# A Design of a Shader Processor based on a dual-phase pipeline architecture

# 듀얼 페이즈 명령어 파이프라인구조의 쉐이더 프로세서 설계

Hyung-Ki Jeong*, Ki-Hun Nam**, Gwang-Yeob Lee*★

정 형 기*, 남 기 훈**, 이 광 엽*★

*서경대학교 컴퓨터공학과, **한양대학교 디스플레이 공학연구소, ★교신저자

**Abstract**

This paper represents a design of a 4 way SIMD processor with multi-thread and dual phase instruction pipeline. 8 threads can be performing in round-robin order, so any hazards can't occur. The dual phase pipeline makes a pipeline operate as two pipelines, and it can fetch maximum 4 unit instructions at once. This variable length instruction set divide into first phase and second phase instructions, and with this function, complex branch and addressing can be executed at one clock cycle. This processor reduces the code size to quarter, pull out the doubled performance improvement than normal SIMD architecture.

**요 약**

본 논문에서는 멀티 스레드와 듀얼 페이즈 명령어 파이프라인을 가진 4way SIMD 프로세서를 설계하였다. 8개의 스레드가 round-robin 방식으로 실행되어, 해저드를 발생시키지 않는다. 또한 듀얼 페이즈 기능은 1개의 코어가 2개의 프로세서처럼 동작하도록 명령어를 최대 4개를 입력 받아 처리한다. 이 가변 명령어 구조는 1차와 2차 페이즈로 나뉘어 명령어를 수식할 수 있으며, 이 기능을 통해 분기명령이나 어드레싱 명령을 단일 클럭에 수행할 수 있도록 한다. 이 프로세서는 명령어 수행 시간을 일반적인 SIMD 구조에 비하여 50% 이하로 단축시킬 수 있으며, 최대 2배의 성능향상을 보이고 25%까지 코드 크기를 줄일 수 있다..

*Key words : multi-thread, dual phase, variable length instruction, Shader, Graphics*

## Ⅰ. 서론

멀티 스레드가 일반화 되기 전까지는 프로세서의 성능을 높이기 위한 방법으로 칩의 집적도를 높이고 프로세서의 속도를 높여 성능을 향상시키는 방법이나 특정 역할을 하는 전용 프로세서를 설계하여 부족한 성능을 해결하려는 방법이 주를 이루었다. 그러나 직접도의 한계와 단일 프로세서의 고성능화에 따른 아키텍처의 복잡성과 높은 전력소모가 이 방법의 문제점을 말해 주고 있다.

* 본 논문은 지식경제부 "시스템칩 2010 사업"과 학술진흥재단(KRF-2006-005-J04101) 지원에 의하여 작성되었습니다.

또한 현재는 높은 성능의 다양한 프로세서가 개발되고, 저전력에 관심이 집중되고 있는 현실에서 멀티 스레드 프로세싱은 그 중 한 대안으로 나타났다.

멀티 스레드에서 각 스레드의 동작속도는 단일 프로세서에 비해 상대적으로 낮아 하나의 태스크를 수행하는 데에는 비효율적이지만 비슷한 동작으로 처리해야 할 데이터 양이 많을 때에는 매우 높은 성능을 보인다. 또한 하나의 스레드에서 명령어가 실행되고 다음 명령어를 받아들이기까지 딜레이가 존재하는데 단일 프로세서에서는 이러한 딜레이를 없애기 위해 매우 복잡한 과정을 거친다. 그러나 멀티 스레드에서는 연계성이 없는 다른 스레드의 명령어가 대신 실행 됨으로써 프로세서가 단순해 질 수 있고 파이프라인의 연계성을 고려하지 않고 늘릴 수 있기

때문에 동작 주파수도 올리기 쉽다.

이러한 장점은 그래픽 프로세서에서 가장 먼저 도입되었다. nVidia의 최신 그래픽 카드인 8800 시리즈에서는 동일한 프로세서가 여러 개가 존재하고 이 프로세서들은 다시 여러 스레드로 구동된다.

