論文97-34C-9-2

다중스트리밍을 이용한 3차원 그래픽 프로세서 구조

(3D graphics processor architecture based on multistreaming)

朴 鏞 珍 *, 李 東 浩 *

(Yong-jin Park and Dong-ho Lee)

요 약

본 논문에서는 3차원 그래픽용 명령어 프로세서의 구조로 다중 명령 수행 다중 스트리밍을 제안한다. 다중스트리밍은 파이프라인 내에서 동시 수행되는 명령어 사이의 상호 간섭을 제거하고, 병렬 처리를 위한 충분한 병렬성을 제공한다. 사이클 수준 모의 실험을 통하여 제안한 프로세서 설계가 적당한 범위의 자원 추가로 전통적인 RISC 프로세서인 MIPS R3000에 비해 3 배 이상의 성능을 가짐을 보였다. 다중 명령 수행 다중스트리밍 프로세서는 대량의 쓰레드 공급이 보장되는 상황에서 좋은 명령어 프로세서 설계가 될 것이다.

Abstract

In this paper, we propose multiple instruction issuable multi- streaming as a processor architecture for 3D graphics processor. Multistreaming can eliminate interferences within concurrently executing instructions in the pipelined processor to allow enough parallelism for parallel processing. Through cycle level simulation study, we show that the proposed architecture outperforms a conventional RISC processor, MIPS R3000 by three times with reasonable resource overheads. Multiple instruction issuable multistreaming processor will be a good architecture for instruction processor when a large number of threads are guaranteed.

I. 서 론

3차원 그래픽은 공간 내의 물체와 광학 현상을 모형화하여 장면의 입체적 시각 효과를 출력 장치에 표현하는 기술이다. 3차원 그래픽 처리 과정은 그림의 구성 요소 또는 화면을 구획으로 나누어 병렬처리가 가능하다[1]. 따라서, 서로 독립적인 명령어 실행 흐름인 쓰레드(thread)를 대량으로 생성하고 이를 단위로 병렬처리가 가능하다. 이러한 특성을 고속의 3차원 그래픽을 효율적으로 실현하는 응용 분야 전용 명령어 프로세서(Application Specified Instruction Processor ; ASIP)의 설계에 이용할 수 있다. 기존의 3차원 그래픽을 위한 병렬형 프로세서 설계에는 전용 처리 장치 형태로의 시스토릭 어레이(Systolic Array)[2], 다중 컴퓨터형의 병렬 처리기 등의 방법들이 이용되고 있다. 전용 처리 장치의 경우에 다양한 그래픽 알고리즘의 사용이 어렵고, 다중 컴퓨터 형태의 병렬 처리기는 처리 요소 각각에 포함되는 자원의 공유가 어려워 하드웨어 활용도가 떨어질 수 있다. 단일 프로세서로 구성된 처리기의 경우 슈퍼스칼라(super-scalar)나 LIW형태의 ILP 프로세서를 이용하면 쓰레드 병렬성의 이용에 한계가 있다. 이러한 점들을 고려하여 한 프로세서가 여러 쓰레드의 명령을 선택적으로 수행하게 하면 동시 복수 명령의 수행이 가

* 正會員, 慶北大學校 電子工學科
(School of Electronics and Electrical Engineering KyungPook National University)

