논문 2010-47SD-5-8

벡터화된 SIMD 프로그램어블 통합 셰이더를 위한 특수 함수 유닛 설계

(Design of Special Function Unit for Vectorized SIMD Programmable Unified Shader)

정 진 하*, 김 경 섭*, 윤 정 희*, 서 장 원*, 최 상 방**

(Jin-Ha Jung, Kyeong-Seob Kim, Jeong-Hee Yun, Jang-Won Seo, and Sang-Bang Choi)

요 약

현실감 있는 3차원 그래픽 영상을 지원하기 위해서는 3차원의 그래픽 데이터를 기반으로 사실감을 부여하여 2차원 영상을 생성하는 렌더링 기술과 방대한 양의 데이터에 대해 복잡한 연산을 효율적으로 처리할 수 있는 고성능 그래픽 프로세서가 요구된다. 이로 인해 그래픽 하드웨어는 급속히 발전하였고 기존에 실시간 처리가 불가능했던 여러 고급 렌더링 효과들을 가능하게 하고 있다. 과거에 비해 셰이딩 기술이 발전하면서 사실적인 영상의 렌더링이 가능하게 되었으나 아직 많은 계산 시간을 필요로 하고 있다. 실사와 같은 영상을 빠르게 처리하기 위해서 그래픽 프로세서는 많은 데이터에 대해 복잡한 부동소수점 연산을 효율적으로 처리 할 수 있도록 다수의 연산유닛이 집적되는 방향으로 발전하고 있다. 본 논문에서는 프로그램어블 통합 셰이더 프로세서에서 고성능 3차원 컴퓨터 그래픽 영상을 지원하기 위해 특수 함수 유닛을 설계하고 구현하였다. 설계한 특수 함수 유닛에 대해 기능적 레벨의 시뮬레이션을 하여 동작을 검증 하였으며, FPGA Virtex-4(xc4vlx200)에 구현하여 하드웨어 리소스 사용율과 동작속도를 확인 하였다.

Abstract

Rendering technique generating 2 dimensional image to give reality and high performance graphical processor for efficient processing of massive data are necessary to support realistic 3 dimensional graphical image. Recently, graphical hardwares have evolved rapidly. This enables high quality rendering effect that we were unable to process in realtime. Improving shading technique enabled us to render realistic images but still much time is required for this process. Multiple operational units are being integrated in a graphical processor for effective floating point operation using massive data to process almost real looking images. In this paper, we have designed and implemented a special functional 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 special functional unit. Hardware resource usage rate and execution speed are measured implementing directly on FPGA Virtex-4(xc4vlx200).

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

I. 서 론

최근 몇 년간 그래픽 하드웨어의 급속한 발전은 기존에 실시간 처리가 불가능했던 여러 고급 렌더링 효과들을 가능 하게 하고 있다. 그 중 정점 셰이더와 픽셀 셰이더의 등장은 고정 파이프라인에 의해 정형화된 기존의 파이프라인에서 벗어나 유연하고 강력한 렌더링을 가능하게 하였다.[1] 또한 그래픽 프로세서는 방대한 폴리곤(Polygon) 데이터를 동시에 처리하기 위해서 작은 크기의 코어가 대규모로 집적되는 방향으로 발전되고 있으며, 그래픽 처리 보조용으로 사용되던 그래픽 프로세서의 연산 능력은 범용 프로세서를 뛰어 넘고 있다.[2]

현실감 있는 3차원 그래픽 영상을 지원하기 위해서

* 정회원, ** 평생회원, 인하대학교 전자공학과
(Electronic Engineering, Inha University)
접수일자: 2010년2월1일, 수정완료일: 2010년4월15일

는 방대한 양의 데이터와 복잡한 연산을 효율적으로 처리할 수 있는 고성능 그래픽 프로세서가 필요하다.[3~4] 개발 초기에 그래픽 프로세서는 z-버퍼 기반의 알고리즘 구현이 성능이나 효율성 면에서 우수하다고 인정된 후 계속해서 발전하여 왔다. 따라서 현재 사용되고 있는 그래픽 프로세서는 z-버퍼를 기반으로 숨겨진 면 처리와 지역 조명 처리 알고리즘 처리 등의 구조를 가지고 있다. 사실적 영상 처리를 위한 알고리즘을 현재의 그래픽 프로세서의 구조에 맞게 변형하여 구동하는 연구 흐름이 있었으나, 그래픽 프로세서의 가속을 받기 위한 부가적인 처리와 컴퓨터의 버스를 통한 데이터 교환에서 오는 비용으로 인하여 적절한 수준의 가속을 받기가 어렵다. 현재의 그래픽 프로세서 구조는 z-버퍼와 지역 조명으로 제한되어 있는 한계를 가지고 있으나, 이러한 한계는 그래픽 프로세서의 구조에 기인한 것으로 프로그램어블 셰이더 프로세서에 의한 것은 아니며 상용의 그래픽 프로세서를 사용하여 연구하기 어려운 그래픽 처리 알고리즘 개발에 필요한 프로그램어블 통합 셰이더 프로세서가 필요하다.

본 논문에서는 효율적인 그래픽 처리 알고리즘 개발에 사용될 수 있도록 DirectX 셰이더 명령어 중 높은 연산 능력을 필요로 하거나 특수한 명령어들을 수행하는 특수 함수 유닛(Special Function Unit)을 설계하였다. 설계된 SFU는 벡터화된 SIMD(Single Instruction Multiple Data) 프로그램어블 통합셰이더[5~7]에서 고성능 3차원 컴퓨터 그래픽 영상을 지원하기 위해 이용된다. 설계된 결과는 시뮬레이터를 통하여 동작을 검증하였으며 FPGA Virtex-4에 구현하였을 때, 전체 모듈의 동작속도는 최대 117 MHz이고, 총 7941개의 슬라이스가 사용되었다.

본 논문은 다음과 같이 구성된다. Ⅱ장에서는 그래픽 프로세서의 특징 및 기존 연구에 대해서 설명하고, 프로그램어블 셰이더의 전체적인 구성과 특징에 대해서 설명한다. Ⅲ장에서는 벡터화된 SIMD 프로그램어블 통합 셰이더 구조에서 DirectX 셰이더 어셈블리의 정점/픽셀 셰이더 3.0에 정의된 명령어 중 높은 연산능력을 필요로 하거나 특수한 명령어들을 수행하는 SFU를 제안하고 내부 로직에 대한 설계를 다룬다. Ⅳ장에서는 설계한 SFU의 동작 검증과 HDL 합성결과를 보여주며, 마지막으로 Ⅴ장에서는 본 논문의 결론 및 향후 연구 과제를 제시하며 끝을 맺는다.

Ⅱ. 관련 연구

본 장에서는 그래픽 처리의 특성 및 기존 연구에 대해서 설명하고, SIMD 구조의 프로그램어블 셰이더의 전체적인 구성과 특징에 대해서 설명한다.

1. 그래픽 처리의 특성 및 기존 연구

최근 사용되는 멀티미디어 데이터는 일반적으로 대용량이고 다양한 형태로 저장되어 있어 복잡한 구조를 갖는다. 그렇기 때문에 멀티미디어 데이터의 처리시간이 많이 걸리고, 연산 집약적인 특성을 갖는다. 또한 영화나 비디오게임 등 다양한 분야에서 사용되는 3차원 영상 역시 많은 수의 폴리곤 계산과 기하 연산, 렌더링(Rendering)등이 요구되므로 많은 시간과 복잡한 과정이 필요하다. 좀 더 사실적인 영상 구현을 위해 여러 가지 효과가 추가로 사용되면서 연산량은 점차 많아졌고, 기존의 프리렌더링(Prerendering)형태가 아닌 실시간 렌더링(Realtime Rendering)이 일반화되면서 신속한 연산처리의 필요성 또한 중요한 고려사항이 되었다.

멀티미디어 데이터와 3차원 영상에 사용되는 데이터는 다수의 요소로 구성되어 있으며, 이러한 구성 요소들을 한 번에 연산함으로써 효율적인 처리가 가능하다. 멀티미디어 데이터는 텍스트, 오디오, 이미지, 그래픽, 비디오 등의 데이터를 반복적으로 처리하는 과정을 거치므로 이를 병렬로 수행하면 효율적인 데이터 처리를 기대할 수 있다. 예를 들어 3차원 영상을 구성하는 각 픽셀 역시 (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 발생하기 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 연산 속도를 높일 수 있는 장점을 가지고 있다. 특히 하나의 명령어로 여러 개의 값을 동시에 계산하는 SIMD는 벡터 프로세서에서 많이 사용하는 방식으로 멀티미디어

분야 역시 많은 프로세서들이 SIMD 구조로 되어있다. SIMD는 인텔의 MMX, 스트리밍 SIMD 확장(SSE, streaming SIMD extensions), AMD의 3DNOW! 등에 적용되었으며, 최근에는 XBOX360, PS3와 같은 고화질의 3차원 게임 영상을 처리해야 하는 비디오 게임 콘솔 기기에도 적용되고 있다.