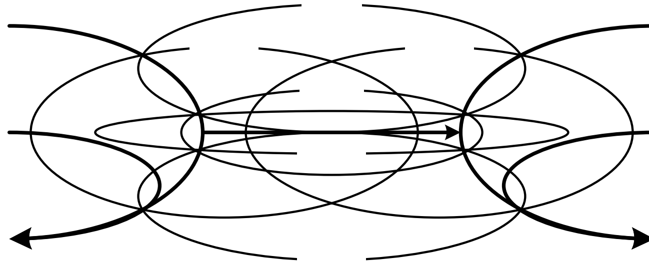

Fig. 1. Concept of processor
그림 1. 프로세서 개념

이러한 다중 스레드들은 프로그램에 따라서 스트림 데이터의 파이프라인 흐름을 재구성 시킨다. 사용자의 목적에 적합하도록 프로세서의 구조를 최적화하여 동작할 수 있으며, 성능 또한 하나 또는 여러 작업으로 집중과 분산시킬 수 있도록 되어있는 구조이다.

이러한 프로세서들은 실제 슈퍼컴퓨터를 대체하는 작업도 가능해지는데, nVidia의 CUDA와 ATI의 CTM은 일반적인 작업을 그래픽 카드에 탑재된 멀티 스레드 프로세서를 통해 분산처리 할 수 있도록 SDK를 공개하고 있다.

본 논문의 프로세서는 일반적인 멀티 스레드 구조를 좀더 효율적으로 구성될 수 있도록 듀얼 페이즈 구조로 프로세서의 연산기 활용을 극대화 시키고 가변길이 명령어 구조로 명령어 레지스터를 차지하는 공간을 최소화 하는 방안을 제시하였다.

## II. 본론

### 1. 아키텍처 기본 구조

그림 1은 본 논문에서 소개하는 멀티 스레드 프로세서의 구조에 대한 기초적인 아키텍처들을 나열한 것이다. 프로세서는 SIMD[1] 구조이며, 하나의 명령어는 2개의 페이즈(Phase)로 나뉘어 입력된다. 각 페이즈는 서로 독립적인 연산을 할 수 있도록 되어 있지만 연산기가 공유되어 있어 명령어가 두 페이즈에서 같은 연산기를 사용하지 않도록 해야 한다. 이는 프로세서의 크기를 작게 하며 연산기 효율을 높이기 위한 방법이다. 각 페이즈는 범용 레지스터(GPRs : General Purpose Registers) 2개의 소스를 입력 받아 1개의 결과를 출력한다. 이 때문에 하나의 스레드의 GPRs는 최대 4개의 읽기 포트와 2개의 쓰기 포트가 필요하게 된다. 이를 위해 각 파이프라인이 홀수 스레드와 짝수 스레드를 번갈아 가며 수행하는 방식으로 GPRs는 2개의 읽기 포트와 1개의 쓰기 포트만 필요하게 된다. 이는 듀얼 포트 SRAM으로 구성될 수 있도록 돕는다.

이러한 듀얼 페이즈 구조는 연산기 활용을 2배로 향상 시키는 것 이외에 페이즈 명령어가 다른 페이즈 명령어를 수식이 가능하도록 하여 명령어를 단순하게 구성하고 수식을 통해 연산, 분기, 메모리 명령어는 보다 복잡한 동작을 가능하게 한다.

이러한 기본 구조를 가지는 4 way SIMD[2] 프로세서는 8개의 멀티 스레드 연산을 수행할 수 있는 듀얼 페이즈 구조이다.

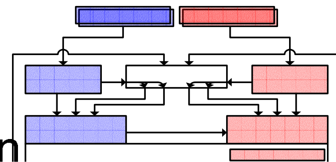


Fig
그림 코어 다이어그램

그림 2는 이 프로세서의 코어의 기본 구조를

나타낸다. 프로세서는 가변길이 명령어로 32비트 유닛 명령어를 최대 4개까지 동시에 입력 받는 가변길이 명령어 구조를 채택하고 있다. 이 유닛 명령어들은 페이즈 0번 또는 1번으로 각각 최대 2개까지 조합 가능하며, 각 페이즈는 유닛 명령어를 받아 GPRs에서 최대 2개의 소스 오퍼렌드를 꺼내어 각각 페이즈에 입력받는다. 연산기를 공유하기 때문에 각 페이즈의 유닛 명령어는 연산기 사용이 겹치지 않도록 해야 하는 배타적 페어링 룰(Exclusive Pairing Rules)이 존재한다. 분기와 메모리 명령어는 페이즈 1에만 존재하는데 그 이유는 분기와 같은 명령어가 여러 페이즈 단에서 동시에 수행될 수 없고 페이즈 1의 유닛 명령어가 페이즈 0과 하위 수식하는 관계이므로 페이즈 1에서 보다 복잡한 명령어 구현이 가능하게 된다. 이는 다양한 분기/연산 구조를 가질 수 있다는 것을 말한다. 이 프로세서의 보다 상세한 코어 아키텍처[3]나 수식과 관련한 가변길이 명령어 구조는[4]는 이전 논문을 참조한다.