接受日字:1997年1月9日, 수정완료일:1997年8月20日

능하게 되고, 각 명령어들의 실행 잠복기에 의한 프로세서의 실행 속도 저하를 제거할 수 있다. 이렇게 하나의 프로세서가 여러 쓰레드를 동시 수행(concurrent execution)하는 명령어 처리 방식을 다중스트리밍(Multistreaming)이라 한다. 다중스트리밍은 파이프라인에 각 사이클마다 독립적인 명령어 흐름을 공급하여 각 명령의 실행 잠복기로 인한 파이프라인 장애(pipeline hazard)를 프로세서 성능의 저하 없이 제거하여 자원 활용도를 높인다. 따라서, 다중스트리밍은 슈퍼스칼라 프로세서에서 사용되는 명령간 의존도 검출과 분기 예측 등을 위한 복잡한 하드웨어 지원 없이 다중 명령어의 효율적인 실행이 가능하게 한다. 다중스트리밍은 다중쓰레딩의 단일 프로세서에서의 적용 형태라고 할 수 있다. 다중프로세서로 구성된 병렬 컴퓨터에서 다중쓰레딩에 의한 자원 이용 잠복기 효과의 제거는 다중쓰레딩 병렬 컴퓨터들에서 이용되고 있다[1-3]. McCrackin은 [4]와 [5]에서 심층 파이프라인 프로세서(deeply pipelined processor)에서 심화되는 파이프라인 장애에 의한 성능 저하의 극복 방법으로 단일 사이클 단일 명령 실행 다중스트리밍 프로세서에서 파이프라인에 공급할 쓰레드를 선택하는 쓰레드 스케쥴링 알고리즘들과 실제 프로세서의 구현을 위한 방안을 제시하고 성능을 평가하였다. Hirata는 [10]에서 다중스트리밍을 광선추적법(ray tracing)의 고속 처리를 위한 병렬 컴퓨터의 처리 요소(processing element) 설계에 적용하고 있다. 이 연구는 다중스트리밍을 광선추적법의 특성에 의한 대량의 메모리 접근에 의한 프로세서의 성능 저하의 극복과 다중명령 실행을 위해 이용하고 있다. 우리는 본 연구에서 다중스트리밍을 단일 칩 형태의 실시간 3차원 쉐이딩용 명령어 프로세서에 적용하기 위해 동시 복수 명령 수행(Parallel execution)이 가능한 다중 명령 수행 다중스트리밍 프로세서의 설계를 제시하고, 설계 요소들의 변화에 따른 프로세서의 성능을 모의실험을 통하여 평가하였다. 본 연구에서 사용된 그래픽 알고리즘은 Gouraud 쉐이딩으로 다각형 표현법에 의한 실시간 고속 그래픽을 위한 것이다. Gouraud 쉐이딩은 광선 추적과 같은 고수준의 화상을 제공하지 못하지만 실시간 처리의 구현이 쉽다. 광선 추적에 비해 자료 메모리 접근보다 레지스터간의 연산이 실행속도에 더 큰 영향을 미치므로 파이프라인의 동작 속도가 중요하다. 따라서, 본 연구에서는 프로세서 내의 연산 장치를 파이프라인으로 구성하고 각 쓰레드별로 비교적 큰 레지스터 파일을 제공하여 연산 속도를 높이고 있다. 그리고, 여러 개의 실행 파이프라인을 마련하여 다중 명령의 실행이 가능하도록 하였다. 다중 명령의 실행에서 나타나는 비순차 실행에 의한 명령어간의 의존성 문제의 해결은 interlocking에 의하지 않고, 컴파일러에 의해서 지연틈이 필요한 곳에 nop 명령을 넣어서 해결하고 있다. interlocking의 제거를 위한 nop의 삽입은 쓰레드 스케쥴링을 위한 지연(delay) 정보로도 쓰이고 있다. 우리는 사이클 수준의 모의 실험을 통하여 이러한 다중스트리밍 프로세서가 많은 연산을 요구하는 쉐이딩과 같은 그래픽의 고속 처리를 위한 적절한 프로세서 형태임을 보인다.

본 논문의 전체적 구성을 보면 II장에서 3차원 그래픽의 특성과 다중스트리밍 프로세서에 대하여 이야기하였다. III장에서 다중스트리밍 프로세서에서 다중 명령의 실행이 가능하도록 하는 쓰레드 스케쥴링 알고리즘을 제안하고 실험에서 사용한 다중 명령의 수행이 가능한 파이프라인 구조를 보였다. IV장에서는 앞의 쓰레드 스케쥴링 알고리즘과 파이프라인 구조를 이용한 프로세서 모형 상에서 사이클 수준 모의 실험으로 Gouraud 쉐이딩을 수행하여 모형의 성능을 평가하였다. 그리고, 프로세서 연산 자원들이 첨가될 때의 프로세서 성능 변화를 측정하였다. V장에서는 전체적인 결론을 내렸다.

II. 그래픽 처리의 특성과 다중스트리밍

1. 3차원 그래픽의 특성

3차원 그래픽은 그 구현에 있어서 다양한 알고리즘이 존재한다. 전체적으로는 MIMD 형태의 병렬성으로 생각할 수 있는데, 공간상의 좌표 변환 같은 SIMD형의 병렬성을 가지는 부분도 있다. 그래픽의 자료 처리 특징은 대량의 자료에 대한 동일 처리 과정을 적용하는 SPMD(Single Program Multiple data stream) 병렬성이다. 예로 다각형 표현에 의한 3차원 그래픽에서 그림은 3차원 공간상의 다각형의 집합, 광원들, 광학효과등으로 이루어진다. 그림을 2차원의 화면상에 나타내는 과정은 각 물체들을 시점에 맞도록 이동, 회전, 변형 등을 행하는 과정(transformation), 3차원을 2차원에 투영하는 과정(projection), 2차원 면에 투영된 다각형을 면의 색과 광원의 효과로부터 계산된 색

