been advanced to programmable unified shader. However, The existing graphic processors have been optimized to commercial algorithms, so development of an algorithm which is not based on it requires an independent shader processor. In this paper, we have designed and implemented a control unit to support high quality 3 dimensional computer graphic image on programmable integrated shader processor. We have done evaluation through functional level simulation of designed control unit. Hardware resource usage rate are measured by implementing directly on FPGA Virtex-4 and execution speed are verified by applying ASIC library. the result of an evaluation shows that the control unit has the commands more about 1.5 times compared to the other shader processors that is a behavior similar to the control unit and with a number of processing units used in a shader processor, compared with the other processors, overall performance of the control unit is improved about 3.1 GFLOPS.

Keywords: Graphic processor, SIMD, DirectX shader model, HDL, FPGA

Ⅰ. 서 론

3차원 그래픽 데이터 처리 기술은 멀티미디어 환경을 구축하기 위한 가장 핵심적인 연구 분야 일 뿐만 아니라 컴퓨터를 기반으로 하는 많은 응용분야에서 필수적으로 요구된다. 또한 과거에는 특정한 응용분야에서 고가의 컴퓨터 시스템에 적용되었지만, 점차 모바일, PDA 등 차세대 휴대용 정보기기에서도 그 수요가 늘어나고 있다. 이러한 필요에 따라 현실감 있는 3차원 그래픽 영상을 지원하기 위해서는 방대한 양의 데이터를 갖는 복잡한 연산을 효율적으로 처리할 수 있는 고성능 그래픽 프로세서가 요구된다.[1~3] 초기에는 사실적 영상 처리를

* 학생회원, ** 정회원, *** 평생회원,
인하대학교 전자공학과
(Dept. of Electronic Engineering, Inha University)
접수일자: 2011년3월1일, 수정완료일: 2011년6월28일

위한 알고리즘을 지금의 그래픽 프로세서 구조에 맞게 변형하여 구동하는 연구를 하였다. 이러한 흐름에 따라 비교적 간단한 알고리즘은 하드웨어로 구현되고 사실성을 높이기 위한 대부분의 렌더링 작업은 소프트웨어의 지원을 받아왔다. 하지만 그래픽 프로세서의 가속을 받기 위한 부가적인 처리와 컴퓨터의 버스를 통한 데이터 교환에서 오는 비용으로 인하여 적절한 수준의 가속을 받기가 어려웠다. 따라서 그래픽 프로세서는 다양한 효과를 위해 하드웨어의 유연성이 필요하게 되었고, 점차 프로그램 가능한 정점 셰이더와 픽셀 셰이더로 발전하였다. 그러나 단순하고 고정된 기능의 지오메트리(geometry) 연산과 렌더링(rendering) 처리가 아닌 프로그래머블 셰이더를 통한 빠른 그래픽 효과가 요구되면서 정점 셰이더와 픽셀 셰이더는 각각의 파이프라인에 종속되지 않는 통합된 형태로 발전하게 되었다.[4~5] 이와 같은 프로그래머블 통합 셰이더 프로세서는 그래픽 처리 알고리즘 개발에 필요한 셰이더 프로세서의 기능뿐만 아니라 그래픽 프로세서와 분리되어 다른 프로세싱 유닛에 사용될 수 있으며, DSP(Digital Signal Processing) 코어로도 사용될 수 있고, 전역 조명을 비롯하여, 수치 해석, 유체 역학과 같은 잠재적 응용 분야를 갖는다.

본 논문에서는 그래픽 처리의 핵심적인 기능을 담당하는 DirectX의 셰이더 명령을 하드웨어에서 직접 수행할 수 있도록 프로그래머블 통합 셰이더 프로세서를 위한 제어 유닛을 설계하였다.

설계된 제어 유닛은 데이터 레벨 병렬성을 고려하여 필요한 모든 데이터에 대하여 네 개의 단정도 부동소수점(IEEE 754) 입력을 병렬로 처리할 수 있는 SIMD(Single Instruction Multiple Data) 구조이며 모든 명령어는 네 개의 입력 데이터를 처리할 수 있다. 또한 하나의 명령어에 의해 정해진 크기의 연속된 데이터를 처리하며 이를 위해 모든 구성 요소들은 벡터 처리에 적합하게 설계된다. 패치 및 디코더 모듈은 처리하는 벡터의 크기를 관리하고, 이슈 모듈은 하나의 명령어에 대하여 지정된 수의 데이터를 처리하기 위한 카운터를 관리하고 연산 유닛에 지속적으로 데이터를 공급한다. 설계된 제어 유닛은 여러 개의 연산 유닛에서 동시에 서로 다른 연산을 수행 할 수 있는 구조이며 비순차 이슈의 필수 해결 요소인 선, 후행 명령어 사이의 데이터 의존성을 처리하기 위해 스코어보드(scoreboard)[6]의 레지스터 결과 상태 테이블과 유사한 레지스터 상태 테이블을 두어 레지스터의 사용 상태를 관리한다. 이때 디코더 모듈은 레지스터 상태 테이블을 이용하여 해당 명령어에서 사용되는 피연산자들의 사용 가능성을 검사한 후 필요한 정보를 생성하고 이를 이슈 모듈에 전달한다. 여러 개의 연산 유닛에서 수행된 연산 결과는 레지스터 파일을 갱신하는 동시에 모든 이슈 모듈에 브로드캐스팅되어 대기 중인 명령어의 레지스터 파일 참조를 위한 이슈 모듈의 지연은 최소화 된다.

설계된 결과는 기능적 시뮬레이션을 통하여 동작을 검증하였으며 FPGA Virtex-4에 구현하여, 최적화된 하드웨어 리소스의 사용율과 성능을 확인하였다. 또한 여러 가지 프로세서와 비교하여 설계된 기법에 대한 성능을 분석하였다.

본 논문의 구성은 다음과 같다. Ⅱ장에서는 그래픽 프로세서의 특징 및 기존 연구에 대해서 설명하고, 프로그래머블 셰이더의 전체적인 구성과 특징에 대해서 설명한다. Ⅲ장에서는 벡터화된 SIMD 프로그래머블 통합 셰이더 프로세서의 구조를 제시하는 동시에 이 구조를 기반으로 제어 유닛을 제안하고, Ⅳ장에서는 설계한 제어 유닛의 다양한 시뮬레이션 결과를 바탕으로 셰이더 프로세서의 성능을 평가한다. 마지막으로 Ⅴ장에서는 본 논문의 결론 및 향후 연구 과제를 제시하며 끝을 맺는다.

Ⅱ. 관련 연구

본 장에서는 SIMD 프로그래머블 셰이더의 전체적인 구성과 특징에 대해서 설명한다.