코어에 연결되는 레지스터는 명령어 레지스터와 상수를 저장하는 글로벌 레지스터 각 스레드 마다 존재하는 GPRs와 PC 카운터이다. 크리티컬 섹션 마스크는 메모리 또는 다른 스레드와의 동기화에 사용되며, 스크레치 카운터는 스레드간의 스트림 데이터의 중복을 피하기 위해 공용으로 사용하는 스트림 카운터 역할을 하게 된다.

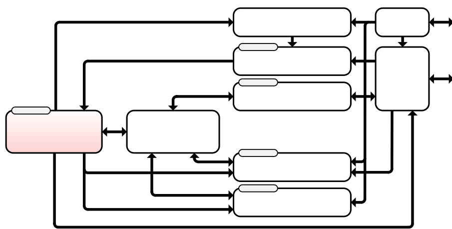

Fig. 3. Registers interconnection
그림 3. 레지스터 연결도

그림 3은 앞서 말한 코어와 레지스터들의 관계를 도식화 하고 있다.

## 2. 파이프라인 설계

이 프로세서는 멀티 스레드를 채용하기 때문에 기본적으로 해저드에 의한 스톨(Stall) 처리를 수행할 필요가 없다. 다만 외부 메모리에 접근에 의한 딜레이가 존재하는데, 이것을 해결하기 위해 크리티컬 섹션 마스크를 두어 해당 비트가 클리어 되었을 때 스레드가 재실행 되도록 할 수 있다.

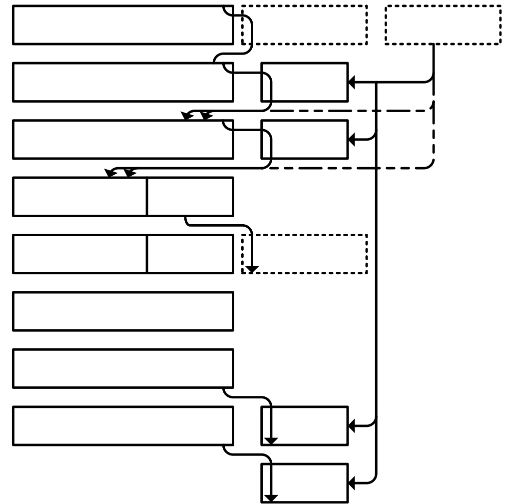

Fig. 4. Process flow diagram of a single thread
그림 4. 단일 스레드의 프로세스 흐름 다이어그램

우선 단일 스레드 관점에서 프로세스 흐름을 살펴보면 그림 4와 같은 단계로 진행된다. 본 프로세서는 8개의 스레드가 라운드로빈 방식으로 실행 되기 때문에 이전에 실행된 명령어 사이클과의 거리가 8 클럭의 여유를 가진다. 때문에 총 파이프라인은 최대 8개를 초과하지 않는 범위 내에서 해저드 없이 여러 단계로 늘려 연산기 성능을 향상할 수 있다. 여기서 스레드가 늘어날수록 파이프라인도 함께 늘릴 수 있지만 그만큼 GPRs를 요구하기 때문에 적정한 선에서 결정해야 한다. 본 논문에서는 비교적 시간이 오래 걸리는 연산기 부분을 3단 파이프라인으로 늘려 성능을 향상 시켰으며, 페이즈 0에서 페이즈 1로의 하위 수식 구조 때문에 2 개의 파이프라인으로 나누어 명령어를 해석하는 구조로 8개의 파이프라인을 가진다. 따라서 스레드를 최소 범위인 8개로 설정한 것이다.

이 프로세서를 설계하는데 가장 큰 문제는 GPRs의 참조가 4번씩 이루어 진다는 것이다. 일반적인 프로세서에 비하여 레지스터 참조가 많은 편이어서 응용에서는 좋은 결과를 가져올 수 있으나 설계에서

있어서 그 복잡도가 높아질 수 있다. 실제 GPRs는 SRAM 구조를 채택한다. SRAM은 최대 듀얼 포트까지 구성될 수 있는데 여기서는 멀티 스레드 구조를 가지기 때문에 GPRs를 참조하는 각 파이프라인이 서로 다른 스레드를 다루고 있어 SRAM 설계가 가능하다. 하지만 8개의 스레드가 각각 GPRs를 가지고 이 출력이 코어에 입력되기 위해 DE-MUX를 통과해야 하기 때문에 시간적 딜레이 문제가 생긴다. 이를 해결하는 방법은 2.2 절에서 자세히 다룬다.