으로 채우는 과정(shading)등으로 나눌 수 있다. 이러한 처리를 각 다각형에 독립적으로 적용하여 전체 그림을 구성하는 것이 가능하고, 각 단계는 계산량과 자료 참조량에서 차이가 크다. 그래픽 처리를 다중쓰레드에 의해서 구현하는 방법으로 자료 단위의 숫자만큼의 동일한 전체 처리 쓰레드를 발생시키는 방법과 처리 과정을 단계별로 구분하여 각 단계별 파이프라인된 형태로 적용하여 각 처리 단계를 수행하는 여러 종류의 쓰레드를 만들어서 처리하는 방법 등을 생각할 수 있다. 동일한 전체 처리 과정을 실행하는 쓰레드의 복수 생성에 의한 처리의 경우 대부분의 처리 과정에서 쓰레드간은 독립적으로 수행된다. 쓰레드간의 파이프라인의 경우는 각 단계 사이에 자료 의존성이 있고, 각 단계간에 동기화가 필요하다.

본 논문에서는 다각형 표현에 의한 3차원 그래픽에서 쉐이딩의 가속화를 위한 연구를 수행하였다. 쉐이딩은 동일한 코드를 상이한 다각형들에 적용하게 되는데 이때 실행의 흐름은 다각형에 따라 다르게 된다. 계산되는 값은 각 점의 z, R, G, B, α 들로 각기 다른 계산 루프에 의해서 생성된다. z는 화면상에서 깊이로 나타나는 좌표이고, R, G, B는 점의 광선 강도로 빛의 삼원색 각각의 성분이다. α 는 투명도로 화면상에서 뒤에 존재하는 점의 색이 얼마만큼 영향을 미칠지를 나타낸다. 이 처리는 각 다각형 단위로 수행될 수 있어서 쓰레드 병렬성의 이용이 손쉽게 이루어질 수 있다.

2. 다중스트리밍

현대적인 마이크로 프로세서들은 단위 시간당 실행 명령의 수를 높이기 위해서 높은 클락 사이클을 가지고, 한 사이클에 여러 명령을 실행할 수 있도록 설계되고 있다. 클락 사이클을 높이는 방법은 집적회로 제작 기술의 개선을 통하거나, 프로세서 실행 파이프라인을 세분하여 더 높은 주파수에서 회로가 동작할 수 있도록 하는 방법을 이용한다. 파이프라인을 세분하는 경우에는 명령이 실행 종료되기까지 거쳐야하는 파이프라인의 단계가 증가하게 되는데, 이렇게 되면 한 명령의 결과가 계산되지 않은 시점에서 따라오는 다음 명령이 이를 이용하면 올바른 결과를 쓸 수 없게 된다. 이러한 문제의 해결을 위해서 문제를 일으킬 수 있는 명령어 쌍을 검출하여 둘 사이에 충분한 시간을 두고 명령을 실행시키는 interlocking이나, 명령 사이에 지연 명령을 써서 사이클을 낭비하는 방법들을 사용하는데, 이러한 해결책으로는 성능을 저하시키게 된다. 만약 파이프라인 각 단계에서 실행되는 명령들이 서로 참조 관계가 없는 명령들이라면 사이클의 낭비 없이 인접하여 실행이 가능하게 되는데, 명령들이 서로 다른 명령의 흐름, 즉 서로 다른 쓰레드에서 온 명령들이라면 사이클 낭비가 없는 실행이 가능하다[4]. 이러한, 다른 쓰레드에서 가져온 명령들의 선택적 실행에 의한 파이프라인 프로세서 성능의 개선은 다중 프로세서 병렬기의 일종인 다중쓰레딩 프로세서에서 자원 접근에서 잠복기 효과의 상쇄를 위해 이용되고 있다. 쓰레드간의 전환은 프로세서가 각 사이클에 실행에 의존성에 의해서 문제가 발생하지 않는 명령을 여러 쓰레드의 실행 준비중인 명령에서 선택함으로써 이루어진다. 명령이 선택되면 명령의 피연산자로 쓰일 레지스터는 각 쓰레드에 할당된 레지스터 집합 내에서 읽어진다. 결과도 참조된 레지스터 집합 속에 저장된다. 이러한 동작은 프로세서 내부의 동작으로 소프트웨어에 의존하지 않는다. 그래픽은 여러 쓰레드의 생성이 용이하므로 프로세서의 성능을 높게 유지하기에 충분한 수의 쓰레드를 손쉬운 공급이 가능하다. McCrackin은 심층 파이프라인 프로세서에서 쓰레드를 이용하여 사이클 낭비의 최소화를 위한 방법으로 다중스트리밍(multistreaming)을 연구하였다. 이 연구에서는 프로세서가 단일 실행 파이프라인을 가지는 경우로 다중명령 실행에 대한 고려가 없다. 본 논문에서는 계산량이 매우 큰 3차원 그래픽을 위한 고속 명령어 프로세서에 다중스트리밍 프로세서를 적용하기 위해 다중스트리밍과 다중 명령 수행의 동시 적용을 연구하였다. Hirata는 [10]에서 광선 추적 방법(ray tracing)에 의한 3차원 그래픽을 고속 처리하는 병렬 컴퓨터의 각 처리 요소(PE)의 설계에서 다중스트리밍에 의한 프로세서의 성능 향상을 연구하였다. 이 연구의 응용은 실행 시에 대량의 메모리 읽기 접근(Memory Read Access)을 요구한다. 병렬 컴퓨터에서 공유 메모리에 대한 접근은 단일 프로세서 컴퓨터의 메모리 접근과는 그 실행 속도면에서 많은 차이를 가진다. 따라서, 이 연구에서 다중스트리밍은 메모리 접근에서 나타나는 실행잠복기의 영향을 제거하는데 기여하고 있다. 반면, 본 연구에서는 단일 프로세서 컴퓨터 상에서 연산 자원에 대한 동시 요구가 큰 Gouraud 쉐이딩의 실시간 실행을 위해 효율적인 복