1. 그래픽 프로세서의 특징 및 기존 연구

3차원 컴퓨터 그래픽이란 대상으로 하는 공간을 3차원의 기하학적 정보로써 모델링 하고, 이 정보를 사용자의 시점을 기준으로 변환하여 컴퓨터의 평면 디스플레이 장치에 보여주는 것이다. 영상의 품질은 모델링과 렌더링 과정의 우수함에 따라 좌우되며, 실시간성은 렌더링 과정에서 소요되는 시간에 따라 결정된다. 기본적인 렌더링 요소는 일반적으로 점, 선, 삼각형으로 구성된다.[7~8]

렌더링은 데이터를 효율적으로 처리하기 위해 그림 1과 같이 파이프라인(pipeline)형태로 구성되며, 이 파이프라인은 그래픽스 파이프라인(graphics pipeline)이라고 불린다. 즉, 데이터 처리를 독립적으로 할 수 있도록

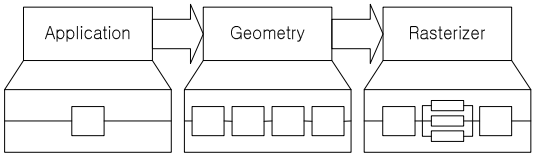

그림 1. 렌더링 파이프라인의 기본 구조
Fig. 1. Basic structure of rendering pipeline.

응용, 기하, 레스터화(rasterization)의 세 단계로 모듈화 되어 데이터가 병렬처리 된다.[9]

Nvidia의 Geforce 8 시리즈와 ATI의 HD2000 시리즈부터는 스트림 프로세서가 도입되어, 기존의 픽셀 셰이더와 정점 셰이더 유닛, 기하 유닛을 대신해 데이터를 처리하는 통합 셰이더 유닛의 성격을 가지게 되었고 하드웨어 구조가 단순화 되었다. 통합 셰이더 유닛이 개발되기 이전에는 프로그래머가 한정된 자원 내에서 프로그래밍을 해야 하는 불편함이 있었다. 그러나 정해진 통합 셰이더 유닛에서 픽셀, 정점, 기하 셰이더의 수를 선택적으로 활용할 수 있게 됨에 따라, 특정 그래픽 처리 상황에 주요한 셰이더를 더 많이 사용하는 유연한 프로그래밍이 가능해지게 되었다.[10~11]

일반적으로 멀티미디어 데이터 처리는 시간이 많이 걸리고, 연산 집약적인 특성을 가지고 있다. 또한 영화나 비디오게임 등 다양한 분야에서 사용되는 3차원 영상 역시 많은 수의 폴리곤 계산과 기하 연산, 렌더링 등을 위해 많은 시간과 복잡한 과정을 필요로 한다. 이와 같은 멀티미디어 데이터와 3차원 영상에 사용되는 데이터는 많은 정보를 한 번에 연산함으로써 효율적으로 처리할 수 있다. 멀티미디어 데이터는 텍스트, 오디오, 이미지, 그래픽, 비디오 등의 데이터를 반복적으로 처리하는 과정을 거치므로 이를 병렬로 수행함으로써 효율적인 데이터 처리를 기대할 수 있다. 3차원 영상을 구성하는 각 픽셀은 (x, y, z, w)로 구성되는 위치정보와 (r, g, b, a)로 구성되는 색채 정보로 표현되므로, 각각의 데이터에 대해 네 가지 정보를 병렬로 수행하여 처리 속도를 향상시킬 수 있다. 특히 3차원 영상의 특성상 한 사물의 움직임을 표현하는 과정이나, 개별 픽셀의 위치를 동일한 패턴으로 이동시키는 연산이 자주 이루어지기 때문에 여러 픽셀 데이터에 대해서 동일한 연산을 반복적으로 수행하는 경우가 대부분이다. 이렇게 독립적이고 반복적인 연산 특성을 지닌 데이터들은 벡터 프로세서를 사용하여 효율적으로 처리할 수 있다. 벡터 프로세서는 하나의 명령어에 의해 하나의 데이터를 처리하는 스칼라 프로세서(scalar processor)와 달리 하나의 명령에 의해 복수의 데이터를 연속으로 처리함으로써 연산속도를 높일 수 있는 장점을 가지고 있다. 특히 하나의 명령어로 여러 개의 값을 동시에 계산하는 SIMD는 벡터 프로세서에서 많이 사용하는 방식으로 멀티미디어 분야에서 자주 사용되고 있다.

2. SIMD 프로그래머블 셰이더 프로세서 구조

프로그래머블 통합 셰이더 프로세서는 파이프라인과 병렬처리의 장점을 모은 것으로, SIMD 및 벡터 프로세서 구조를 적용하여 설계하였다.

SIMD 구조는 파이프라인을 사용하는 일반적인 프로세서와 동일하게 여러 개의 명령어를 시간적으로 중첩시켜 실행하고, 동시에 독립적인 여러 개의 파이프라인 유닛을 사용하여 명령어를 병렬로 수행하는 구조이다. 일반적으로 그래픽 프로시저는 개개의 픽셀에 대해 동일한 연산을 수행하기 때문에 프로그래머블 통합 셰이더는 많은 데이터에 대해 하나의 명령어로 동일한 연산을 연속적으로 수행한다. 이러한 특성을 이용하여 벡터 프로세서 구조는 연산 파이프라인의 효율적인 사용을 추구할 수 있는 방법으로 동일한 연산을 연속된 데이터에 대하여 처리한다. 스칼라 프로세서의 피연산자는 단일 값이지만 벡터 프로세서의 피연산자는 연속된 데이

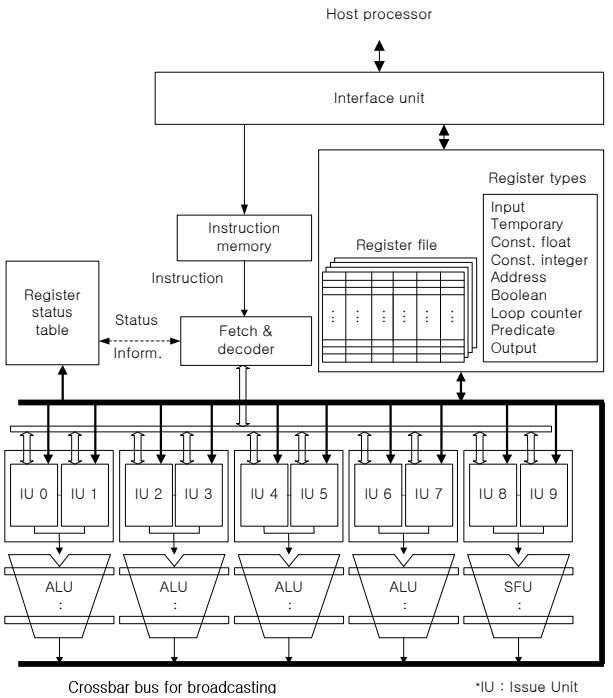

그림 2. 제안된 프로그래머블 통합 셰이더 프로세서
Fig. 2. Proposed programmable unified shader processor.