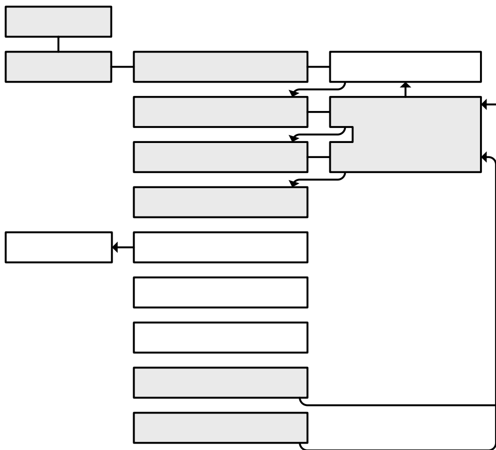

Fig. 5. Pipelines
그림 5. 파이프 라인

그림 5는 실제 설계된 코어의 파이프라인을 나타내고 있다. 총 8단계의 파이프 라인으로 이루어져 8개의 스레드가 실행되기 적합한 구조를 가진다. 각 파이프 라인의 설계에 대한 설명을 다음 단에서 설명한다.

## 2.1 Instruction fetch

명령어는 32비트 유닛 명령어를 최대 4개까지 조합하여 입력된다. 이를 위해서 128비트 명령어 레지스터 모듈이 필요하며, 가변 길이 명령어를 구현하기 위해 4개의 32비트 레지스터 뱅크들을 조합하여 입력 받는 PC값에 따라 적절한 명령어를 코어에 전달해야 한다. 이 때 O/S 드라이버에서는 32비트의 단일 레지스터로 접근이 가능하도록 설계하거나 64비트나 128비트로 버스폭에 맞추어 burst 전송하도록 설계할 수 있다.

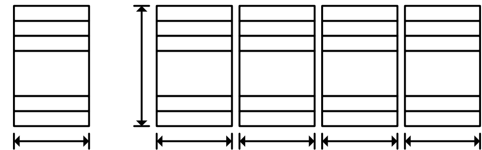

Fig. 6. Instruction registers organization
그림 6. 명령어 레지스터의 조합

그림 6은 4개의 명령어 레지스터 뱅크를 두어 설계한 것이다. 이 명령어를 올바르게 패치 하기 위해 그림 7과 같이 입력되는 PC 값에 대하여 각 뱅크가 적절한 위치의 명령어를 출력하는 것을 볼 수 있다.

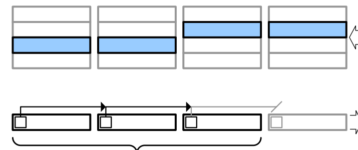

Fig. 7. Instruction Fetch
그림 7. 명령어 패치

## 2.2 Instruction decode

4개의 유닛 명령어가 동시에 코어에 입력되면 엔드 비트를 체크하여 실제 몇 개의 유닛 명령어가 사용되지는 판단하고 다음 명령어의 PC값을 계산한다. 또한 유닛 명령어의 페이즈 번호에 따라 해당 페이즈에 패치하며, 페이즈 0일 경우 decode



2개의 소스 오퍼렌드를 입력 받는데 이는 GPRs에서 decode #0, #1에서 각각 최대 2개의 소스를 입력 받아야 하는 것과 같으며 결과적으로 각 GPRs는 2개의 출력 포트와 1개의 입력 포트가 필요하다.

이를 위해서 그림 8처럼 GPRs 레지스터를 DE-MUX를 통하여 코어와 연결하는 방법이 있다.

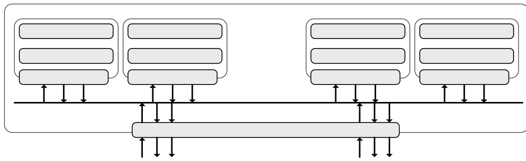

Fig. 8. Conventional GPRs module
그림 8. 일반적인 GPRs 모듈

이 방법은 많은 메모리 드라이브와 MUX에 의한 딜레이가 발생하기 쉽다. 실제 FPGA 합성툴에서 SRAM을 제외하고 슬라이스를 15%를 차지했으며 7.5ns의 post delay가 발생했다.(xc2vp30 기준) 속도를 빠르게 하기 위해 SRAM을 사용하지 않고 레지스터로 변환할 경우 속도는 2.5ns로 안정적이지만 슬라이스 면적의 94%를 차지할 만큼 매우 커서 실용적이지 못했다.