스트림 프로세서(Stream Processor)는 주어진 데이터 열에 대하여 일련의 연산을 수행하는 프로세서로써, 병렬처리, 의존성의 해결, 온 칩 메모리의 사용 등을 통해 사용자에게 보다 편리한 프로그램 환경을 제공한다. 일반적으로 멀티미디어 데이터 또는 3차원 영상에 사용되는 데이터 대부분은 데이터 레벨 병렬성을 가지므로 빠른 스트림 처리가 가능하다. Nvidia의 Geforce 8 시리즈와 ATI의 HD2000 시리즈부터는 스트림 프로세서가 도입되어, 기존의 픽셀 세이더와 정점 세이더 유닛, 기하 유닛을 대신해 데이터를 처리하는 통합 세이더 유닛의 성격을 가지게 되었고 하드웨어 구조가 단순화 되었다. 이러한 구조의 가장 큰 특징은 프로그래머가 프로세서의 용도를 임의로 지정할 수 있다는 점이다. 통합 세이더 유닛이 개발되기 이전에는 프로그래머가 한정된 자원 내에서 프로그래밍을 해야 하는 불편함이 있었다. 즉, 네 개의 픽셀 세이더와 한 개의 정점 세이더가 있다면, 해당 구조 내에 맞추어 프로그래밍을 해야 하는 한계가 있었다. 그러나 정해진 통합 세이더 유닛에서 픽셀, 정점, 기하 세이더의 수를 선택적으로 활용할 수 있게 됨에 따라, 특정 그래픽 처리 상황에 주요한 세이더를 더 많이 사용하는 유연한 프로그래밍이 가능해지게 되었다.[8~10]

2. SIMD 프로그램어블 세이더 프로세서 구조

프로그램어블 통합 세이더 프로세서는 DirectX 9의 정점/픽셀 세이더 명령어의 대부분을 프로세서에서 직접 실행할 수 있도록 구성된다. 즉, 프로그램 작성에 있어 다양하고 강력한 명령어를 제공함으로써 프로그램의 효율을 높이고 원하는 기능을 용이하게 구현할 수 있는 기반을 제공한다. 연산에 사용되는 데이터는 동시에 4입력 IEEE 754 단정도 부동 소수점 연산을 수행할 수 있도록 세이더 명령에서 정의된 기능을 지원한다.

일반적인 세이더 처리는 서로 상호 연관성이 없는 데이터에 대하여 동일한 동작을 반복한다. 이러한 특성을 이용하여 연산 파이프라인의 효율적인 사용을 추구할 수 있는 방법은 동일한 연산을 연속된 데이터에 대하여

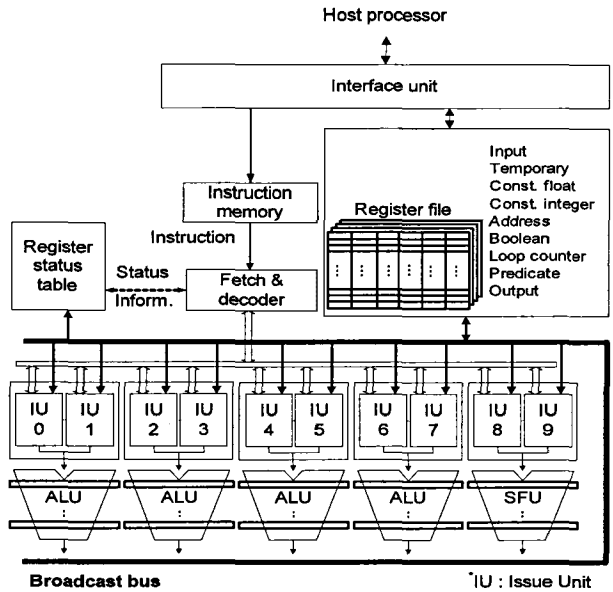


그림 1. 설계된 프로그램어블 통합세이더 프로세서
Fig. 1. Designed programmable unified shader processor.

처리하는 벡터 컴퓨팅 기법이다. 스칼라 프로세서의 피연산자는 단일 값이지만 벡터 프로세서의 피연산자는 연속된 데이터이다. 하지만 하드웨어의 한계 때문에 벡터의 길이를 무한정 길게 할 수 없으므로 본 논문에서 사용된 세이더 프로세서는 한 번의 프로그램 흐름에서 하나의 벡터를 구성하는 벡터 크기만큼의 번들을 대상으로 연속 처리하도록 구성되어 있다. 여기서 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.

스트림 처리는 지정된 데이터 셋(스트림)에 대하여 일련의 동작(커널)을 수행하는 것으로써, 프로그램의 패러다임 중 하나이다. 스트림 처리를 위하여 프로세서는 명령어의 병렬 처리가 가능한 구조이어야 하고, 명령어 간의 데이터 의존성을 검출하여 이를 해결할 수 있어야 한다. 설계된 프로그램어블 통합 세이더는 다수의 연산 유닛을 이용하여 동시에 여러 명령어가 수행될 수 있는 병렬 처리 구조이며, 레지스터 상태 테이블, 패치 및 디코더 유닛, 이슈 유닛, 내부 버스 등을 이용하여 데이터 의존성을 검사하고 그 결과에 따라 적절한 동작을 수행할 수 있도록 설계되었다. 설계된 프로그램어블 통합 세이더 프로세서의 구조는 그림 1과 같다.

Ⅲ. SFU 설계

프로세서의 연산 장치를 설계하기 위해서는 먼저 프로세서에서 실행될 명령어를 정의하고 명령어의 실행 과정을 분석한 후 명령어를 수행할 수 있는 최적화된 구조로 연산 장치를 설계해야 한다. 본 장에서는 벡터화된 SIMD 프로그램어블 통합 셰이더 구조에서 DirectX 셰이더 어셈블리의 정점/픽셀 셰이더 3.0에 정의된 명령어 중 높은 연산능력을 필요로 하거나 특수한 명령어들을 수행하는 SFU를 제안한다. 제안된 SFU는 각 연산 모듈을 파이프라인으로 구성하여 동일한 연산에 대한 연속적인 수행이 가능하도록 하여 전체적인 성능을 향상 시킬 수 있는 구조로 설계하였다.

1. SFU에서 수행할 수 있는 DirectX 명령어

표 1은 DirectX 정점/픽셀 셰이더 3.0에서 제시하고 있는 명령어 중 특수한 연산을 하는 명령어를 정리한 것이다. 각 명령어는 SFU에 하드웨어적으로 구현하였기 때문에 DirectX 셰이더 명령어를 직접 수행할 수 있는 장점이 있다.

표 1. SFU 명령어와 기능
Table 1. SFU instructions and functions.

명령어	기능
RCP	역수 연산
RSQ	상호제곱근 연산
EXP	2를 밑으로 하는 지수 (높은 정밀도)
LOG	2를 밑으로 하는 로그 (높은 정밀도)
LIT	조명지수 연산
POW	n 제곱 연산
SIN/COS	사인, 코사인 연산 (radian)
SQRT	루트연산
NRM	벡터 정규화
DST	벡터간 거리 연산
FRC	입력 값의 소수부 출력

2. SFU의 구조

SFU의 전체구조는 그림 2와 같이 구성된다. 각각은 스위즐(swizzle), 입력 변경자(input modifier), 특수함수처리(special function operation), 명령어변경자(instruction modifier), 쓰기 마스크(write mask), 컨트롤 파이프라인으로 구성된다.

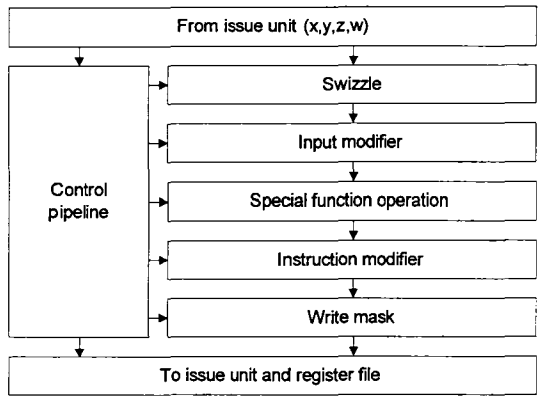


그림 2. SFU의 구조
Fig. 2. Structure of SFU.

가. 스위즐 모듈

스위즐 모듈은 SFU의 구성에서 최상위에 위치한 모듈로 입력 데이터를 받아 각 소스의 x, y, z, w 요소에 대하여 명령어에 포함된 스위즐 값을 사용하여 실제 입력 데이터의 위치를 결정하여 연산 유닛에 입력한다. 스위즐 모듈 하나의 내부 구조는 그림 3과 같다. 스위즐의 구성은 조합 논리 회로인 4입력 1출력 멀티플렉서 네 개를 이용하여 구성하였다. 각 멀티플렉서는 입력되는 각 소스 데이터의 x, y, z, w 값을 모두 입력으로 받고 명령어에 포함된 스위즐 선택 신호인 Mux_sel에 의해 출력되는 값을 선택함으로써 원하는 값을 다음 모듈로 전달한다.

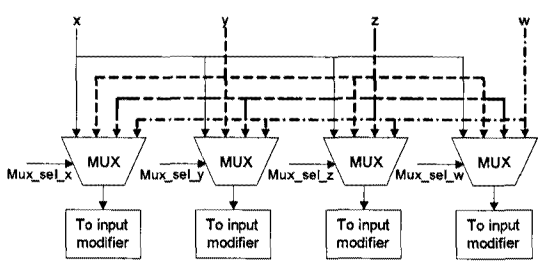


그림 3. 스위즐 모듈의 구조
Fig. 3. Structure of swizzle module.