터이다. 그러나 하드웨어의 한계 때문에 벡터의 길이를 무한정 길게 할 수 없으며, 본 논문에서는 서로 연관성 있는 데이터의 묶음(예를 들어 v0~v15, r0~r31, ...)을 하나의 번들(bundle)이라 하고, 다시 이를 벡터 처리를 위한 개수만큼 묶어 벡터(vector)라고 하며, 그 개수를 벡터 크기라 한다. 설계된 프로그래머블 통합 셰이더 프로세서의 구조는 그림 2와 같다.

인터페이스 유닛은 상위 호스트 프로세서와 셰이더 프로세서 사이의 데이터 및 제어 신호의 전달을 담당하고, 명령어 메모리는 호스트 프로세서로부터 인터페이스 유닛을 통하여 전달되는 명령어를 가지고 있으며, 인터페이스 유닛의 쓰기 동작과 패치 및 디코더 유닛의 읽기 동작을 위해 이중 포트 램을 이용하여 구성되었다. 패치 및 디코더 모듈 중 패치 모듈은 명령어 메모리의 내용을 읽어서 이를 디코더 모듈에 전달한다. 하나의 명령어를 읽어 이를 디코더 모듈에 전달한 후에는 명령어 포인터를 하나 증가시켜 다음 명령어를 읽을 준비를 한다. 다음 사이클에 디코더 모듈이 명령어를 받을 준비가 되면 같은 동작을 반복하며 그렇지 않으면 디코더 모듈이 준비될 때까지 기다렸다가 다음 명령으로 진행하는 동작을 반복한다. 디코더 모듈은 패치 모듈로부터 받은 명령어를 분석하여 필요한 제어 신호를 만들게 되며, 이 제어 신호는 비어있는 이슈 유닛에 전달된다. 레지스터 상태 테이블은 디코더에 의해 특정 레지스터가 목적지 레지스터로 사용됨을 입력받아 레지스터가 사용 중임을 테이블에 업데이트하고, 연산 유닛에서 해당 레지스터의 쓰기 동작이 일어날 때 레지스터의 사용이 끝났음을 테이블에 업데이트하며, 이 정보를 디코더 모듈에 제공한다. 이슈 모듈은 전달받은 명령어의 소스 레지스터들이 모두 사용 가능하고 자신이 연산 유닛으로 데이터를 전달할 수 있는 상태가 되면 레지스터 파일 또는 연산 유닛과 연결된 데이터 버스에서 소스 레지스터의 값을 읽어 이를 제어 정보와 함께 연산 유닛에 전달하여 명령어가 수행되도록 한다. 연산 유닛은 ALU(Arithmetic Logic Unit)와 SFU(Special Function Unit)로 나누고, 4개의 IEEE 754 입력에 대하여 병렬로 필요한 연산을 수행할 수 있는 구조로 되어 있으며, 이슈 유닛으로부터 필요한 컨트롤 정보와 데이터를 받아 연산을 수행한 후 그 결과를 목적지 레지스터에 쓰기 위한 신호를 생성한다. 마지막으로 레지스터 파일 구조는 필요한 레지스터들에 대해서 번들의 개수만큼 다중화 되어 벡터 연산을 수행할 수 있도록 하였고, 입출력 레지스터의 경우 다시 이중 버퍼 구조로 하여 명령어를 수행함과 동시에 호스트에서는 처리가 완료된 결과를 읽어가고 다음에 처리할 데이터를 준비할 수 있도록 하였다.

Ⅲ. 통합 셰이더 프로세서를 위한 제어 유닛 설계

본 장에서는 설계한 프로그래머블 통합 셰이더 프로세서의 명령어 구조, 패치 및 디코더 모듈, 레지스터 상태 테이블, 이슈 모듈에 대해서 설명한다.

1. DirectX 셰이더 어셈블리 명령어

마이크로소프트사의 DirectX 셰이더 어셈블리 언어 v3.0에서 지정된 셰이더 명령의 일반적인 형태는 아래와 같으며, []안에 있는 항목은 선택적으로 존재하는 항목이고 그 세부적인 내용은 아래와 같다.[15]

[Predicate] Instruction [Destination][Source0][Source1][Source2]

명령어 앞에 [Predicate]이 지정된 경우는 연산의 결과를 목적지 레지스터(destination register)에 쓸 때 프레디컷 레지스터(predicate register)의 값을 참조하여 그 값이 1인 요소에 대해서만 쓰기를 수행하도록 지정하는 것이다. 예를 들어 p0의 값이 (1, 1, 0, 0)이라면 명령어 수행 후 그 결과의 x, y, z, w 항목 중 x와 y만이 최종적으로 목적지 레지스터에 반영되는 것이다.

Instruction 항목은 반드시 존재하는 항목으로 명령어들은 크게 네 가지로 분류될 수 있다. 그 첫 번째는 입력 레지스터를 지정하는 명령어, 텍스쳐의 속성을 지정하는 명령어, 상수를 지정하는 명령어와 같은 셋업 명령어로 프로세서에 의해 어떠한 연산이 일어나지 않는 명령어이다. 두 번째 명령어 유형은 연산 명령어로 입력된 값을 이용하여 수학적 연산의 결과를 목적지 레지스터에 저장하는 기능을 한다. 세 번째 명령어 유형은 텍스쳐 명령어로 3차원 그래픽 처리의 데이터베이스를 참조하는 것과 같은 기능을 하는 명령어로 독립적인 텍스쳐 유닛을 필요로 한다. 마지막 명령어 유형은 흐름 제어 명령어로 루프, 반복, 분기와 같은 프로그램의 흐름을 제어하는 명령어이다.

[Destination]은 연산의 결과가 저장되는 목적지 레지스터이며, 그 일반적인 형식은 register[.write mask]로 표현된다. 쓰기 마스크를 고려한 일반적인 형태는 register.[x][y][z][w]와 같이 표시될 수 있으며, 연산의 결과를 적용하고 싶은 항목을 개별로 지정하도록 되어

있다. 셰이더 명령어의 쓰기 마스크 기능을 제공하기 위하여 프로그래머블 통합 셰이더 프로세서에서는 연산이 수행되는 동안 데이터의 흐름을 따라 같이 이동하여 최종적으로 레지스터 파일에 전달되어 연산의 결과가 쓰일 때 반영되도록 하였다. 여기서 [Source0] [Source1] [Source2]는 연산의 입력으로 사용되는 소스 레지스터이며 명령의 종류에 따라 그 수가 다르다.