수 명령 동시 공급의 방법으로 기여하고 있다. Gouraud 쉐이딩은 광선추적에 비해 메모리 접근을 많이 요구하지 않고, 대부분의 명령은 레지스터간의 연산으로 이루어진다. 따라서, 전체적인 프로세서 성능 향상에는 연산 장치의 효율적 이용과 병렬 처리 성능 향상이 요구된다.

Ⅲ. 다중 명령 실행 다중스트리밍 프로세서

1. 다중스트리밍 프로세서

일반적인 명령어 프로세서의 파이프라인은 명령어 패치 파이프라인(instruction fetch pipeline)과 실행 파이프라인(instruction execution pipeline)의 두 부분으로 나눌 수 있다. 명령어 디코더(decoder)는 패치 파이프라인에 포함된다. 다중스트리밍 프로세서는 일반적인 프로세서와 명령어 패치 파이프라인의 구조와 레지스터 파일의 형태에서 다르다. 다중스트리밍 프로세서는 각 쓰레드의 상태를 저장하기 위한 분리된 레지스터 파일(register file)과 상태 레지스터(status register)를 가지고, 명령어 패치 파이프라인은 쓰레드 스케쥴링 알고리즘을 구현하는 하드웨어를 포함한다.

쓰레드 스케쥴링 알고리즘은 명령의 실행에서 한 쓰레드에서 가져온 명령들 사이의 의존성이 없어지도록 명령 사이에 지연틈(delay slot)을 두어 장애를 제거한다. 이 지연틈에 다른 쓰레드에서 가져온 명령을 수행하여 각 쓰레드들이 파이프라인 장애 없이 수행되도록 한다. 다중스트리밍 프로세서에서 파이프라인 장애가 완전히 제거되기 위해 필요한 최소 쓰레드수인 무장애 쓰레드 수는 파이프라인의 구조에 의해서 결정된다. 무장애 쓰레드 수는 프로세서의 명령어들이 투입에서 연산의 종료까지 요구되는 사이클 수 중 가장 큰 값에 1 더한 수다. 쓰레드의 개수가 부족하면 프로세서의 성능은 저하되게 된다. 이때 성능 저하의 정도는 쓰레드 스케쥴링 알고리즘에 따라 다른 형태로 나타난다[4]. 다중스트리밍 프로세서가 파이프라인 장애에 전혀 영향을 받지 않기 위해서는 무장애 쓰레드 수 이상의 쓰레드를 유지하여야 한다.