나. 입력 변경자 모듈

입력 변경자 모듈은 조합 논리 회로로 구성하였으며 스위즐 모듈에서 입력되는 값이 바이패스, 절대치연산, 부호변환 모듈로 동일하게 입력되어 처리되고, 각 모듈에서 출력된 값은 멀티플렉서의 입력으로 전달되어 제어 파이프라인의 제어 신호에 의해 선택된 값이 출력되어 특수 연산 모듈의 입력으로 사용된다. 입력 변경자

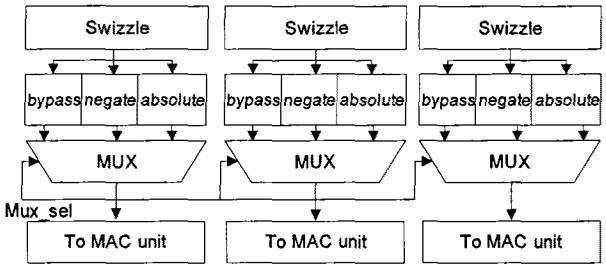


그림 4. 입력 변경자 모듈의 구조
Fig. 4. Structure of input modifier module.

모듈에서 수행되는 동작은 세 가지로 구분되는데 첫 번째는 입력 데이터의 수정 없이 바이패스, 두 번째는 절대치 연산(absolute), 세 번째는 입력데이터 값의 부호를 반전(negate)하는 것이다.

이러한 동작을 위한 입력 변경자 모듈의 구조는 그림 4와 같이 바이패스 모듈, 절대치 연산 모듈, 부호변환 모듈, 3입력 1출력의 멀티플렉서로 구성된다. 절대치연산 모듈은 입력되는 IEEE 754 부동 소수점 데이터의 최상위 비트인 부호 비트를 양수로 바꾸어 주며, 부호변환 모듈은 입력 데이터의 최상위 비트인 부호 비트를 반전시킨다.

다. 특수 연산 모듈

특수 연산 모듈은 4입력 32비트 부동 소수점 데이터 처리가 가능한 파이프라인 구조이며, 대부분의 명령어는 하드웨어적으로 구성되도록 설계하였다. 본 논문에서는 각 명령어의 연관성과 최대 요구 정밀도를 고려하여 그림 5와 같이 특수 연산 모듈을 설계하였다.

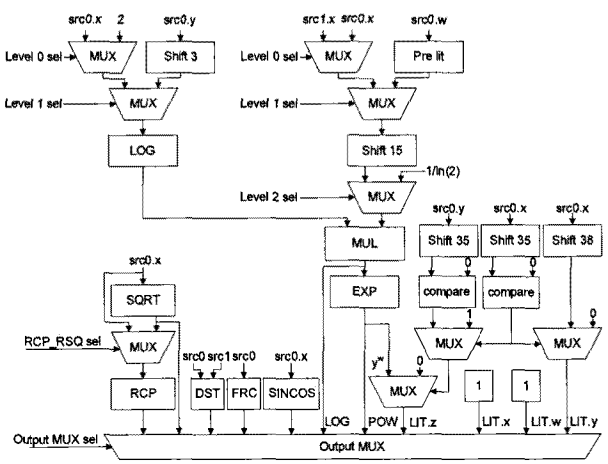


그림 5. 특수 연산 모듈의 구조
Fig. 5. Structure of special function operation module.

(1) RCP, SQRT RSQ, NRM 모듈

역수 연산을 하는 RCP와 루트 연산을 하는 SQRT, 루트의 역수를 구하는 RSQ간에는 서로 연관성이 있기 때문에 그림 6과 같이 구성하였다. 입력은 소스 레지스터 0의 x가 SQRT 모듈로 들어가거나 멀티플렉서로 들어가며 SQRT에서 받은 입력은 SQRT와 RSQ에서 사용되고 멀티플렉서로 들어간 입력은 RCP만을 위해 사용된다.

SQRT 모듈의 출력은 연산이 SQRT인 경우 바로 결과 값으로 사용되고, 연산이 RSQ이면 RCP의 입력으로 들어가 최종적으로 RCP 연산을 거쳐 RSQ의 결과가 된다. RCP는 소스 레지스터 0의 x값이 멀티플렉서를 통해 입력되어 RCP로 전달되어 계산된다. NRM 명령어는 SQRT와 나누기 명령어를 조합하여 계산된다.

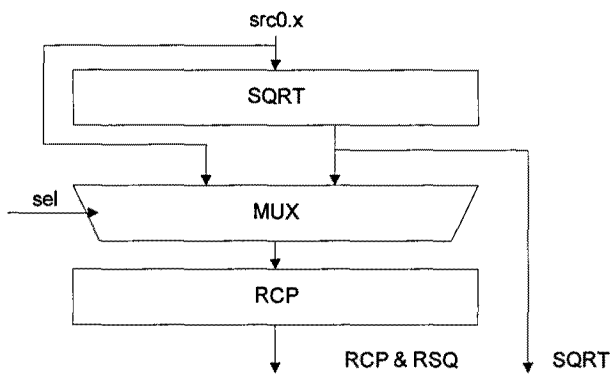


그림 6. RCP, SQRT, RSQ 모듈의 블록 다이어그램
Fig. 6. Block diagram of RCP, SQRT, RSQ module.

(2) DST 모듈

DST 명령어는 벡터간의 거리를 계산하기 위한 것으로써 그림 7과 같이 정의된다.

이와 같은 연산과정을 설계하기 위해 소스 레지스터 0의 y와 소스 레지스터 1의 y를 곱셈기를 사용하여 출력하고, 소스 레지스터 0의 z와 소스 레지스터 1의 w를 곱셈기의 연산 속도에 맞추어 지연하여 출력하며, x의 결과 값은 항상 1이므로 상수 1을 출력한다. DST 연산

```
dest.x = 1
dest.y = src0.y × src1.y
dest.z = src0.z
dest.w = src1.w
```

그림 7. DST 명령의 연산
Fig. 7. Operation of DST instruction.

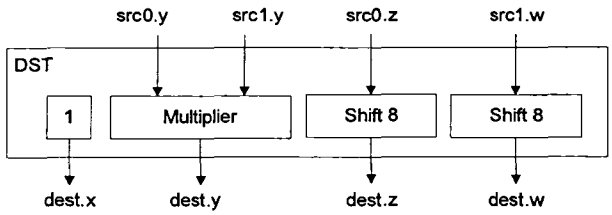


그림 8. DST 모듈의 블록 다이어그램
Fig. 8. Block diagram of DST module.

의 블록 다이어그램은 그림 8과 같다.

(3) FRC 모듈

FRC 명령어는 입력의 각 성분에 대한 소수부를 구하는 연산이며 그림 9와 같이 정의된다.

floor 함수는 입력된 값보다 크지 않은 정수를 반환하는 함수로써, FRC 명령의 결과는 0.0에서 1.0 범위내의 결과를 가지게 된다. 구현한 FRC 모듈은 x, y, z, w에 대해서 동시에 FRC 명령어를 수행하기 위해서 그림 10과 같이 입력의 각 요소마다 FRC 모듈을 연결하여 구성하였다. FRC의 내부 구조는 무한대 값이나 부정확한 값에 의해 에러 처리가 필요한 값이 입력이 되었을 때 예외 처리를 하는 모듈과 0.0에서 1.0 범위 내에 입력이 들어왔을 때 연산을 하지 않고 통과시켜 FRC 연산의 결과를 얻는 모듈, 그리고 두 가지를 제외한 다른 입력에 대하여 FRC 연산을 수행하는 모듈로 나누어진다. FRC 연산을 하는 모듈은 고정 소수점으로 FRC 연산을 하는 부분과 계산된 결과를 IEEE 754 형식의 부동 소수점으로 변환하는 부분으로 구성하였다.

그림 10은 그림 9에서 가장 오른쪽 파이프라인의 일반적인 입력 값에 대한 연산 모듈에 대한 내부 구조를

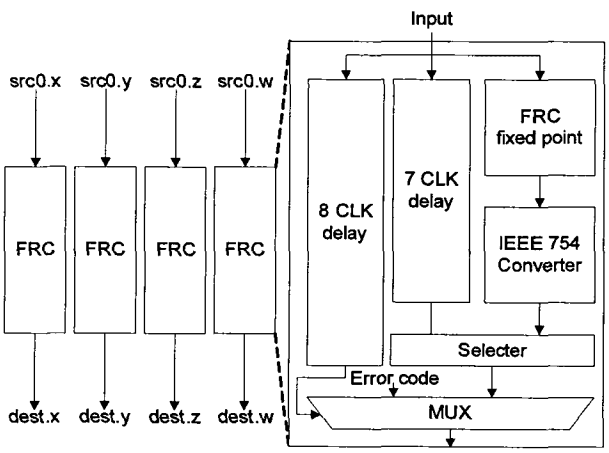


그림 9. FRC 모듈의 블록 다이어그램
Fig. 9. Block diagram of FRC module.

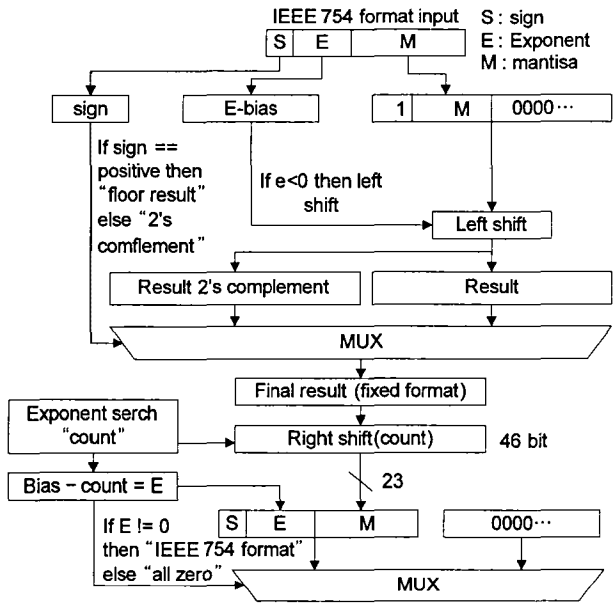


그림 10. FRC 고정 소수점 연산과 IEEE 754 변환
Fig. 10. Fixed point FRC operation and IEEE 754 conversion.