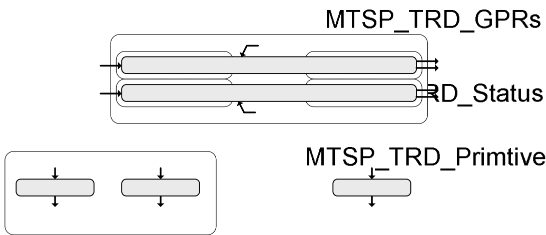


Fig. 9. Optimized GPRs module
그림 9. 최적화된 GPRs 모듈

이를 해결하기 위해서 그림 9와 같이 레지스터를 홀수 열과 짝수 열로 나누어 처리하고 2:1 MUX를 통하여 데이터가 이동하도록 한다. 그 결과 약 2% 면적을 차지하면서 속도는 2.15ns post delay로 그림 10과 같이 만족스런 결과가 나왔다.

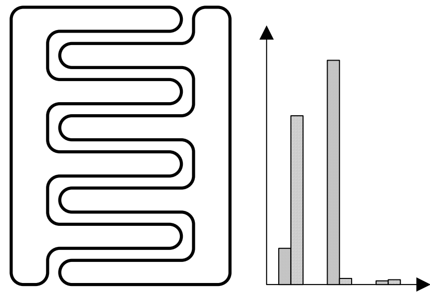

Fig. 10. Comparison of GPRs methods
그림 10. GPRs 방식의 비교

## 2.3 Source modifier

Source modifier는 4 way SIMD 구조에 맞게 각 x, y, z, w 컴퍼넌트의 위치를 변경하거나 또는 부호를 반전시키는 일반적인 내용 이외에 소스 오버라이드 기능을 가진다.

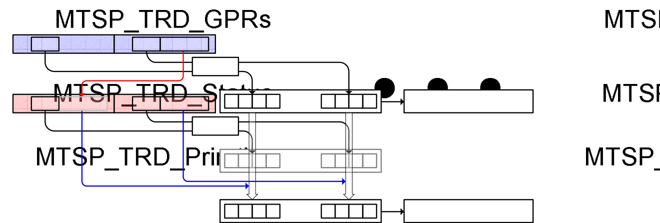


Fig. 11. Source override
그림 11. 소스 오버라이드

소스 오버라이드는 그림 11과 같이 각 페이즈 명령어는 최대 2개의 소스와 1개의 소스 마스크를 각각 가지는데 이 소스 오버라이드는 소스 마스크를 이용하여 페이즈 1번의 소스가 페이즈 0번의 소스로 대체되도록 설정할 수 있다. 이것은 페이즈 1번의 명령어가 더 많은 소스를 참조할 수 있도록 수식하여 최대 4개의 소스를 페이즈 1에서 지정할 수 있게 한다. 이 기능은 참조할 데이터가 여러 레지스터에 골고루 분산되어 있을 때 한 번의 명령어로 처리할 수 있도록 하기 위해 사용된다.

## 2.4 Execution

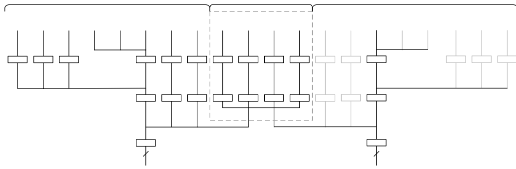

Fig. 12. Execution module
그림 12. 실행 모듈

실행 모듈에 있는 연산기들은 각 x,y,z,w 컴퍼넌트마다 독립적으로 한 개씩 존재하며 이를 페이즈 0과 1에서 공유하여 사용한다. 따라서 각 페이즈는 동일한 컴퍼넌트에 대하여 동일한 연산을 수행할 수 없고 서로 다른 연산에 대해서는 동시에 수행될 수 있다. 그림 12의 옅은 색 부분의 연산기는 페이즈 1이 페이즈 0의 연산기와 공유함을 나타내며, 역수기와 역제곱근기와 같은 특수 함수들은 각 컴퍼넌트에 존재하지 않고 코어 내에 오직 하나씩만 존재한다. 이러한 명령어의 조합과 연산기 공유를 피하는 것은 모두 컴파일러의 부담으로 작용된다. 하지만 연산기의 활용성, [illegible] 명령어 조합으로 성능을 향상시킬 수 있고 코어는 보다 단순화 시킬 수 있다. 실행 모듈에서는 한 사이클에 최대 8개의 연산기가 동시에 수행될 수 있으며 연산 결과를 2개까지 낼 수 있다.