2. 다중 명령 수행 다중스트리밍 프로세서

파이프라인 프로세서에서 연속적인 명령의 투입에 의해 파이프라인 장애가 발생할 수 있는 시간 영역은 "프로그램 수행 방향으로 몇 사이클"의 형태로 기술된다. 이 시간 영역을 지연틈(delay slot)이라 한다. 다중스트리밍 프로세서에서 쓰레드 스케쥴링에는 명령별 지연틈에 대한 정보가 이용된다. [5]에서는 지연틈의 길이를 컴파일 시에 계산하고 별도 메모리에 보관해 실행 시에 쓰레드의 스케쥴링에 반영하였다. 그래픽과 같이 전 실행 시간에 걸쳐 충분한 수의 쓰레드가 공급 가능한 경우에는 쓰레드 스케쥴링에 의한 프로세서의 실행 효율 향상보다 병렬성의 활용이 중요하다. 따라서, 단일 사이클에 다중 명령을 실행할 수 있는 다중스트리밍 스케쥴링 방법이 요구된다. 여기에서 우리는 nop(no operation)명령을 지연이 필요한 곳에 넣어 지연틈을 나타내고 이를 바탕으로 쓰레드를 스케쥴링하는 방법을 제시한다. nop으로 주어진 지연 정보를 바탕으로 프로그램을 실행하기 위해서는 명령들을 실행 전에 미리 프로세서 내부로 읽어 들여 각 명령들의 정보를 분석하여야 한다. 따라서, 프로세서 내에 미리 읽어진 명령을 저장하기 위한 명령어 풀이 필요하다. nop 명령에 의한 지연틈 표현의 예를 그림 1에 보였다. 일부 nop은 코드 재구성(code reorganization)을 통해 유효명령으로 대치될 수 있다[6].

```
0008    .
000c    ld      $Dest, 100($Target)     delay 3
0010    nop
0014    nop
0018    nop
001c    addi    $Dest, $Dest, 0xf0      delay 0
0020    sw      100($Target), $Dest     delay 2
0024    nop
0028    nop
.
.
```

그림 1. nop 명령 삽입에 의한 지연틈의 표시
Fig. 1. Description of delay slot with nop insertion.

(1) 명령어 풀에 의한 쓰레드 스케쥴링

[4]와 [5]의 쓰레드 스케쥴링 알고리즘들은 한 사이클에 한 개의 명령을 패치하고 실행한다. 본 논문에서 사용하는 스케쥴링 방법은 nop 명령에 의한 지연틈의 표현과 명령어 풀을 사용하여 복수 명령을 단일 사이클에 실행한다. nop 명령에 의한 지연틈의 표현에서 nop은 "이 자리에 다른 쓰레드가 스케쥴링되어야 된다"는 의미를 가진다. 프로세서는 현재 명령어 풀에 nop을 가지고 있지 않은 다른 쓰레드로부터 명

령을 패치하여 투입한다. 그림 2, 3, 4, 5는 5 개의 쓰레드를 유지하는 다중스트리밍 프로세서가 프로그램을 수행하는 과정을 보이고 있다. 첫 번째 열이 명령어 풀의 내용이 된다. active는 명령이 투입 가능함을 나타내고, nop은 지연틈을 나타낸다. 그림 2, 3, 4, 5에서 각 쓰레드의 수직 배열은 프로그램이 실행될 때의 프로세서가 처리하는 쓰레드별 명령 흐름이다. 실행하는 모든 쓰레드의 현재 PC(Program Counter)에 위치한 명령을 단일 사이클 내에 동시에 패치할 수 있는 이상적인 프로세서를 가정한다.

inst. flow	Thread #1	Thread #2	Thread #3	Thread #4	Thread #5
↓	active 1	active 1	active 1	active 1	active 1
	active 2	nop	nop	active 2	active 2
	nop	active 2	nop	active 3	nop
	nop	active 3	active 2	nop	nop
	nop	active 4	nop	nop	active 3
	nop	active 5	active 3	nop	active 4

그림 2. 쓰레드별 명령 흐름 1
Fig. 2. Thread-wise instruction flow 1.

그림 2에서 프로세서가 동작 시작하여 첫 번째 사이클에 들어오면, 모든 쓰레드의 명령들이 유효 명령으로 투입 가능하다. 따라서, 프로세서는 이들 중 하나의 쓰레드를 선택하여 실행 파이프라인에 투입한다. 여기서, 1번 쓰레드를 선택했다면 그림 2는 그림 3과 같이 바뀐다.

inst. flow	Thread #1	Thread #2	Thread #3	Thread #4	Thread #5
↓	active 2	active 1	active 1	active 1	active 1
	nop	nop	nop	active 2	active 2
	nop	active 2	nop	active 3	nop
	nop	active 3	active 2	nop	nop
	nop	active 4	nop	nop	active 3
	nop	active 5	active 3	nop	active 4

그림 3. 쓰레드별 명령 흐름 2
Fig. 3. Thread-wise instruction flow 2.