나타낸다. 연산과정의 첫 번째 과정은 정밀도와 관련된 부동 소수점의 소수부(Mantissa)의 유효 숫자를 최대한 확보하기 위해 23 비트의 소수부를 46 비트의 버퍼에 저장한다. 이때 부동 소수점의 숨겨진 비트를 버퍼의 MSB에 표현하였고, 소수부를 저장한 나머지 공간은 0으로 채워 넣는다. 동시에 지수부에서는 지수 값을 읽고 쉬프트 연산을 통해 소수만 고정 소수점 형식으로 추출한 후, 부호부의 비트를 읽어 양수이면 전단의 결과를 다음 단으로 보내고 음수이면 2의 보수 연산을 통해 고정 소수점에서 FRC 연산을 마친다. 최종적으로 결과를 IEEE 754 부동 소수점 형태로 변환시키기 위해 고정 소수점 결과의 최상위 비트에서 최하위 비트 방향으로 1이 나올 때까지 카운터를 증가시킨다. 이때 나온 카운터를 IEEE 754의 지수부 바이어스 127에서 빼면 고정 소수점 결과에 대한 지수부의 값을 구할 수 있고 카운터를 사용하여 고정 소수점 결과에 대해 쉬프트 연산을 하여 소수부의 값을 얻게 된다. FRC 연산 결과는 항상 양수이므로 부호는 항상 양수로 고정하여 최종적인 FRC 부동 소수점 결과를 얻는다. 이때 지수부의 모든 비트가 0이면(-127) 최종 결과를 모두 0으로 세트시켜 결과를 출력한다.

(4) SINCOS 모듈

SINCOS의 정확한 결과를 구하기 위해서는 많은 하

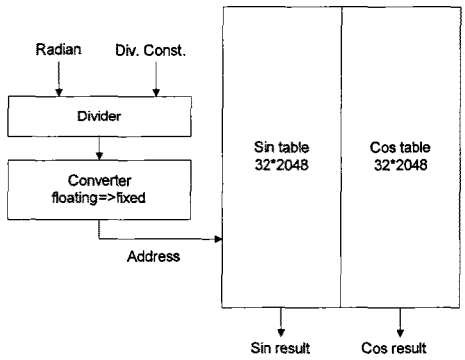


그림 11. SINCOS 모듈의 구조
Fig. 11. Structure of SINCOS module.

드웨어와 복잡한 연산이 필요하므로 본 논문에서는 사전에 계산한 결과를 FPGA의 블록 메모리에 저장한 후 구하고자 하는 값에 해당하는 결과를 테이블에서 읽어 출력하도록 구성하였다. SINCOS 모듈의 구조는 그림 11와 같으며, 블록 메모리에 저장되어 있는 SIN과 COS 값을 읽어오기 위한 방법은 다음과 같다.

SINCOS 모듈로 입력되는 값의 범위는 -π에서 π까지이며, 이를 2048개의 테이블로 구성하여 π/1024 라디안의 해상도로 입력을 처리한다. 따라서 SIN 테이블의 구성은 0 번지는 SIN(0)의 값, 1024 번지는 SIN(π) 값, 1025 번지는 SIN(-π/1024)의 값, 2048번지는 SIN(-π+π/1024)의 값으로 구성하며, COS 테이블의 값 역시 같은 방법으로 구성한다. 이렇게 구성된 테이블에 입력되는 각도를 π/1024로 나눈 후, 이를 반올림하여 정수로 변환하여 이를 테이블의 어드레스로 사용함으로써 SIN과 COS의 값을 구할 수 있다.

(5) LOG 모듈

LOG는 이진로그 함수의 값을 구하는 명령어로써, 최소 21 비트의 정밀도가 필요하다. 이진로그의 값을 구하는 방법은 참조 테이블(Look-up table) 기반의 알고리즘과 반복 알고리즘을 사용하는 것이다. 참조 테이블 기반의 알고리즘은 빠른 연산속도와 간단한 하드웨어 구조를 가지는 장점이 있으나 정밀도가 낮은 단점이 있다. 반면 반복 알고리즘은 높은 정밀도를 가지나 연산속도가 느리다는 단점이 있다. 반복 알고리즘이 연산속도가 느린 이유는 이용되는 테일러 시리즈(Taylor series)에 나눗셈과 같은 느린 연산이 사용되고 연산의 횟수가 많기 때문이다. 그래서 덧셈기와 쉬프터(Shifter)와 빠른 수렴 알고리즘을 사용하는 곱정규화(Multiplicative Normalization) 방법을 사용하여 높은 정밀도와 비교적 빠른 속도로 동작하는 LOG 모듈을 설계하였다. 설계된 LOG 모듈은 자연로그의 결과가 나오기 때문에 이진로그의 결과 값을 얻기 위해 LOG 모듈 외부에 곱셈기를 사용하여 이진로그의 결과를 얻는다. 자연로그를 구하기 위한 곱정규화 방법으로 계산하는 식은 식 (1)과 같다.

$$
\begin{aligned}
&y = \ln(x) \\
&\frac{1}{x} \approx \prod_{j=0}^{m} \ln(p[j]) = \prod_{j=0}^{m} \ln(1 + s_j 2^{-j}) \\
&\ln(x) \approx -\sum_{j=0}^{m} \ln(p[j]) = -\sum_{j=0}^{m} \ln(1 + s_j 2^{-j}) \\
&y[j+1] = y[j] - \ln(p[j]) \\
&y[m+1] = y[0] + \ln(x)
\end{aligned}
\tag{1}
$$

식 (1)에서 x는 입력받은 부동 소수점 입력의 가수부를 의미한다. 이 값을 이용하여 w[0]을 초기화 시키고, y[0]은 0으로 초기화 시킨다. 두 번째 순환부분에서 m만큼 루프를 돌게 되는데 m은 입력받은 부동 소수점의 가수부의 비트 수로 결정하게 된다.

단정도 부동 소수점 가수부는 23 비트를 사용하고 히든 비트(hidden bit)로 1 비트를 사용하기 때문에 m은 24로 설정하였다. s값은 w[j] 값의 범위에 따라 0, -1, 1 중 하나를 가지게 된다. L은 현재 루프 횟수를 나타내는 j에 따라 값이 결정되며, y[m+1]은 ln(x)의 근사치가 된다. 이때 입력받는 x의 범위는 0.5에서 1사이의 소수가 된다. 실질적인 사용을 위해서는 IEEE 754 단정도 부동 소수점의 표현 범위에 대한 계산이 가능하도록 확장이 필요하다, 기존 알고리즘을 확장시키는 방법은 식 (2)와 같다.

$$
\begin{aligned}
&v = -1^{s} \times 2^{E} \times 1.M \\
&x = 2^{(E-1)} \times 0.M \\
&\ln(x) = \ln(2^{(E-1)} \times 0.M) \\
&= (E-1) \times \ln(2) + \ln(0.M)
\end{aligned}
\tag{2}
$$

곱정규화 방법은 루프 횟수가 늘어날수록 구하려는 값에 더욱더 가까운 근사치가 나오지만, 하드웨어 로직과 연산속도에 영향을 미치게 된다. 이러한 문제를 해결하기 위해서 곱정규화 식에서 식 (3)과 같이 고유의 오차를 구한 뒤 멱급수로 풀면 z에 대한 다항식으로 표현한 방법을 이용할 수 있다. 이때 2차 계수를 ln(x)의 근사치인 y[m+1]에 더하게 되면 좀 더 높은 정밀도의 결과를 얻을 수 있게 된다. 또한 x[m+1]과 y[m+1]은 연산상의 의존도가 없으므로 한 루프에 x와 y를 동시에

구할 수 있어 루프 횟수 m을 반으로 줄인다 해도 높은 정밀도는 유지가 된다[11].

$$
\begin{aligned}
& y[m+1] \approx y[0] - \ln(p[j]) \\
& error = \ln(x) - y[m+1] \\
& \ln(x) = \ln(x \times \frac{\prod p[j]}{\prod p[j]}) \\
& = \ln(x \times \prod_{j=0}^{m} p[j]) - \prod_{j=0}^{m} \ln(p[j]) \\
& \ln(x) = \ln(x[m+1]) + y[m+1] \\
& \therefore error = \ln(x[m+1]) \\
& \ln(z) = (z-1) - \frac{(z-1)^2}{2} + \cdots (0 < z \le 2) \\
& \ln(x) = y[m+1] + x[m+1] - 1
\end{aligned}
\tag{3}
$$

식 (3)을 반영하여 설계한 이진로그 연산기의 블록 다이어그램은 그림 12와 같다. Input 단에서 단정도 부동 소수점으로 연산될 값을 입력 받은 뒤 Normalization 단에서 sign 값을 참고하여 입력받은 부동 소수점의 지수부와 가수부를 분리하게 된다. 내부의 반복적인 계산은 부동 소수점을 사용하는 것보다 고정 소수점을 사용하는 것이 하드웨어적인 측면이나 실행 속도의 측면에서 유리하다. 따라서 이진로그의 계산부는 모두 고정 소수점 방식으로 설계하였다. 부동 소수점으로 받은 값을 고정 소수점으로 변환시켜 줄 때 입력범위 확장 방법을 적용하기 위해서 가수부를 오른쪽으로 1 비트 이동시킨 후 xx.xx 포맷의 고정 소수점 형식으로 변환하여 x[0]에 저장한다. Initialization 단에서 입력 받은 x[0]을 이용하여 w[0]을 초기화하고 y[0]에

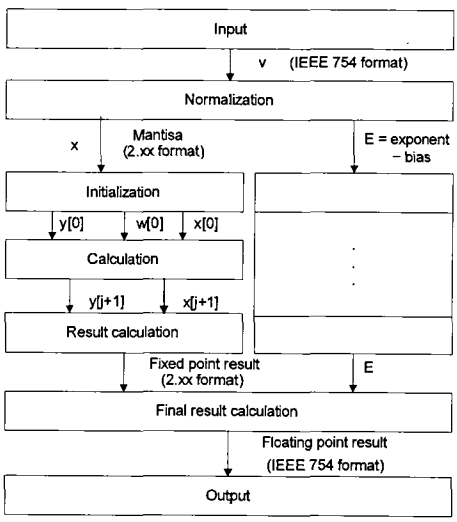


그림 12. LOG 모듈의 블록 다이어그램
Fig. 12. Block diagram of LOG module.