2. 명령어의 형식

DirectX의 셰이더 어셈블리 언어를 이용하거나 HLSL(High Level Shader Language)을 이용하여 프로그램을 작성한 경우, 이는 DirectX에서 실행하기 위해 어셈블 또는 컴파일 과정을 거쳐 실행 가능한 이진 코드를 생성해야 한다. 프로세서의 이진 코드를 생성하는 데 있어 DirectX에 있는 이진 코드를 사용하는 것도 하나의 방법이 될 수 있으나, 이는 실질적으로 프로세서에서 필요한 이진 코드에 비해 길기 때문에 하드웨어적인 최적화가 필요한 프로세서 구조에는 적합하지 않다. 따라서 본 논문에서는 이진 코드를 DirectX 의 정점 셰이더와 픽셀 셰이더 명령어를 참조하여 제안하는 마이크로 아키텍처에 맞도록 단순화시켜 구성하였다.

표 1. 명령어의 이진 형식
Table 1. Binary format of instruction.

op code(20bit)	dest (14bit)	src0 (22bit)	src1 (22bit)	src2 (22bit)
101 - 81	80 - 66	65 - 44	43 - 22	21 - 0

제안된 셰이더 전체 명령어의 이진 형식은 표 1과 같이 구성된다. 동작 코드(operation code) 항목은 20 비트를 차지하며 수행될 연산에 관계된 정보를 가진다. dest은 14 비트를 가지고 연산이 수행된 후 최종 결과를 저장할 레지스터 및 관련 정보를 가지게 된다. src0, src1, src2는 각각 22 비트를 가지고 연산에 사용될 정보를 갖는다.

3. 제어 유닛의 구조

본 절에서는 패치 및 디코더 모듈, 레지스터 상태 테이블, 이슈 모듈로 구성된 제어 유닛을 설명한다.

가. 패치 및 디코더 모듈

설계된 셰이더 프로세서는 패치 모듈과 디코더 모듈이 통합된 형태로 구성은 그림 3과 같다. 패치 모듈과 디코더 모듈은 밀접한 관계를 가지고 있어 하나의 파일로 작성하는 것이 용이하다.

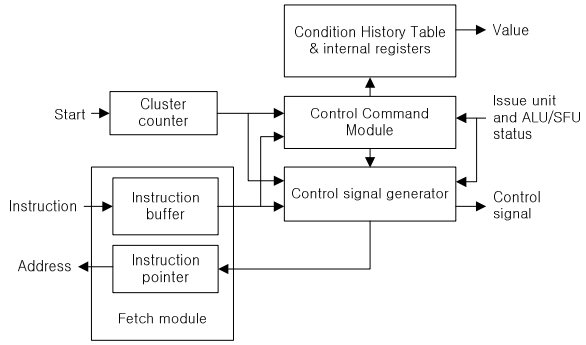

그림 3. 패치 및 디코더 유닛의 블록 다이어그램
Fig. 3. Block diagram of fetch and decoder unit.

(1) 패치 모듈

명령어 패치 모듈은 명령어 메모리부터 명령어를 읽어 디코더 모듈로 전달하고 다음 수행할 명령어를 가리키도록 명령어 포인터(instruction pointer)를 증가시켜 명령어 메모리에 전달해주는 역할을 한다. 즉, 패치 모듈은 레지스터 파일로부터 디지털 회로의 기본적인 입력 신호인 클럭 입력을 받고 인터페이스 유닛으로부터는 리셋(reset), 동작 시작 신호를 받으며, 명령어 메모리에 읽고자하는 명령어의 주소를 내보내어 명령어 메모리로부터 이진 명령어 코드를 받아, 이를 디코더 모듈로 전달한다. 패치 모듈의 동작은 그림 4와 같다.

일반적인 벡터 동작이나 순차적인 스칼라 동작에서 패치 모듈은 비어있는 이슈 유닛이 있어 디코더 모듈에서 명령어를 처리하여 다음 명령어로 진행할 수 있는 상황이면 클럭이 인가됨에 따라 명령어 포인터를 증가시켜 명령어 메모리에 다음 어드레스에 해당하는 명령

```
if (Start signal on)
   Read one instruction
   Transfer one instruction to decoder module
   while (Issue unit is not ready)
      wait
   end
   Increase instruction pointer
   Transfer instruction pointer to instruction memory
else
   wait
end
```

그림 4. 패치 모듈의 동작
Fig. 4. Operation of fetch module.

어를 요청한다. 그러나 진행할 수 없는 상황이면 현재의 명령어 포인트를 유지하여 처리되지 않은 명령어가 대기할 수 있도록 한다.

(2) 디코더 모듈

디코더 모듈은 패치 모듈에서 전달받은 명령어 코드, 목적지 레지스터, 소스 레지스터 0, 1, 2의 정보에 대해 그 내용을 분석하여 연산 유닛의 타입을 결정하고 결정된 연산 유닛의 이슈 모듈 상태와 사용되는 레지스터 상태를 점검하여 명령어의 전달이 가능하면 이슈 모듈과 연산 유닛에서 필요한 제어 신호를 생성하여 그 정보를 이슈 모듈로 보내 해당하는 연산을 수행하도록 한다.

(가) 프로그램 반복 횟수 결정

3차원 데이터의 처리를 위해서는 많은 벡터를 처리하여야 하고 이를 위해서는 프로세서가 프로그램의 끝까지 수행한 후에는 다시 새로운 벡터에 대하여 처음부터 명령어를 다시 수행해야 한다. 이러한 프로그램의 반복 횟수는 외부에서 디코더 모듈로 입력되며 디코더는 처음 프로그램을 시작할 때 이 값을 카운터에 저장한 후 프로그램의 마지막에 도착하면 저장된 값을 하나씩 감소시켜 카운터가 0이 될 때까지 프로그램의 수행을 반복함으로써 필요한 모든 데이터에 대한 처리를 진행한다.

(나) 디코딩

디코더 모듈의 주된 기능 중 하나는 입력 레지스터들에 대한 의존성을 검사하는 것이다. 즉, 입력으로 사용된 레지스터가 이전의 명령에 의해 사용되고 있는 목적지 레지스터이면 RAW 의존성(read after write dependency)이 있는 것으로, 이슈 모듈은 해당하는 레지스터의 값이 생성된 후에 사용해야 한다. 따라서 이러한 경우 디코더는 의존성에 대한 정보 신호를 생성하여 이슈 모듈에 전달한다. 쓰기 의존성이 없거나 해결된 연산 명령어에 대해 디코더 모듈은 이슈 유닛을 결정하여 필요한 정보를 전달함으로써 연산을 수행할 준비를 하도록 한다. 디코더 모듈의 이슈 유닛 할당에 대한 알고리즘은 그림 5와 같다.

디코더 모듈은 명령어에 의해 결정된 연산 유닛의 종류에 따라 해당 이슈 유닛의 상태를 검사한다. 하나의 연산 유닛에는 두 개의 이슈 모듈이 있으므로 두 개의

```
if (No WAW dependency)
   if (Find (Issue unit with two empty issue module))
      Send control signal
   else
      if (Find (Issue unit with two empty issue module))
         Select issue unit that has not received last instrction
         Send control signal
      else
         wait
      end
   end
else
   wait
end
```

그림 5. 디코딩 모듈의 이슈 모듈 선택 알고리즘
Fig. 5. Issue module selection algorithm of decoding module.