## 2.5 Write back

Write back 단에서는 연산 결과를 GPRs에 다시 전달하는 역할을 하며 동시에 Post-modifier의 역할을 하기 때문에 명령어에 saturate postfix가 붙을 경우 결과 값을 0~1 사이의 값으로 클리핑 되어 저장된다.

## 3. 응용 프로그램에 의한 프로세서 검증

프로세서를 검증하기 위해 32비트 PowerPC405가 탑재된 XUP2VP 보드를 사용하였다. 512MB의 DDR 메모리와 PLB(Processor Local Bus)와 DCR(Device Control Registers) 버스를 이용하여 본 프로세서를 검증하였다.
그림 13은 코어 1개를 이용하여 프로세서를 구성한 모듈을 나타낸 것이다.

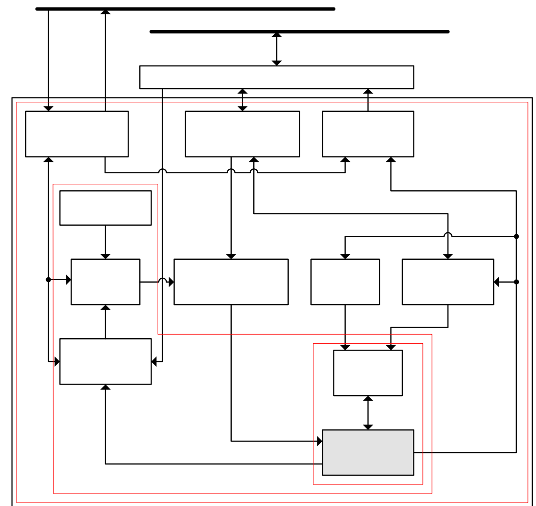

Fig. 13. Verification environment diagram
그림 13. 검증 환경 다이어그램

예제로 사용된 어플리케이션은 Vertex Shader[5]를 이용한 조사광 [illegible] 계산과 원근 투영 매트릭스 변환을 테스트하였다. 그림 14는 사용자가 입력하는 셰이딩 프로그램이 여러 스레드에 걸쳐 실행 될 수 있도록 분산 프로그래밍 환경의 드라이버 프로그램의 구조를 나타낸다.

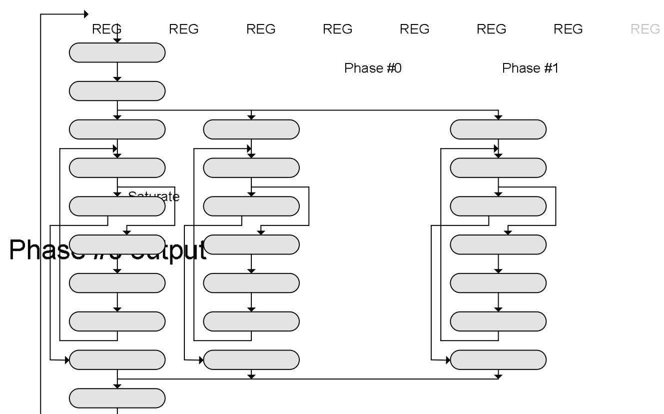


Fig. 14. Application flow diagram
그림 14. 어플리케이션 흐름도

이것은 다중의 스트림 데이터를 처리하기 위한 한 방법으로 일반적으로 호스트 CPU에서 임의의 한

스레드에 실행 명령을 내리면 호스트 CPU에서 정한 실행 스레드 개수만큼 프로그램을 확장하며 각 스레드는 서로 데이터를 겹치게 실행하지 않도록 스크레치 카운터를 참조하여 데이터를 처리한 후 더 이상 처리할 데이터가 존재하지 않을 때 종료되며, 마지막에 종료되는 스레드는 호스트 CPU에게 다음 작업을 수행할 수 있음을 알린다. 이러한 프로그램 구조는 사용자 프로그램이 실행되기 위한 환경을 구성하는 헤더코드(header code), 실제 사용자에게 입력 받은 유저코드(user code), 유저코드로부터 계산된 데이터를 메모리에 저장하고 새로운 스레드가 실행 가능한지 조사하는 풋터코드(footer code)로 나뉜다.