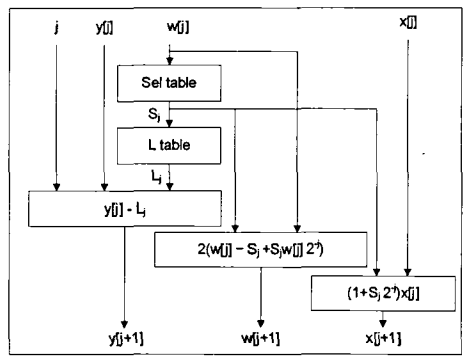


그림 13. LOG Calculation 모듈의 블록 다이어그램
Fig. 13. Block diagram of LOG Calculation module.

는 0을 대입하여 각 변수 초기화를 마친다.

Calculation 단에서는 입력받은 x, w, y 값을 곱정규화 방법으로 자연로그를 계산한다. 이때 Calculation의 블록 다이어그램은 그림 13와 같다. m은 입력받은 v의 가수부의 정밀도 비트 수가 되므로 히든 비트까지 합하여 24가 되고 루프 횟수를 반으로 줄이기 위한 방법을 적용하였으므로 m/2 = 12회의 연산으로 결과를 구하게 된다. 계산중 Lj 값은 테이블로 만들어 사용하였다. 계산 후 x[j+1]과 y[j+1]로 Result calculation 단에서 고정 소수점 형태의 최종 값을 도출한다. 마지막 계산인 Final result calculation 부분에서는 고정 소수점 형식의 결과와 Normalization 단에서 받은 지수부인 E 값을 이용하여 최종 결과를 계산한다. 이때 E 값은 -127~127까지 값을 가지므로 모든 E에 대한 (E-1)*ln(2)의 결과를 테이블로 작성한 후 이를 참조하여 Final result calculation을 하고 고정 소수점을 부동 소수점으로 변환시켜 결과를 얻는다.

곱정규화로 설계한 모듈은 자연로그이며 세이더의 요구대로 이진로그를 얻기 위해서는 식 (4)를 이용하여

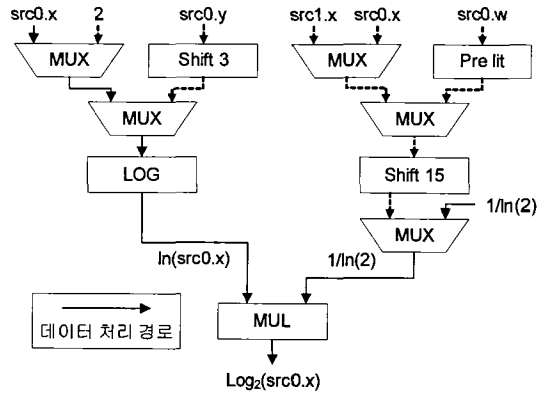


그림 14. 이진 로그 연산을 위한 블록 다이어그램
Fig. 14. Block diagram of base-2 LOG calculation.

변환함으로써 가능하다.

$$\log_2 x = \log_e x \times \frac{1}{\log_e 2} \tag{4}$$

자연로그를 이진로그로 변환하기 위한 하드웨어 구조는 그림 14와 같다. 설계한 자연로그 모듈로 입력된 소스 0의 x값을 계산한 결과와 오른쪽 멀티플렉서를 통해 입력된 1/ln(2)의 값을 곱셈기를 통하여 최종적인 이진로그 값을 얻을 수 있다.

(6) EXP 모듈

EXP 명령은 지수함수의 값을 구하는 명령어이다. 지수함수 연산을 하기 위해 LOG 함수와 마찬가지로 반복수행 알고리즘을 사용하고 연산의 속도 향상을 위해 합정규화(additive normalization) 방법을 사용하여 구현하였다. 합정규화 방법을 이용해 지수함수의 결과를 구하는 알고리즘은 그림 15와 같다. 합정규화 방법의 알고리즘은 입력 값의 범위가 $-\ln(2)<x<\ln(2)$로 한정되어 있으므로 IEEE 754 단정도 부동 소수점의 표현 범위

1. INITIALIZE

$y[0]=e^{-0.5}$, $w[0]=x+0.5$

2. RECURRENCE

for $j=0$ **to** m

$s_j = sel(\hat{w}[j])$

$w[j+1] = 2(w[j]-L_j 2^j)$

$y[j+1] = y[j]-y[j]s_j 2^{-j}$

end

3.RESULT

$y[m+1] \approx e^x$

$$sel(\hat{w}[j]) = \begin{cases} 1 & \text{if } w[j] \geq 0.5 \\ 0 & \text{if } -0.5 \leq w[j] \leq 0.25 \\ -1 & \text{if } w[j] \geq -0.75 \end{cases}$$

$$L_j = \begin{cases} \ln(1+2^{-j}) & \text{if } S_j = 1 \text{ and } j \leq m/2 \\ \ln(1-2^{-j}) & \text{if } S_j = -1 \text{ and } j \leq m/2 \\ 0 & \text{if } S_j = 0 \text{ and } j \leq m/2 \\ s_j 2^{-j} & \text{if } j > m/2 \end{cases}$$

그림 15. 합정규화 방법을 사용한 지수함수 알고리즘
Fig. 15. Exponential function algorithm using additive normalization.

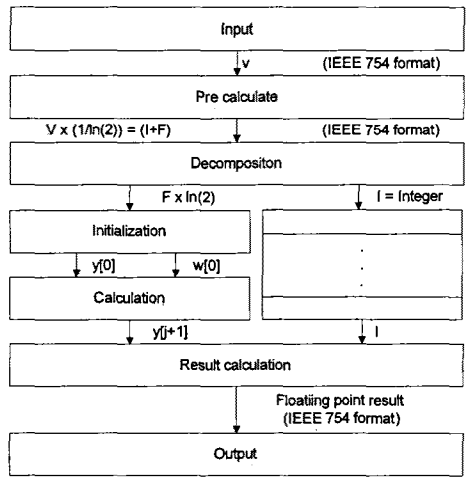


그림 16. EXP 모듈의 블록 다이어그램
Fig. 16. Block diagram of EXP module.

대해서 연산이 가능하게 하기 위해 입력 값의 범위를 개선한 식 (5)와 같은 방법을 사용한다.

식(5)는 입력을 I+F의 형태로 변환한 후 지수함수 알고리즘의 입력으로 F값을 사용하여 계산하고 그 결과에 2의 I승을 곱해 최종결과를 얻는다.

$$\begin{aligned} e^{inx} &= e^{inx(\log_2 e)(\log_e 2)} \\ &= e^{(I+F)\ln(2)} = e^{I\times\ln(2)} \times e^{F\times\ln(2)} = 2^I e^x \end{aligned} \tag{5}$$

식 (5)를 참고하여 만든 하드웨어 모듈 구조는 그림 25과 같다. 입력이 들어오면 (I+F)의 형태로 만들기 위해 Pre calculate 모듈에서 입력 값 v에 (1/ln(2))를 곱한다. 그 결과는 Decomposition 모듈에서 입력받은 값의 지수부를 참고하여 F값과 I값으로 분리하여 다음 단으로 넘겨주게 된다. F값을 받은 Initialize 모듈은 식 (5)의 초기화 단계를 수행하여 y, w값을 Calculation 모듈로 넘긴다.

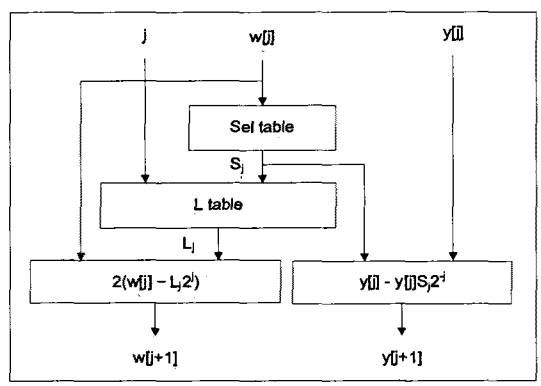


그림 17. EXP Calculation 모듈의 블록 다이어그램
Fig. 17. Block diagram of EXP Calculation module.

Calculation 모듈은 그림 16과 같이 설계하였다. 그림 16의 RECURRENCE에 해당하는 연산을 하는 24단의 계산 파이프라인을 구성하여 결과를 얻는다. 그림 17의 Result calculation 모듈에서는 Calculation 모듈에서 계산한 y[j+1]값을 이용하여 식 (5)의 계산을 수행하여 지수함수의 최종 결과를 얻는다.