여기에서 쓰레드 2가 선택되면 그림 3은 그림 4와 같이 변하게 된다

여기에서 쓰레드 3이 선택되게 되면 그림 4는 그림 5와 같이 바뀐다. 여기에서 쓰레드 2의 nop은 쓰레드 3의 첫 번째 active가 투입되면서 명령 배열에서 삭제된다. 이는 쓰레드 3이 투입되는 사이클은 쓰레드 2에 대해서는 지연틈으로 작용하기 때문이다.

inst. flow	Thread #1	Thread #2	Thread #3	Thread #4	Thread #5
↓	active 2	nop	active 1	active 1	active 1
	nop	active 2	nop	active 2	active 2
	nop	active 3	nop	active 3	nop
	nop	active 4	active 2	nop	nop
	nop	active 5	nop	nop	active 3
	nop	nop	active 3	nop	active 4

그림 4. 쓰레드별 명령 흐름 3
Fig. 4. Thread-wise instruction flow 3.

inst. flow	Thread #1	Thread #2	Thread #3	Thread #4	Thread #5
↓	active 2	active 2	nop	active 1	active 1
	nop	active 3	nop	active 2	active 2
	nop	active 4	active 2	active 3	nop
	nop	active 5	nop	nop	nop
	nop	nop	active 3	nop	active 3
	nop	nop	active 4	nop	active 4

그림 5. 쓰레드별 명령 흐름 4
Fig. 5. Thread-wise instruction flow 4.

(1) 모든 쓰레드의 현재 PC가 지시하는 위치의 명령들을 패치하여 디코드하여 명령어 풀에 넣는다. 이는 초기화 단계이다.

(2) 명령어 풀 내의 모든 nop 명령을 제거한다. 명령어 풀에서 유효 명령을 담고 있는 쓰레드를 선택하여 그 명령을 실행 파이프라인으로 투입하고 (4)로 진행한다. 명령어 풀에 유효 명령이 없는 경우에 (3)으로 진행한다.

(3) nop 명령을 실행 파이프라인에 투입하고, (1)로 진행한다.

(4) (2)에서 선정된 쓰레드와 nop이 제거된 쓰레드의 다음 명령을 패치하고 디코드하여, 명령어 풀의 빈자리를 채운다. (2)로 진행한다.

그림 6. 쓰레드 스케줄링 알고리즘
Fig. 6. Thread scheduling algorithm.

어떤 쓰레드에도 유효 명령이 존재하지 않은 경우에는 풀을 비우고, 파이프라인에 nop 명령을 투입한다. 각 쓰레드별로 명령이 소비되면 다음 명령을 패치하여 풀을 채운다. 이와 같은 방법으로 명령어 풀에 의한 쓰레드 스케줄링을 수행할 수 있다. 이 스케줄링 알고리즘을 정리하면 그림 6과 같다. 여기에서, 명령어 풀은

쓰레드별로 스케쥴링에 쓰일 명령들의 연산코드(op-code)와 피연산자들(arguments)을 담고 있는 레지스터 배열이다. 큰 명령어 패치 대역폭에 대한 요구는 다음에 보이는 명령어 풀의 큐에 의한 구성으로 보상된다.

(2) 명령어 풀의 설계

넓은 패치 대역폭에 대한 요구는 명령어 풀을 큐(queue)로 구성하여 해결할 수 있다. 쓰레드 스케쥴링에는 쓰레드들의 PC가 지시하는 위치의 모든 명령이 필요하지만 선택되는 쓰레드는 일부다. 따라서, 초기에 명령어 풀을 채우면 프로세서가 동작하면서 명령어 풀에서 명령들을 선택적으로 소비하고, 명령이 소비된 큐는 패치 파이프라인으로부터 명령을 공급받아서 적정한 수의 명령을 유지하게 된다. 따라서, 명령어 패치 대역폭에 대한 요구는 완화된다. 패치 대역폭은 큐로 이루어진 명령어 풀이 바닥나지 않을 정도가 되어야 한다. 명령어 풀의 구조는 그림 7과 같다. 패치된 명령은 오른쪽 끝으로 공급되고, 명령이 소비됨에 따라 왼쪽으로 이동하여 명령어 풀을 빠져나간다.