이슈 모듈이 모두 비어있는 곳에 명령어를 보내는 것이 전체적인 실행시간을 줄일 수 있는 방법이다. 따라서 디코더 모듈은 두 개의 이슈 모듈이 모두 비어있는 곳이 있는가를 먼저 검사하고, 그렇지 않으면 하나의 이슈 모듈이 비어있는가를 검사하며 모두 사용 중이면 대기하게 된다. 하나의 연산 유닛에 해당하는 두 이슈 모듈들의 사용 상태가 같은 조건일 경우는 마지막으로 명령어를 받은 이슈 유닛을 제외한 다른 이슈 유닛을 우선적으로 선택한다. 디코더 모듈은 현재의 명령을 받아들일 수 있는 이슈 모듈에 관한 플래그를 설정하고 이슈 유닛이 결정될 때마다 해당 플래그를 반전하는 방법으로 명령어의 전달이 가능한 이슈 모듈에 관한 정보를 유지한다. 이슈 유닛이 결정되면 디코더는 해당 이슈 유닛에 제어 신호를 생성하여 디코더의 정보를 수신하게 한다.

(다) 제어 명령어

셰이더 프로세서에서 처리하는 많은 그래픽 프로시저는 분기나 반복 없이 일정한 연산을 순차적으로 처리하는 경우가 많다. 하지만 필요에 따라서는 입력 데이터의 조건에 따른 프로그램의 흐름 제어가 필요한 경우도 있다. 이러한 경우의 데이터 처리를 위하여 DirectX에서는 조건문 및 반복문을 지원하고 있다. 셰이더 어셈블리에서 지원하는 제어 명령어는 `LOOP`, `REP`, `CALL`, `BREAK`, `IF` 명령어 등이 있으며 개별적인 제어 명령의 수행 뿐 아니라 중첩된 제어 명령에 대한 처리도 가능하다. 프로그램의 수행 중 변하지 않는 조건인 고정 조건 제어문은 일반적인 명령어와 마찬가지로 벡터 방식의 처리가 가능하다. 반면 변동 조건은 번들의 입력 또는 계산의 중간 값을 이용하여 생성되는 조건으

로 번들마다 다른 영향을 끼칠 수 있으며 상황에 따라 벡터 처리가 가능할 수도 있고 그렇지 않을 수도 있다.

벡터 처리가 불가능한 제어문의 처리를 위하여 본 논문에서는 세이더 프로세서를 기본적인 벡터 처리 이외에 스칼라 모드로 동작할 수 있도록 설계하였고, 스칼라 처리를 위하여 스칼라 모드의 시작과 끝을 알리는 명령어를 추가하였다. 세이더 프로세서가 스칼라 모드 시작 명령을 만나면 번들의 번호를 하나로 고정시킨 후 지정된 구간의 명령어를 연속적으로 실행하고, 스칼라 모드 끝 명령어를 만나게 되는 경우, 다음에 처리할 번들의 번호를 확인하여 벡터 내의 모든 번들에 대한 처리가 끝났으면 다음 명령어로 진행하고, 그렇지 않으면 다시 스칼라 모드 시작 명령 다음부터 명령어들을 수행하는 동작을 반복한다.

(라) 조합 연산 명령어

DirectX 명령어들 중에서 M2X4, M2X3 와 같은 명령어들은 기본 명령어들의 조합으로 이루어진 명령어이다. 이와 같은 명령어들을 모두 연산 유닛에서 하나의 명령어로 처리하는 것은 파이프라인의 깊이를 깊게 하여 처리 효율을 떨어드리고 하드웨어의 사용률을 높이는 결과를 초래하기 때문에 매우 비효율적이다. 하지만 이러한 명령어들은 기본 명령어를 여러 번 수행하여 결과를 얻을 수 있으므로 이러한 조합 연산 명령어들을 위한 구성을 따로 하지 않았고 어셈블 과정에서 기본 명령어로 분리하여 처리하는 방법을 택하였다. 그러나 이렇게 기본 명령어를 이용하여 조합 명령어를 수행하기 위해서는 계산의 중간 과정을 저장할 별도의 버퍼가 필요하게 된다. 따라서 조합 명령어를 수행하기 위해서 임시 레지스터와 같은 기능을 하는 별도의 레지스터를 추가하여 임시 레지스터를 확장하고 조합 명령어의 처리는 이 레지스터를 사용하도록 하였다. 확장된 레지스터는 임시 레지스터와 동일한 구조를 가지며 디코더 모듈 및 레지스터 상태 테이블 외의 다른 유닛에서도 임시 레지스터와 동일하게 취급된다.

나. 레지스터 상태 테이블

설계된 세이더 프로세서에 적용된 비순차 이슈 방식은 동시에 여러 개의 명령어를 수행하기 때문에 명령어 수행 시 오퍼랜드를 참조하는 순서가 명령어를 순차적으로 하나씩 수행할 때의 순서와 다르게 나타날 수 있다. 즉, 명령어 사이에 존재하는 데이터간의 의존성

```
if (Register set from decoder module)
   Update status table entry
else
   if (Broadcasting listen of busy register entry)
      Clear status table entry
   end
end

Send status table information to decoder unit
```

그림 6. 레지스터 상태 테이블의 동작
Fig. 6. Operation of register status table.

(dependency)에 의해 비정상적인 결과가 발생할 수 있다. 이러한 의존성을 해결하기 위해 본 논문에서는 레지스터의 사용 가능성을 검사 할 수 있도록 레지스터들의 상태를 표시하고 관리하는 레지스터 상태 테이블을 구성하였다.

그림 6은 레지스터 상태 테이블의 동작에 대한 의사코드이다. 레지스터 상태 테이블은 소스 및 목적지 레지스터로 사용될 수 있는 레지스터들이 어느 이슈 모듈에서 사용되는지의 정보를 디코더 모듈에서 전송받아 테이블에 기록해 둠으로써 현재 사용되는 레지스터 상태정보를 유지하고 ALU와 SFU에서 레지스터에 쓰기 동작(브로드캐스팅(broadcasting))이 일어나면 이를 검사하며 해당하는 레지스터의 상태를 비사용 상태로 업데이트한다. 이렇게 유지되는 레지스터 상태정보는 디코더 모듈로 보내져 레지스터 사이의 의존성을 해결할 수 있도록 한다.