(7) POW 모듈

POW는 그림 18과 같이 입력받은 두 값에 대해서 거듭제곱 연산을 하는 명령어로 스위즐 단에서 선택한 소스 레지스터 0과 소스 레지스터 1의 x, y, z, w 인자들 중에서 하나를 선택하여 입력으로 받게 된다.

POW 명령을 수행하기 위해서 따로 모듈을 분리하여 구현하지 않고 식 (6)과 같이 기존의 LOG 모듈과 EXP 모듈을 사용하여 POW 연산을 하도록 구성하였다.

$$\exp(src1 \times \log(src0)) = src0^{src1} \quad (6)$$

POW 연산을 처리하기 위한 블록 다이어그램은 그림 19와 같다. 소스 레지스터 0의 x가 LOG 모듈을 통해 ln(src0.x)값이 나오게 되고, 컨트롤 파이프의 제어에 따라 선택된 소스 레지스터 1의 x값과 곱셈 연산을 하게 된다. 이 결과는 EXP 모듈을 통과하면서 식 (6)과 같이 최종적인 POW 값을 얻게 된다.

```
dest = pow(abs(src0), src1)
```

그림 18. POW 명령의 연산
Fig. 18. Operation of POW instruction.

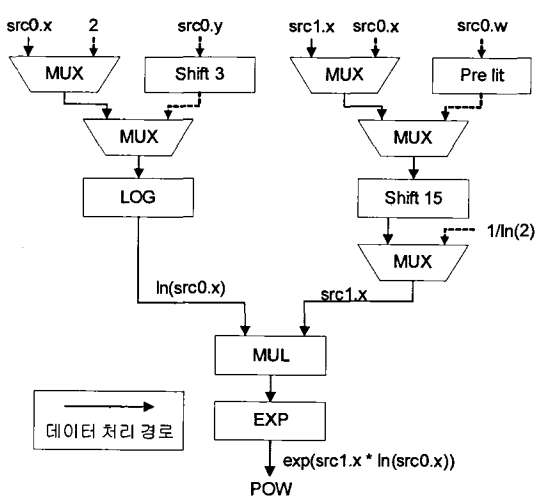


그림 19. POW 함수의 블록 다이어그램
Fig. 19. Block diagram of POW function.

(8) LIT 모듈

LIT는 내적(dot product)과 지수(exponent) 연산을 통해서 조명 지수 계수를 계산해 조명에 대한 부분적인 지원을 해주는 명령어로써 그림 20과 같이 정의된다.

LIT 연산은 기존에 설계한 LOG, POW, EXP 연산모듈을 사용하여 최종 결과를 얻을 수 있다. LIT 연산을 하드웨어로 구현한 것은 그림 22과 같다. 소스 레지스터 0의 w인자는 그림 21의 첫 번째 if 문을 수행하기 위해 그림 20의 Pre lit 모듈 안에서 maxpower보다 큰지, -maxpower보다 작은지, -maxpower<w<maxpower 인지 비교되어 그 결과에 따라 처리되어 결과가 곱셈기의 입력으로 보내진다. 이 과정에서 소스 레지스터 0의 y인자는 LOG 연산을 통하여 곱셈기의 입력으로 들어간다. 두 입력을 곱한 결과는 EXP 모듈을 통해 POW(src.y, power)의 연산 결과를 준비한다.

그림 20의 두 번째 if 문을 수행하기 위해 소스 레지스터 0의 x값과 y값은 POW(src.y, power) 계산과 동기가 맞도록 쉬프트 모듈에 대기하고 있다가 compare 단을 통해 0보다 작은지 비교하게 되고 x의 비교 결과는 최종 LIT의 y인자에 x와 0중 어느 값이 들어갈 지 선택하게 된다. 또한 이 결과는 최종 LIT의 z값에 POW(src.y, power)와 0중 어느 값이 들어갈지 선택하기 위해서 x를 비교한 결과를 참고하여 y를 비교한 결과를 반영할 것인지 아닌지를 선택하게 된다. x가 0보다 작

```
dest.x = 1
dest.y = 0
dest.z = 0
dest.w = 1

power = src.w
maxpower =127.9961

if (power < -maxpower)
    power = -maxpower
elseif (power > maxpower)
    power = maxpower
end

if (src.x > 0)
    dest.y = src.x
    if (src.y > 0)
        dest.z = pow(src.y, power)
    end
end
```

그림 20. LIT 명령의 연산
Fig. 20. Operation of LIT instruction.

지 않고 y 또한 0보다 작지 않다면 최종 멀티플렉서 단을 통해 POW(src.y, power) 값이 LIT의 z 결과 값이 되고 그렇지 않으면 0을 출력하여 LIT 연산을 마치게 된다. 그림 20에서 LIT의 x와 w값은 무조건 1을 출력하고 있기 때문에 x와 w의 LIT 결과는 항상 1이 되도록 설계하였다.

라. 명령어 변경자 모듈

명령어 변경자 모듈은 특수 연산 모듈에서 처리된 결과를 레지스터에 쓰기 전에 연산 결과를 그대로 전달하거나 포화시킨다. 포화는 연산 결과가 1보다 크면 1로 0보다 작으면 0으로 변경하는 것이다. 명령어 변경자 모듈의 구조는 그림 22와 같다. 명령어 변경자는 2입력 1출력 멀티플렉서, 비교기(less than, greater than), 4입력 1출력 멀티플렉서, 쉬프트 레지스터(shift register)이다. 연산 결과의 [0,1]사이의 값을 추출하기 위해 특수 연산 모듈의 결과와 0값을 멀티플렉서의 입력으로 받고 포화 선택 신호에 의해 멀티플렉서의 출력이 결정된다.

멀티플렉서 출력을 두 비교기에 각각 입력한 뒤 0 및 1과 비교하여 그 결과를 출력한다. 비교기에서 출력된

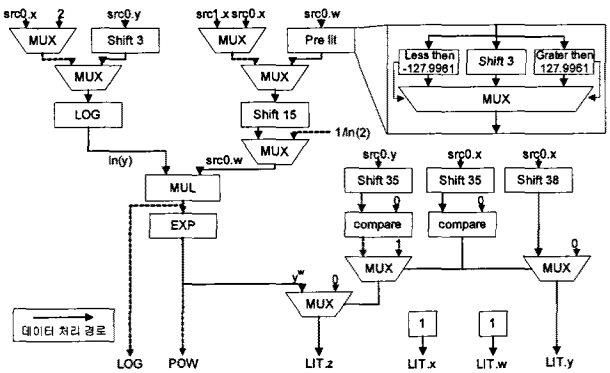


그림 21. LIT 명령의 데이터 흐름
Fig. 21. Data flow of LIT instruction.

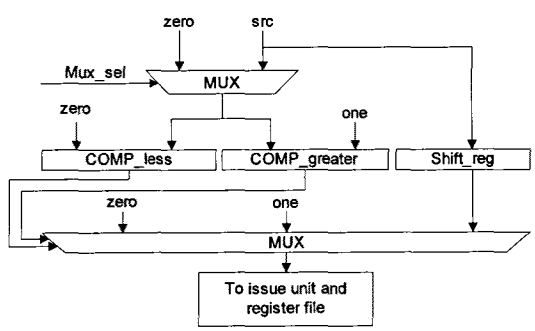


그림 22. 명령어 변경자 모듈의 구조
Fig. 22. Structure of instruction modifier module.

신호는 최종 출력을 결정하는 멀티플렉서의 선택 신호로 사용되어 최종 출력을 결정한다. 쉬프트 레지스터는 최종 멀티플렉서의 입력에 원래의 값과 비교된 값이 동일한 시점에 입력되게 하기 위해 비교기의 파이프라인 수만큼 데이터를 지연하는 역할을 수행한다.

마. 제어 파이프라인

제어 파이프라인은 연산 유닛의 데이터 패스를 결정하여 올바른 연산이 수행될 수 있도록 하기 위한 부분으로 연산 유닛이 정확한 동작을 하도록 제어한다. 또한 이슈 유닛으로부터 입력받은 제어 정보에서 연산의 종류에 따라 제어 정보의 입력 위치를 결정하는 멀티플렉서와 연산의 종류별 깊이에 맞게 제어 정보를 지연하여 특수 연산 모듈로 제공하기 위한 쉬프트 레지스터로 구성되며 그 구조는 그림 23과 같다.

제어 파이프라인은 제어 정보의 유효 비트(Valid bit)인 최상위 비트를 검사하여 그 값이 0이면 모든 제어 정보를 0으로 설정하여 제어 파이프라인 내부로 전달하여 최종적으로 연산 유닛에서 출력되는 값이 연산의 결과가 아닌 무의미한 값임을 표시하게 된다. 또한 유효 비트가 1일 경우에는 전달받은 내용을 제어 파이프라인 내부의 각 멀티플렉서 입력으로 전달하여 추후 이루어지는 동작을 제어하도록 하며 최종적인 출력이 적절한 연산의 결과임을 다른 유닛에 전달한다. 내부로 제어 정보를 전달할 때는 입력되는 연산의 종류에 따라 해당

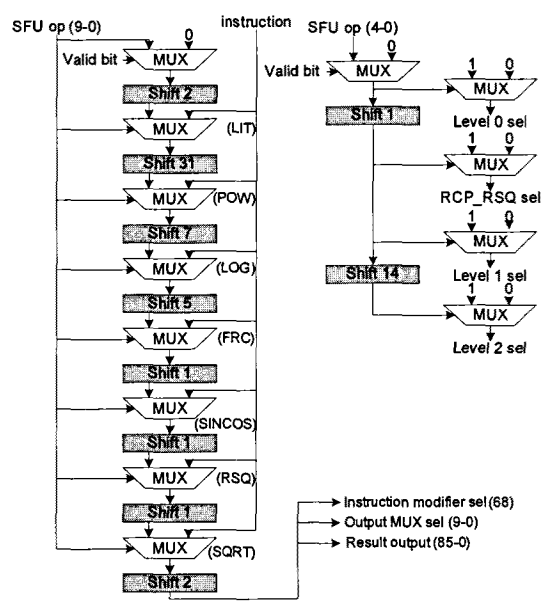


그림 23. 제어 파이프라인 구조
Fig. 23. Structure of control pipeline.