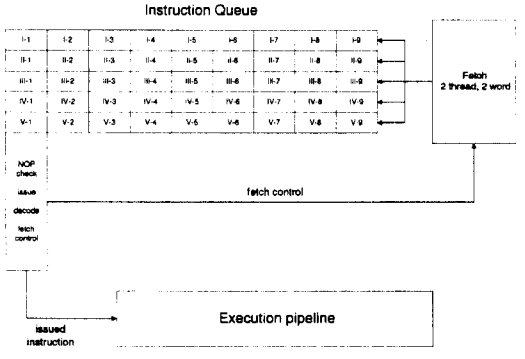

그림 7. 명령어 풀의 구조
Fig. 7. Structure of instruction pool.

풀의 동작에는 명령어 메모리로부터 적당한 수의 명령을 동시에 패치하여 풀에 공급하기 위한 명령 패치 장치와 풀에서 명령을 선택하여 실행 파이프라인에 공급하는 장치가 필요하다. 명령 선택장치는 명령어 패치 장치가 어느 쓰레드를 우선적으로 패치해야 하는지를 지시한다. 이렇게 설계된 명령어 풀을 사용하게 되면 그림 6에서 제시된 쓰레드 스케쥴링 알고리즘을 실제 프로세서에 적용할 수 있다. 명령어 소비율의 최대값과 명령어 패치 대역폭이 명령어 풀을 구성하는 큐의 깊이를 결정한다. 명령어 풀의 레지스터는 프로세서의 물리적인 면적에 영향을 주는 요소로 무한히 크게 할당할 수 없다. 명령어 풀의 폭은 프로세서가 유지하는 쓰레드 수의 최대 값으로 무장애 쓰레드 수 이상으로 유지되어야 프로세서가 최고 성능을 가질 수 있다. 명령어 풀의 깊이는 최고 명령소비율로 명령이 소비되어도 명령어 풀에 충분한 수의 명령이 존재하도록 충분히 깊게 구성되어야 한다. 따라서, 무장애 쓰레드 수가 적도록 설계해야 실제 프로세서를 실현하기 용이하다. 지연틈의 길이를 표현하기 위해 nop을 지연틈 길이 만큼 나열하지 않고 지연틈에 대한 정보를 표시하는 명령을 따로 두어서 이 명령 내에 길이에 대한 정보를 포함시켜서 지연틈을 표시하면 메모리 활용도와 패치 대역폭에서 유리하게 된다.

[4]와 [5]에서 실험된 스케쥴링 방법들은 프로그램의 실행을 고속화하기 위해 실행 파이프라인에 다수의 자원을 두고 여러 명령을 동시 실행하는 경우에 적용이 힘들다. [5]에서 쓰인 방식은 다수의 실행 장치가 쓰이는 경우에 각 명령 별로 소요 자원에 대한 구별이 필요하기 때문에 지연 필드가 길어져 메모리 워드가 넓어지고, 많은 메모리를 요구하게 된다. 넓은 메모리 워드는 캐쉬의 워드도 넓어져야 함을 의미한다. 그리고, [5]에서 쓰이는 쓰레드 선택 후 패치 방식은 최악의 경우에 프로세서 내부 자원수 만큼의 명령어를 한 번에 패치해야 한다. 따라서, 많은 수의 쓰레드의 명령을 한 번에 패치해야 하는 경우가 발생한다. 한 번에 많은 쓰레드에서 여러 명령을 패치하는 장치는 실현이 어렵다.

(3) 파이프라인 구조

다중 명령 수행 다중스트리밍 프로세서에서 파이프라인 구조는 무장애 쓰레드 수를 결정하는 요소다. 프로세서의 명령을 자원 사용에 따라 분류하면 일반 연산 명령, 메모리 접근 명령, 다중 실행 사이클 명령 등의 세 가지가 있다. 실행 파이프라인을 명령들이 종류에 따라 서로 다른 파이프라인 경로를 통해 처리되도록 분리 구성하면 명령 수행에 쓰이지 않는 자원을 경유하지 않고 실행되어 실행 잠복기간이 한 두 사이클씩 줄어들게 되고, 여러 명령을 동시에 수행할 수 있다.

메모리 저장 명령은 ALU(EX단계)를 사용하지 않고, load/store형 기계의 경우 일반 연산 명령들은 모든 연산이 레지스터 사이에서 발생하여 메모리 접근(ME)을 사용하지 않는다. 따라서, EX와 ME를 분리

하여 파이프라인을 구성하면 명령들의 실행 사이클이 줄어든다. 곱셈과 나누기 등의 다중 실행 사이클 명령(MTC)의 경우는 일반 연산 명령과 같다.