다. 이슈 모듈

이슈 모듈의 기능은 디코더로부터 전달받은 정보를 이용하여 적절한 시점에 연산 유닛으로 제어 코드 및 연산에 사용될 데이터를 전달하는 것이다. 이를 위해 두 이슈 모듈 중 연산 유닛으로 데이터 전달이 가능한 모듈을 선택하는 이슈 모듈 선택기 및 이슈 지연 카운터가 필요하다. 이슈 지연 카운터는 명령어가 파이프라인으로 전달되는 위치가 달라서 발생하는 선, 후행 명령어 사이의 충돌을 방지하기 위해 선행 명령어와 후행 명령어의 상관관계를 고려하여 명령어를 연산 유닛으로 이슈 가능한 시점을 결정한다. 슈퍼 스칼라 프로세서는 이슈 윈도우를 사용하여 여러 개의 명령어 중 적절한 명령어를 먼저 실행하는 비순차 이슈를 통하여 순차 이슈에 비해 성능을 향상시킨다. 그러나 이러한 이슈 윈도우를 사용한 비순차 이슈는 제어 로직이 복잡하여 FPGA에 구현하기에는 적절하지 않다.

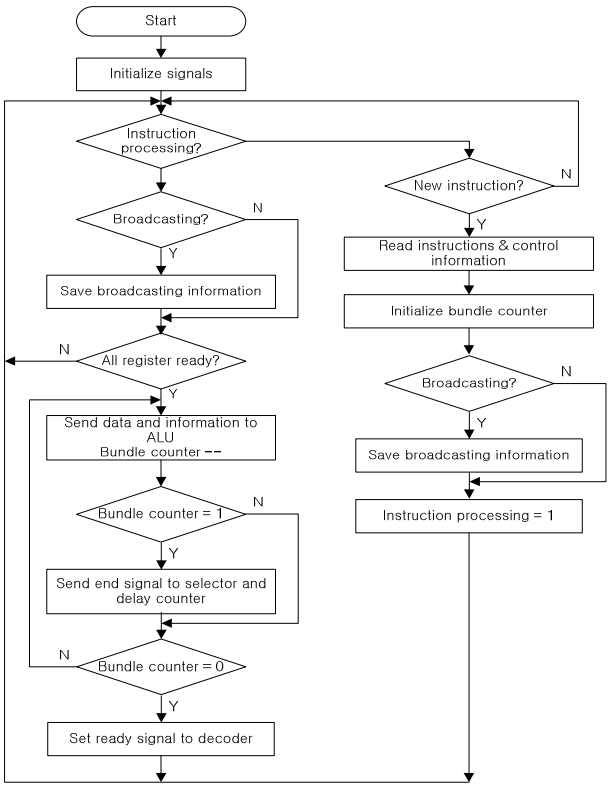

그림 7. 이슈 모듈의 순서도
Fig. 7. Flowchart of issue module.

따라서 본 논문에서는 연산 유닛의 앞단에 두 개의 이슈 모듈을 구성하여 비교적 복잡도가 낮으면서 비순차 이슈가 가능한 구조를 선택하였다. 설계된 이슈 모듈의 순서도는 그림 7과 같다.

셰이더 프로세서는 하나의 명령에 대하여 벡터 크기만큼 번들의 번호를 증가시켜 벡터 처리를 한다. 하나의 레지스터는 벡터 크기에 해당하는 배열로 구성되어 있고, 고유한 레지스터의 주소는 레지스터 오프셋과 번들 번호로 결정이 된다. 이슈 모듈이 소스 레지스터의 값을 준비하기 위하여 연산 유닛의 쓰기를 감시하는 경우, 이미 해당 연산 유닛에 의해 쓰기가 진행되고 있는 중일 수가 있다. 즉, 해당 연산 유닛에서 필요한 레지스터 오프셋에 대한 결과는 나오고 있지만 첫 번째 번들은 지나간 상태인 것으로 이때는 RAW(Read After Write) 해저드가 해결되어 필요한 내용은 레지스터 파일에 들어있고 레지스터 파일에서 값을 읽어 사용할 수 있다. 소스 레지스터의 의존성 해결을 검사하는 데 있어 이슈 모듈은 필요한 소스 레지스터의 번들 번호, 오프셋 정보를 참조하여 소스 레지스터에서 실제로 사용되는 요소에 대해서만 사용 가능 여부를 판단한다. 이는 다른 유닛에서 하나의 소스 레지스터 내의 네 개의 요소에 대하여 마치 개별 레지스터로 간주하여 처리하는 것과 같다.

모든 소스 레지스터의 입력 값이 준비되면 이슈 모듈은 컨트롤 정보와 함께 그 값을 연산 유닛으로 전달하고 해당 이슈 모듈이 동작 중임을 알리는 신호를 내보내며 동작이 완료되면 동작 신호의 출력을 중지한다. 이 신호는 디코더 모듈, 이슈 지연 카운터, 이슈 선택기에 전달되어 사용된다. 이슈 모듈이 연산 유닛에 필요한 정보를 전달함에 있어 번들 번호를 관리하는 것은 큰 비중을 차지한다. 셰이더 프로세서가 벡터 모드로 동작하는 경우 이슈 모듈은 번들 번호를 0부터 차례로 증가시키며 벡터 크기만큼 연산 유닛에 전달하고 레지스터 파일에서 값을 읽어올 때도 현재의 번들 번호에 해당하는 값을 읽어온다. 그러나 동작 모드가 스칼라 모드이면 번들 번호에 대한 관리는 디코더 모듈에서 이루어지고 이슈 모듈은 디코더 모듈으로부터 전달받은 번들 번호를 참조하여 동작을 수행하게 된다.

Ⅳ. 합성 결과 및 성능 평가

본 논문에서 구현된 모든 구조는 HDL(Hardware Description Language) 언어를 사용하여 기술하였으며, Xilinx사의 ISE 9.2 프로그램을 사용하여 합성한 후, Mentor Graphics사의 ModelSim 6.0 SE를 이용하여 동작 검증을 하였다.

1. 합성 결과

본 논문에서 구현한 제어 유닛은 Xilinx 사의 FPGA 중 Virtex4 계열의 XC4VLX200을 사용하였고 입력 클럭 90MHz에 동작한다. 다른 유닛들에 비해 상대적으로 복잡하고 유기적인 구조를 갖는 제어 유닛은 하나의 동작 레벨 코딩을 통하여 구현하고 이를 시뮬레이션을 통하여 검증하였다. 또한 ASIC으로 구현하였을 때의 성능을 평가하기 위하여 제어 유닛을 구성하는 여러 모듈 중 크리티컬 패스(critical path)를 가지고 있는 이슈 유닛에 대하여 SMIC사의 0.18μm ASIC 라이브러리를 적용하여 배치 및 배선을 하였다. 그 결과 제안된 셰이더 프로세서의 동작 속도는 227MHz이며 벡터 크기 128에서 3.7GFLOPS(GPU FLoating point Operations Per Second)의 성능을 보인다. 표 2는 연산 유닛의 설계 구조에 따른 모듈별 슬라이스 사용량을 나타낸 것이다. 패치 및 디코더 모듈은 비교적 구조가 단순하고 하드웨

표 2. 모듈별 슬라이스 사용량
Table 2. Slice usage of each module.