하는 멀티플렉서가 현재의 정보를 받아 출력하여 적절한 위치에 제어 정보가 삽입될 수 있도록 하고, 선택된 입력 지점이 제일 상위의 쉬프트 레지스터가 아닌 경우에는 최상위의 입력에 0을 주어 초기화된 값이 유지되도록 한다. 이는 아랫단의 멀티플렉서들의 구조를 단순화하기 위하여 현재 입력된 값을 받도록 선택되지 않았을 때는 상위단의 쉬프트 레지스터에서 값을 받도록 되어있기 때문에 모든 명령어의 처리가 끝난 후에 초기화된 값이 아래로 전달되어 정확한 동작을 할 수 있도록 하기 위해서이다. 사용된 멀티플렉서는 2입력 1출력 멀티플렉서로 하나의 입력은 최상위단의 제어 정보이고 다른 하나의 입력은 쉬프트 레지스터의 출력을 사용하며 최상위단의 제어 정보에 의해 어떤 입력을 출력할 것인가 선택된다.

제어 파이프라인은 그림 5의 특수 연산모듈이 올바르게 수행될 수 있도록 제어 하기위한 부분으로써 연산의 최종 출력을 결정하는 파이프라인과 중간 단계의 결과를 선택하는 파이프라인으로 구분된다. 중간 단계의 결과를 선택하는 파이프라인은 그림 23에서 Level 0 sel, Level 1 sel, Level 2 sel, RCP_RSQ sel이며 이는 RSQ, RCP, LIT, LOG, EXP와 같이 어떤 연산을 할 것인지 결정하는 역할을 한다. 연산의 최종 출력을 선택하는 제어 파이프라인은 수행되는 연산 중 연산 수행 시간이 가장 긴 명령어를 기준으로 최대 깊이를 설정하여 설계하였다. SFU에서 수행되는 연산이 종류별로 다른 파이프라인 깊이를 가지므로 최종 결과를 선택하는 제어 파이프라인의 구조는 각 연산이 시작되고 결과가 출력되는 시점에서 멀티플렉서 선택 신호를 전달하도록 구성된다. 제어 파이프라인으로 입력되는 제어 정보의 구성은 그림 24와 같으며 하나의 완성된 제어 정보는 85 비트로 구성되어 연산이 수행될 때 데이터와 동시에 제어 파이프라인으로 입력된다.

그림 24의 SFU op 항목은 특수 연산 모듈에서 이루어지는 연산의 데이터 경로를 결정하는 멀티플렉서들의 제어 신호로 전체 11 비트로 구성하였다. 특수 연산 모듈의 구조에서 각 연산별 데이터 처리 경로를 찾고 멀티플렉서들의 입출력 관계를 고려하여 제어 비트를 구분하여 SFU의 모든 명령이 수행될 수 있도록 중복 되지 않는 범위에서 제어 비트를 결정하였다. Compare condition 항목은 그림 4에서 ALU(Arithmetic Logic Unit)[6]의 내부 연산 모듈 중 ADD, SUB, COMPARE의 동작을 정의 하는 항목이다. Swizzle 항목은 입력되는 데이터의 스위즐을 결정하기 위해 사용된다. 입력되는 데이터 수는 연산의 종류에 따라 차이가 있으므로 최대로 입력되는 데이터 수만큼 스위즐을 처리할 수 있도록 하였다. 데이터는 최대 세 개까지 처리될 수 있으며 각 데이터가 x, y, z, w로 구성되어 있으므로 하나의 데이터 당 2비트의 제어 정보가 필요하고 최대 입력데이터 수만큼 구성된 총 비트 수는 24비트의 스위즐 제어 정보가 필요하다. Register type 항목과 Counter 항목은 연산이 완료된 데이터가 저장 되어야 할 레지스터의 종류와 위치를 결정하기 위해 필요한 항목이다. 이 값은 실제 연산에 사용되는 항목은 아니며 연산 결과를 저장하기 위한 상위 컨트롤 부분에서 필요에 의해 컨트롤 정보에 포함되어 들어오는 정보이다. 레지스터 타입은 4 비트로 구성되어 있으며, 카운터는 7 비트로 구성되어 있다. 입력 변경자 항목은 입력되는 데이터의 입력 변경자를 결정하기 위해 사용되는 항목으로서 스위즐 항목과 동일하게 입력되는 최대 데이터 수에 대한 처리가 가능하도록 하나의 데이터가 2비트의 제어 신호로 구성되어 총 8 비트로 구성된다. 각 2비트 정보에 의해 입력 변경자 모듈에서 입력 데이터의 절대치연산, 부호변환, 바이패스 동작을 결정한다. 명령어 변경자 항목은 특수 연산 모듈 연산이 완료된 값에 대한 변경을 결정하는 정보이다. 이 값에 의해 실제 연산의 완료된 값이 SFU에서 최종 출력될 때 변경될 수 있으며 모든 결과에 대하여 동일하게 적용 되므로 1비트의 정보로 결정을 하고, 이 항목의 내용에 따라 연산 결과를 그대로 출력하거나 포화시켜 출력하게 된다. Mask 항목은 SFU에서 최종 연산이 완료된 후 레지스터에 저장될 때 사용되는 쓰기 마스크 항목이다. SFU는 모든 입력 요소들에 대하여 동일한 동작을 하는 파이프라인으로 구성되어 있으므로 마스크 항목을 사용하여 연산 유닛의 내부에서 모든 데이터에 대한 연산을 수행한 후 실제 필요한 결과만을 레지스터에 저장할 수 있도록 하였다. 마스크 항목은 4 비트로 구성하였는데 최종 출력되는 데이터는 x, y, z, w 이므로 이 중 마스크 값이 1로 세팅 된 값만이 최종적으로 레지스터에 저장된다. 제어 항목은 SFU 연산에 사용되는 항목은 아니며 상위 제어 부분에서 조건 제어문이나 반복문 등 셰이더 프로세서

Valid (1)	Control (22)	Mask (4)	Instruction Modifier (1)	Input Modifier (8)	Counter (7)	Register type (4)	Swizzle (24)	Compare Condition (3)	SFU op (11)

그림 24. 제어 파이프라인의 정보
Fig. 24. Information of control pipeline.

전체 제어를 위해 사용되는 항목이다. Valid 항목은 SFU 연산에 사용되는 컨트롤 정보와 데이터 그리고 최종 연산 결과의 유효함을 나타내는 항목이다. 이 항목은 다른 제어 부분에서 명령어와 연산에 필요한 데이터의 준비가 완료되어 SFU에 데이터와 제어 정보를 전달할 때 해당 정보가 유효한 정보임을 표시하기 위해 이 항목을 1로 세팅해서 넘겨주며, 연산 결과를 기다리는 레지스터나 이슈 선택 모듈 등에서 판단을 위하여 사용되기도 한다. 제어 파이프라인은 제어 정보가 입력되면 최상위 비트인 유효 항목을 검사하고 이 비트가 1인지를 판단하여 다음 동작이 결정된다.

제어 파이프라인의 제어 정보는 85 비트로 구성하였으나 제어 정보의 유용성을 검사하는 데 있어 valid 항목만을 판별하여 나머지 정보의 사용 유무를 결정하는 것이 효율적이며, 구현에 있어 리셋 신호의 입력 시 유효 비트만을 초기화하는 것이 효율적인 하드웨어의 사용이 가능하므로 유효 비트만이 리셋 기능을 갖도록 구성하였다.

3. 특수 연산 유닛의 명령어 간의 지연

본 논문에서 설계한 SFU에서 수행되는 명령어의 파이프라인 깊이는 연산의 종류마다 차이가 있다. 그러므로 연속적으로 여러 명령어를 수행하기 위해서는 명령어 간의 지연 상관관계에 따라 수행되어야 한다. 명령어 간의 상관관계를 고려하지 않으면 후행 명령어가 선행 명령어보다 앞의 파이프라인으로 삽입될 수 있고, 연산 유닛 내부에서 올바른 동작이 보장되지 않으며 이로 인하여 정확한 연산 결과를 얻을 수 없다.

설계한 SFU의 각 명령어별 파이프라인 깊이에 따른 명령어 간의 지연은 표 2와 같고 명령어간의 지연은 선행 명령어의 깊이가 후행 명령어보다 클 때 현재 수행하고자 하는 명령어는 표에 따라 일정 클럭을 대기한 후 입력되어, 연산 유닛 내부에서 데이터 충돌 없이 올바른 연산이 이루어질 수 있도록 하고 이는 이슈 지연 카운터에 의해 제어된다.

표 2. SFU 명령어 간의 지연
Table 2. Delay between SFU instruction.