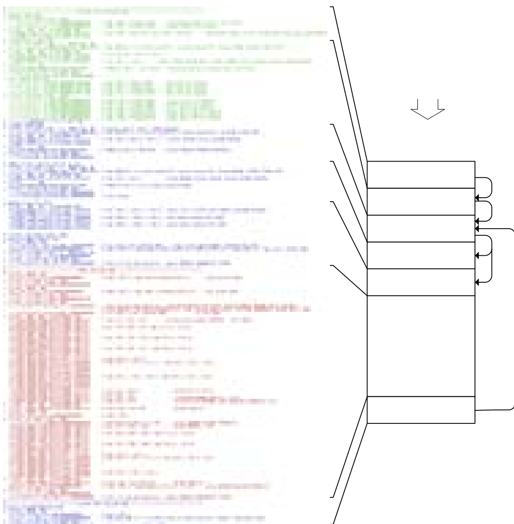

Fig. 15. Distributed program
그림 15. 분산 프로그램

그림 15는 실제 구동할 사용자 프로그램 이외에 이를 각 스레드가 정상적으로 실행할 수 있도록 관리하는 코드가 덧붙여 수행되어야 함을 보인다.

사용자 프로그램에는 가장 단순한 직사광 연산과 Projection matrix 연산 divide by W를 수행하며 17사이클이 소요된다. 여기에 반복적인 드라이버 코드 14사이클과 합쳐 총 31사이클이 소요되는 셈이다. 코어는 100MHz로 동작하고, 이상적일 때 3.2M Vertex를 계산한다는 결론이 나오며 실제 동작에서는 이보다 더 성능이 낮아진다. 하나의 코어만으로 높은 성능과 효율을 차지하기란 어렵다. 하지만 프로그래머블에 의한 유연성으로 멀티 코어로 병렬화가 가능하기 때문에 성능 비교는 무의미하며 이보다는 효율성이 중요시 된다.

## 4. 성능

Table 1. Design results of a core
표 1. 코어의 디자인 결과

표 1은 본 논문에서 설계된 프로세서의 코어 하나에 대한 결과이다. 약 13만 게이트 크기를 가지고 100MHz 동작이 가능하며, 드라이버 코드를 포함하는 Geometry와 Raster 부문에서 3.2MVertices/s와 44MPixels/s의 성능을 가진다.

효율적인 면에서는 코어당 약 13만 게이트의 비교적 적은 사이즈를 차지하면서 멀티 스레드와 듀얼 페이즈 기능을 조합하며 이에 맞는 가변 길이 명령어 체계를 확립함으로써 보다 높은 효율을 낼 수 있다. 그림 16은 연산기 수가 동일한 조건에서 최대 2개의 연산기를 한 명령어로 동시에 사용 가능하고 128비트의 VLIW 명령어 구조를 가지는 싱글 스레드 프로세서와 멀티 스레드 구조이면서 한 명령어가 하나의 연산을 수행 가능한 64비트 VLIW 프로세서를 비교하되 별도의 메모리 스톨이 없는 이상적인 상황에서 비교한 것이다. 이 성능 지표를 비교해 볼 때 파이프라인 간의 데이터 의존성이 존재하는 단일 스레드 프로세서보다 최대 2배의 성능차이를 보이고, 다양한 연산기를 조합하여 사용할 수 있는 가변길이 명령어 구조로 일반적인 멀티 스레드 구조보다 성능 효율이 높거나 같다.

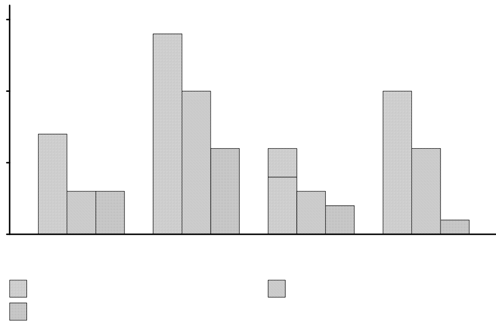


Fig. 16. Comparison of execution cycles
그림 16. 동작 사이클 수 비교

특히 분기 명령어의 경우는 멀티 스레드의 기본 특성인 해저드가 없으면서, 페이즈 간의 명령어 수식과 동시에 연산이 가능하기 때문에 스택의 push/pop이 필요한 nested call/return과 같은 복잡한 분기 명령어를 별도의 스택 관리 모듈 없이도 단일 사이클에 수행 할 수 있다. 이러한 연산과 분기 명령어에서의 장점은 프로세서를 적은 연산기 자원으로 높은 효율을 가지게 할 수 있다.