모듈	슬라이스(Slices) 사용량					
	2ALU 1SFU	3ALU 1SFU	3ALU 2SFU	4ALU 1SFU	4ALU 2SFU	5ALU 1SFU
패치 및 디코더	3048	3069	3108	3241	3129	3210
레지스터 상태 테이블	8743	8743	8743	8743	8743	8743
이슈	20979	27972	34965	34968	41958	41958

어 사용량이 많지 않으며, 레지스터 상태 테이블은 모든 임시 레지스터의 사용 상태를 보관하여 이 정보를 패치 및 디코더 모듈에 제공해야 하기 때문에 데이터의 양이 크지만 블록 램으로 구성할 수 없어 많은 하드웨어를 차지하게 된다. 이슈 모듈은 브로드캐스팅의 검사나 레지스터의 가용성 검사 등을 수행하는 로직이 복잡하여 비교적 많은 하드웨어를 사용한다.

2. 성능 평가

본 논문에서 구현한 제어 유닛은 성능 평가를 위하여 2개의 ALU와 1개의 SFU로 구성된 2ALU/1SFU 구조와 3ALU/1SFU, 3ALU/2SFU, 4ALU/1SFU, 4ALU/2SFU, 5ALU/1SFU의 6가지 구조에 대하여 성능을 측정하였다. 또한 각 구조에 대해 벡터 크기를 16, 32, 64, 128로 하여 성능을 측정하였으며, 입력 픽셀의 수는 320, 1600, 3200, 16000, 32000에 대하여 성능의 변화를 측정하였다. 세이더 프로세서의 성능 평가의 지표가 되는 가장 핵심적인 항목은 IPC(Instruction Per Clock)이다. IPC는 프로시저를 수행하며 실행한 명령어의 수를 소요된 클럭의 수로 나누어 얻을 수 있다. 설계한 세이더 프로세서는 벡터 방식의 처리를 하므로 전체 실행된 명령어의 수는 프로시저를 실행하며 수행된 명령어의 수에 처리된 픽셀의 수를 곱하였고, 프로시저 실행에 소요된 클럭은 첫 번째 명령어를 시작하여 최종적인 결과의 쓰기가 끝나는 시점까지의 클럭을 사용하였다.

그림 8은 세이더 프로세서의 구조별 로직에 대한 효율을 나타낸 것이다. 연산 유닛의 증가에 따라 효율 또한 증가를 하게 되는데 이는 패치 및 디코더 유닛, 이슈 유닛, 레지스터 상태 테이블 등의 제어 로직이 차지하는 하드웨어의 사용률에 비해 연산 유닛이 차지하는 하드웨어 사용량의 비중이 낮으며 연산 유닛이 증가함에

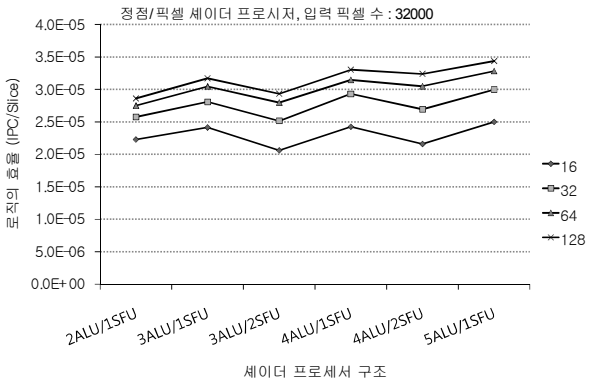

그림 8. 세이더 프로세서 구조별 로직의 효율
Fig. 8. Shader processor structure versus efficiency of logic.

따라 비례적으로 증가하는 하드웨어의 양이 많지 않기 때문이다. 연산 유닛의 증가에 따라 증가하는 하드웨어 부분은 이슈 및 디코더 유닛의 연산 유닛 할당 로직과 이슈 모듈의 브로드캐스팅 검사 로직, 레지스터 상태 테이블의 브로드캐스팅 검사 로직 및 연산 유닛의 출력 버스 등이다. 또한 벡터 크기가 증가함에 따라 하드웨어의 효율이 증가하는데 이는 제어 로직 및 연산 유닛들은 벡터 크기가 변함에 따라 하드웨어의 사용량이 거의 많이 변하지 않기 때문이다.

표 3은 여러 프로세서의 성능에 대한 비교표이다. 상용 제품으로 출시되어 있는 대부분의 프로세서들은 좋은 제조 공정을 사용하기 때문에 최소 300MHz에서 최대 3.1GHz의 높은 주파수로 동작하며 이에 따라 높은 연산 성능을 나타낸다.

각 프로세서들은 그 용도 및 특성이 다르기 때문에

표 3. 여러 프로세서의 성능
Table 3. Performance of various processors.

프로세서	동작 주파수 (MHz)	성능 (GFLOPS)	트랜지스터 수 (million)	기타
TMS320C674x	300	2.4	3	6 ALU, 2 MUL
Intel Pentium 4(SSE)	3080	12	42	2 ALU
AMD (3DNOW!)	2500	10	21	2 ALU
Radeon X1950 XT	625	30	384	8 VS, 48 PS
Radeon HD4870	1050	1200	830	800 PU
Imagine	150	6.1	NA	6 ALU * 8 Cluster
ATTILA	X	X	NA	시뮬레이션 모델
통합 세이더 프로세서	**277**	**3.7**	**1.6**	**4 ALU, 2 SFU**

표 4. 여러 프로세서의 특징
Table 4. Feature of various processors.

프로세서	명령어 종류	Vector/ Scalar	SIMD	VLIW
TMS320C674x	240	scalar	O	O
Intel Pentium 4(SSE)	195	scalar	O	X
AMD(3DNOW!)	27	simple vector/ scalar	O	X
Radeon X1950 XT	44	scalar	O	X
Radeon HD4870	180	scalar	O	X
Imagine	62	scalar	O	O
ATTILA	40	vector/ scalar	O	X
통합 셰이더 프로세서	**70**	**vector/ scalar**	**O**	**X**

직접적인 성능의 비교가 어렵지만 Radeon X1950 XT에 사용되는 셰이더 프로세서의 프로세서 당 성능은 0.53GFLOPS 이며, Radeon HD4870에 사용된 셰이더 프로세서의 프로세서 당 성능은 1.4GFLOPS이다.[12~13] 이러한 결과를 보았을 때 셰이더 프로세서를 지원하기 위해 본 논문에서 구현한 제어 유닛은 비슷한 기능을 하는 셰이더 프로세서에 비해 약 1.5배 정도 많은 수의 명령어를 지원하며 사용하는 연산 유닛의 수에 비해 전체적인 성능은 약 3.1GFLOPS 만큼 성능이 향상되었다.