앞 \ 뒤	LIT	EXP	POW	LOG	FRC	SINCOS	RSQ	SQRT	RCP	DST
LIT		2	2	33	40	45	46	47	47	47
EXP	0		0	31	38	43	44	45	45	45
POW	0	0		31	38	43	44	45	45	45
LOG	0	0	0		7	12	13	14	14	14
FRC	0	0	0	0		5	6	7	7	7
SINCOS	0	0	0	0	0		1	2	2	2
RSQ	0	0	0	0	0	0		1	1	1
SQRT	0	0	0	0	0	0	0		0	0
RCP	0	0	0	0	0	0	0	0		0
DST	0	0	0	0	0	0	0	0	0	

IV. 설계 검증 및 성능 평가

본 논문에서 제안한 모든 구조는 HDL 언어를 사용하여 설계하였으며, Xilinx사의 ISE9.2를 사용하여 합성한 후, Mentor Graphics사의 ModelSim 6.0 SE를 이용하여 동작 검증을 하였다.

1. 설계 검증

본 논문에서 사용된 FPGA는 Xilinx사의 Virtex4 xc4v1x200을 타겟으로 하여 설계되었다.

구현된 모듈 중 기본적인 부동소수점 연산을 담당하는 부분은 Core Generator의 IP(Intellectual Property)인 Floating Point Operator v3.0을 사용하였다. 설계된 SFU 로직을 Xilinx사의 ISE 9.2를 사용하여 합성한 결과는 표 3과 같다. 전체 모듈의 동작속도는 최대 117MHz의 속도로 수행되었고, 총 7941개의 슬라이스가 사용되는 것을 알 수 있다. 또한 LOG, EXP, SIN/COS 내부에서 사용하는 테이블 때문에 총 128Kbit의 블록램을 사용한다. 개별 명령어의 동작과 여러 명령어에 대한 동작 검증 및 연속된 입력 데이터에 대한 동작도 검증하였다. 기본적으로 SFU에 대한 검증을 위해서 수행한 항목은 다음과 같다.

표 3. SFU의 합성결과
Table 3. Synthesize result of SFU.

세부 모듈	Slices	동작속도
swizzle	384	150 MHz
input modifier	6	180 MHz
special function operator	6255	117 MHz
instruction modifier	592	200 MHz
control pipeline	704	180 MHz
TOTAL	7941	117 MHz

- 개별 및 연속 연산에 대한 기능적 레벨 시뮬레이션
- 각 명령어의 연산 결과 값을 C언어의 Math.h의 함수 결과 값과 정밀도 비교
- 프로그램어블 통합 셰이더 내에서 명령어 읽기 및 결과 값 쓰기 테스트

2. 성능 평가

일반적으로 그래픽 데이터 처리를 위한 기존의 연구는 DirectX 9의 정점/픽셀 셰이더 명령어를 처리하기 위한 그래픽 프로세서에 관한 것들이다. 또한 성능 평가는 대부분 벡터 크기, 프로그램 종류 등 다양한 파라미터에 대하여 그래픽 프로세서를 비교하여 수행되고 있다. 따라서 그래픽 프로세서에서 동작하는 SFU만을 가지고 비교 및 성능을 평가하기에는 무리가 있기 때문에, 본 논문에서는 설계된 SFU가 포함된 프로그램어블 통합 셰이더 프로세서에서 정점 셰이더 프로시저와 픽셀 셰이더 프로시저의 영향을 복합시켜 IPC의 평균에 대한 결과와 여러 프로세서의 성능에 대한 비교를 통해 평가를 수행한다. IPC는 프로시저를 수행하며 실행한 명령어의 수를 소요된 클럭의 수로 나누어 얻을 수 있고, 셰이더 프로세서의 성능 평가의 지표가 되는 가장 핵심적인 항목은 IPC이다. 전반적으로 SFU의 수가 증가함에 따라 IPC 또한 증가한다. 이는 벡터 크기가 작을 때는 전체적으로 명령어 수행의 효율이 낮고, 벡터 크기가 증가함에 따라 명령어의 수행 효율이 좋아지면서 SFU명령어의 비율과 이에 따른 프로세서 구조의 적

표 4. 여러 프로세서의 성능
Table 4. Performance of various processors.

프로세서	동작 주파수 (MHz)	성능 (GFLOPS)	트랜지스터수 (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	856 US
Imagine	150	6.1	NA	6 ALU * 8 Cluster
ATTILA	X	X	NA	시뮬레이션 모델
통합 셰이더 프로세서	277	3.7	1.6	4 ALU, 2 SFU

합성이 성능을 결정하기 때문이다. 따라서 전체 명령의 구성상 SFU 명령어가 차지하는 비율이 높지는 않더라도 셰이더 프로시저의 흐름을 고려하면, SFU의 수가 증가하는 구조가 가장 적절하다.

표 4는 여러 프로세서의 성능에 대한 비교표이다. 이러한 결과를 보았을 때, 본 논문에서 설계된 SFU가 포함된 프로그램어블 통합 셰이더 프로세서는 비슷한 기능을 하는 셰이더 프로세서에 비해 많은 수의 명령어를 지원하며 사용하는 연산 유닛의 수에 비해 전체적인 성능도 우수하다.

V. 결론 및 향후 연구 과제

본 논문에서는 고성능 3차원 컴퓨터 그래픽 영상을 지원하기 위해 방대한 양의 데이터와 복잡한 연산을 효율적으로 처리할 수 있는 SFU를 설계하였다. 설계된 SFU는 DirectX 정점/픽셀 셰이더 3.0에서 제시하고 있는 명령어 중 특수한 연산을 하는 명령어를 하드웨어적으로 직접 수행하여 빠른 그래픽 처리가 가능하다. 또한 각 연산 모듈은 파이프라인으로 구성되어 있어 동일한 연산에 대한 연속적인 수행이 가능하도록 해 전체적인 성능을 향상 시킬 수 있는 구조로 설계되었다.

설계된 SFU는 Xilinx사의 ISE9.2를 사용하여 HDL 합성을 하였고, Virtex-4 xc4v1x200을 타겟으로 합성한 결과 7941개의 슬라이스로 구성되었으며, 전체 연산 유닛이 최대 117 MHz에서 동작한다. 또한 개별 명령어의 동작과 연속된 입력 데이터에 대한 동작과 요구 정밀도의 만족여부를 검증하였다.

향후 그래픽 데이터의 병렬성을 이용하여 처리율의 증가와 설계된 통합 연산 유닛을 여러 개 연결하여 사용함으로써 선형적인 성능향상을 기대할 수 있을 것이다. 그러나 실제 셰이더 처리에서 텍스쳐 명령어에 관한 처리도 필요하며 동작 속도를 높이기 위한 하드웨어 설계 방법에 관한 연구가 이루어져야 한다. 또한 분기 명령어와 반복 명령어 처리에 대한 구현 방법도 차후 중요한 과제가 될 것이다.

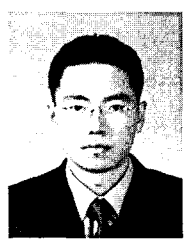
참 고 문 헌

[1] T. A. Moller and E. Haines, "Real-Time Rendering," A.K. Peters, 2002.
[2] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, a

nd Tim Purcell. "A Survey of General-Purpose Computation on Graphics Hardware". Computer Graphics Forum, volume 26, number 1, 2007.
[3] B. Atabek and A. Kimar, "Implementability of Sh ading Models for Current Game Engines," ICCES, pp. 427-432, 2008.
[4] A. Watt, "3D Computer Graphics Third Edition," ADDISON WESLEY, 2000.
[5] 이윤섭, 정진하, 김도형, 최상방, "그래픽 스트림 프로세서의 마이크로 아키텍쳐와 제어부 설계에 관한 연구," 대한전자공학회 하계종합학술대회 논문집II, pp. 571-572, Jul., 2007.
[6] 윤준철, 정진하, 신광식, 김도형, 최상방, "3D 그래픽처리를 위한 ALU 설계," 대한전자공학회 하계종합학술대회 논문집II, pp. 569-570, Jul. 2007.
[7] 김경섭, 장문석, 윤완오, 최상방, "벡터연산 SIMD 그래픽 프로세서를 위한 멀티포트 레지스터 파일 설계," 대한전자공학회 논문지 제 45권 SD편 9호, 2008, pp. 85-95.
[8] U. J. Kapasi, W. K. Dally, S. Rixner, J. D. Owe ns, and B. Khailany, "The Imagine Stream Proce ssor," in proceedings of 2002 IEEE International Conference on Computer Design, 2002.
[9] B. Khailany, "Imagine: Media Processing with St reams," IEEE Micro, Vol. 21, No. 2, pp. 35-46, Mar. 2001.
[10] I. Buck, et al., "Brook for GPUs: Stream Compu ting on Graphics Hardware," in proceedings of ACM SIGGRAPH, 2004.
[11] M. D. Ercegovac and T. Lang, "Digital Arithmet ic," Morgan Kaufmann Publishers - An Imprint of Elsevier Science, 2004.

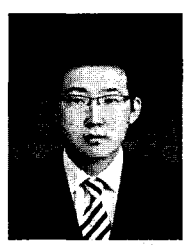
저 자 소 개

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>



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, 차량용 네트워크 시스템>

윤 정 희(학생회원)
1998년 인하대학교 전자계산 공학과 학사 졸업.
2006년 인하대학교 정보컴퓨터 교육학과 석사 졸업.
2008년~현재 인하대학교 전자공학과 박사과정.
<주관심분야 : 컴퓨터 구조, 병렬프로그래밍>



서 장 원(학생회원)
2008년 인하대학교 전자공학과 학사 졸업.
2010년 인하대학교 전자공학과 석사 졸업.
<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-tolera nt computing>