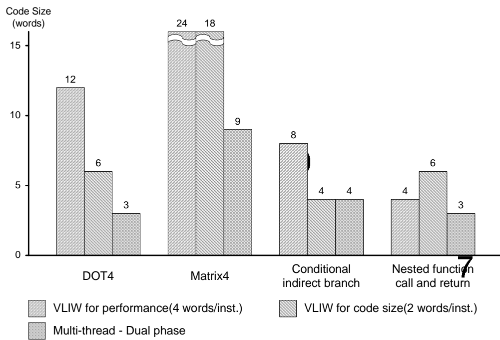


Fig. 17. Comparison of code size
그림 17. 코드 크기 비교

실행 사이클 이외에도 명령어 코드 사이즈에서도 이득을 보게 되는데 그림 17은 그림 16과 동일한 조건에서 2종류의 연산과 분기를 동시에 실행할 수 있는 성능 위주의 128비트 VLIW 구조와 단일 연산 또는 분기를 수행할 수 있는 코드 크기에 특성화된 64비트 VLIW 구조와 비교를 나타낸다. 성능위주의 VLIW 구조에 비하여 최대 25%까지 코드 사이즈를 감소시킬 수 있으며 이로 인해 명령어 레지스터를 더 효율적으로 사용할 수 있다.

## Ⅱ. 결론

본 논문에서는 그래픽 쉐이더 프로세서 설계에서 멀티 스레드의 장점인 구조의 단순화와 듀얼 페이즈 구조로 명령어가 효율적으로 구성될 수 있도록 하여 프로세서의 성능을 향상시켰다. 설계된 그래픽 쉐이더 프로세서는 성능효율의 향상과 칩 면적의 최소화에 중점을 두었다. 명령어 배열의 최적화를 위하여 컴파일러의 복잡도가 증가하는 단점이 있으나 그 성능 상의 이점이 일반적인 프로세서에 비해 높고 하드웨어가 단순하기 때문에 그래픽 처리와 같은 고성능 프로세서로 적합한 것으로 보인다.

## 참고문헌


[1] 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.* 2003

[2] Hajime Kubosawa, Naoshi Higaki and Hiromasa Takahashi, "Four-way VLIW Geometry Processor for 3D Graphics Applications", *FUJITSU Sci. Tech. J.,36, 1*, June 2000

[3] Hyung-Ki Jeong, Kwang-Yeob Lee and Jae-Chang Kwak, "A Multi-thread Processor Architecture with Dual Phase Variable-Length Instructions", *The 23rd International Technical Conference on Circuits/Systems, Computers and Communications*, pp209-212, 2008

[4] Hyung-Ki Jeong, Jea-Chang Kwak and Kwang-Yeob Lee, "A Multi-thread Stream Processor with Variable-Length Instructions for High Performance 2D/3D Graphics", *The 22nd International Technical Conference on Circuits/Systems, Computers and Communications*, pp794-750, 2007

[5] James C. Lelterman, "Lean Vertex and Pixel Shader Programming with DirectX9" *Wordware Publishing Inc.*




## 저 자 소 개

**정 형 기** (학생회원)

2005년 : 서경대학교 컴퓨터공학과 졸업 (공학학사)
2007년 : 서경대학교 대학원 컴퓨터공학과 (공학석사)
2007년~현재 : 서경대학교 컴퓨터 공학과 공 학박사과정
<주관심분야> 마이크로 프로세서, 멀티 스레드 프로세서, 3D Graphics System

**남 기 훈** (정회원)

2006년 : 서경대학교 컴퓨터과학학과 졸업(이학박사)
2006년~현재 : 한양대학교 디스플레이 공학연구소 전임연구원
<주관심분야> 마이크로 프로세서, 3D Graphics System, Display관련

**이 광 엽** (정회원)

1985년 : 서강대학교 전자공학과 졸업 (공학학사)
1987년 : 연세대학교 대학원 전자공학과 (공학석사)
1994년 : 연세대학교 대학원 전자공학과 (공학박사)
1989~1995년 : 현대전자 선임연구원
1995년~현재 : 서경대학교 컴퓨터공학부 부교수
<주관심분야> 마이크로 프로세서, Embedded System, 3D Graphics System