표 4는 여러 프로세서들의 특징을 비교한 표이다. 일반적으로 TMS320C674x, Intel Pentium 4(SSE) 등과 같은 범용 프로세서들은 여러 가지 용도로 사용되기 때문에 다양한 명령어를 지원한다.[14] 그러나 설계된 셰이더 프로세서와 비슷한 용도로 사용되는 프로세서들 중 AMD(3DNOW!), Radeon X1950 XT, ATTILA 등의 프로세서들에 비해 설계된 셰이더 프로세서는 다양한 명령어를 제공한다. 더불어 현재까지의 셰이더 프로세서의 설계 동향은 벡터 처리 방식을 적용하지 않고 있다. AMD(3DNOW!)는 간단한 벡터 명령어를 지원하지만 이는 다양한 벡터 명령어를 지원하는 것은 아니며, ATTILA 프로젝트는 설계된 셰이더 프로세서와 유사하게 벡터 방식의 셰이더 프로세서에 관한 연구를 진행하고 있지만 현재 소프트웨어 시뮬레이션 모델만이 완성되어있는 상태이다. 반면 통합 셰이더 프로세서를 위해 본 논문에서 구현된 제어 유닛은 모든 명령어들에 대해 벡터 처리가 가능한 구조이며 구현 및 검증이 이루어졌다.

이상과 같은 비교를 종합하였을 때 본 논문에서 구현된 제어 유닛은 동급의 셰이더 프로세서에 비해 많은 수의 명령어를 지원하며 사용하는 연산 유닛의 수에 비해 전체적인 성능도 우수하다.

V. 결론 및 향후 연구 과제

통합 셰이더 프로세서를 위해 본 논문에서 구현한 제어 유닛은 SIMD 방식으로 데이터 처리가 가능한 다중 연산 유닛을 갖는 비순차 이슈 방식의 벡터 프로세서를 지원한다. 또한 단일 프로세서로서 우수한 성능을 나타내므로 기존의 상용 그래픽 프로세서를 사용하여 연구하기 어려운 전역 조명 등의 그래픽 처리 알고리즘 개발에 필요한 셰이더 프로세서로 사용될 수 있으며, DSP 코어, 수치 해석 등 다양한 분야에 유용하게 사용될 수 있다.

향후 연구과제로는 상대적으로 높은 하드웨어 면적과 동작 속도를 차지하는 이슈 모듈의 패스를 줄이기 위한 방안 및 제어로직의 최적화, 대용량의 데이터 저장을 위한 메모리의 확장, 높은 동작 주파수를 얻기 위한 ASIC의 제작 및 다중 셰이더 프로세서에 관한 연구가 함께 이루어져야 할 것이다.

참 고 문 헌

[1] A. Watt, "*3D Computer Graphics Third Edition*", ADDISON WESLEY, 2000.
[2] W. K. Jeong, "A SIMD-DSP/FPU for High-Performance Embedded Microprocessors", *Phd Thesis*, Yonsei University, Dec. 2002.
[3] B. Atabek and A. Kimar, "Implementability of Shading Models for Current Game Engines", *ICCES*, pp. 427-432, 2008.
[4] K. Chung, C. Yu, D. Kim, and L. Kim, "Tessellation-Enabled Shader for a Bandwidth-Limited 3D Graphics Engine", *IEEE CICC*, pp. 367-370, 2008.
[5] Foley, van Dam. Feiner, Hughes, *Computer Graphics Principle and Practice*, Addison & Wesley, 1996.
[6] J. L. Hennessy and D. A. Patterson, "*Computer Architecture, A Quantitative Approach, Fourth Edition*", Elsevier, Sep. 2006.
[7] T. A. Moller and E. Haines, "*Real-Time Rendering*", A. K. Peters, 2002.
[8] B. Gooch and A. Gooch, "*Non-Photorealistic Rendering*", A. K. Peters, Natick, MA, 2001.
[9] J. Jeong, "A Proposal of 3D Graphics Rendering

Hardware Using Parallel Processing", *Master Thesis*, Yonsei University, Dec. 2001.
[10] B. Khailany, "Imagine: Media Processing with Streams", *IEEE Micro*, vol. 21, no. 2, pp. 35-46, Mar. 2001.
[11] I. Buck, et al., "Brook for GPUs: Stream Computing on Graphics Hardware", *in proceedings of ACM SIGGRAPH*, 2004.
[12] V. Moya, C. Gonzalez, J. Roca, A. Fernandez, and R. Espasa, "Shader Performance Analysis On a Modern GPU Architecture," in proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecuture(MICRO'05), pp. 355-364, Nov. 2005.
[13] http://www.amd.com/us/products/Pages/graphics.aspx.
[14] Texas Instruments, "TMS320C6713, TMS320C6713 Floating-Point Digital Signal Processors," sprs186c-december 2001-revised march 2003, Mar. 2003.
[15] Microsoft MSDN, *Vertex/Pixel Shader 3.0*.

저 자 소 개

김 경 섭(학생회원)
2002년 한남대학교 정보통신공학과 학사 졸업.
2009년 인하대학교 전자공학과 석사 졸업.
2009년～현재 인하대학교 전자공학과 박사과정.
<주관심분야 : 컴퓨터 구조, SoC & 임베디드 시스템 디자인, 병렬 및 분산 처리 시스템>

유 병 철(학생회원)
2010년 인하대학교 전자공학과 학사 졸업.
2011년～현재 인하대학교 전자공학과 석사과정.
<주관심분야 : 컴퓨터 구조, 병렬 및 분산 처리 프로그래밍>

최 상 방(평생회원)
1981년 한양대학교 전자공학과 학사 졸업.
1981년～1986년 LG 정보통신(주)
1988년 University of washinton 석사 졸업.
1990년 University of washinton 박사 졸업.
1991년～현재 인하대학교 전자공학과 교수.
<주관심분야 : 컴퓨터 구조, 컴퓨터 네트워크, 무선 통신, 병렬 및 분산 처리 시스템, Fault-tolerant computing>

이 윤 섭(학생회원)
2006년 경희사이버대학교 정보통신학과 학사 졸업.
2008년 인하대학교 전자공학과 석사 졸업.
2008년～현재 인하대학교 전자공학과 박사과정.
<주관심분야 : 컴퓨터 아키텍처, SoC & 임베디드 시스템 디자인, 차량용 네트워크 시스템>

정 진 하(정회원)
1992년 인하대학교 전자공학과 학사 졸업.
1994년 인하대학교 전자공학과 석사 졸업.
2010년 인하대학교 전자공학과 박사 졸업.
<주관심분야 : 컴퓨터 구조, 임베디드 시스템 디자인, 병렬 및 분산 처리 시스템>