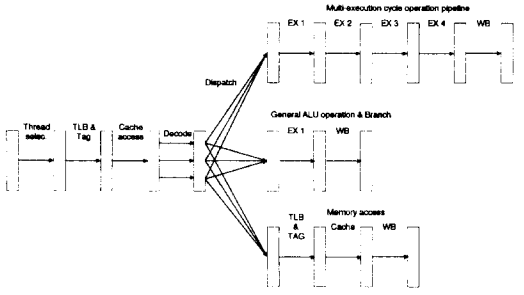

그림 8. 파이프라인의 구조
Fig. 8. Structure of pipeline.

그림 8은 MIPS R3000의 명령들을 수행하도록 파이프라인을 분리 구성한 구조다[7]. 곱하기와 나누기를 위한 파이프라인 연산 장치를 가정하였다. 이 실행 파이프라인은 DEC Alpha 21064의 실행 파이프라인과 유사한 구조다[8],[9]. 다중 실행사이클을 요구하는 명령의 파이프라인 처리는 자원 공유를 가능하게 하여 실행 효율을 높인다. [10]에서는 명령간의 의존성에 의한 문제를 interlocking에 의해서 처리하기 때문에 의존성 검출을 위한 장치가 파이프라인에 포함되어 구현에서 좀더 복잡하게 된다.

(4) 다중 명령 투입 다중스트리밍 쓰레드 스케줄링 알고리즘

다중 명령투입을 고려하면 그림 6에 제시된 쓰레드 스케줄링 알고리즘은 그림 9와 같이 수정된다. 명령어 풀의 최좌단에 존재하는 모든 명령들은 자원이 충분한 경우 파이프라인 장애나 명령간 의존성에 대한 고려 없이 모두 동시 실행될 수 있다. 따라서, 실행 파이프라인을 동일 종류 파이프라인을 복수로 확보하여 실행 능력을 높이는 것이 가능하다. 그림 6에서 (2) 단계는 명령어 풀에서 투입 가능한 하나의 명령을 검색하지만, 그림 9의 (2) 단계에서는 프로세서에 존재하는 실행 파이프라인별로 명령을 검색하여 해당 파이프라인에 명령을 공급한다. (3) 단계는 공통적으로 쓰이지 않은 자원은 nop 공급으로 사이클을 낭비하는 동작이다. 이들 처리 단계들은 풀의 입장에서 명령의 공급과 소비의 두 단계로 나누어질 수 있다. (1), (4) 단계는 공급에 (2), (3)는 소비에 해당하고 이 두가지 단계는 동시 수행이 가능하다. 동시에 풀에서 명령을 꺼내면서 명령을 투입하는 것이 가능하다. 그리고, 다수의 풀에서 다수의 명령들을 선택하는 것도 단일 사이클에 실행가능하다. 따라서, 이 알고리즘의 구현은 파이프라인 구현이 가능하다.

(1) 모든 유효한 쓰레드의 현재 PC가 지시하는 위치의 명령들을 패치하고 디코드하여 명령어 풀에 넣는다. 이는 초기화 단계이다.

(2) 명령어 풀 내의 모든 nop 명령을 제거한다. 명령어 풀에서 모든 실행 파이프라인에 해당하는 명령을 담고 있는 쓰레드를 파이프라인별로 하나씩 선택하여 실행 파이프라인으로 투입한다.

(3) 모든 파이프라인에 명령이 투입되었으면 (4)로 진행한다. 명령이 투입되지 않은 파이프라인이 있으면 nop을 명령이 투입되지 않은 파이프라인에 투입하고 (4)로 진행한다.

(4) (3)에서 선정된 쓰레드들과 nop이 제거된 쓰레드들의 다음 명령을 패치하고 디코드하여, 명령어 풀의 빈자리를 채운다. (2)로 진행한다.

그림 9. 다중 명령어 투입을 지원하는 쓰레드 스케줄링 알고리즘
Fig. 9. Thread scheduling algorithm with multiple instruction issuing support.

Ⅳ. 성능 평가

1. 모의 실험기 모형

모의 실험 프로세서는 MIPS R3000의 명령들 중에서 부동 소수점 연산 장치를 제어하는 명령들과 system call 등이 제거되었다. 각 프로세서 모형은 각각의 쓰레드들을 위해 32개의 레지스터를 갖고 있는데, 각 쓰레드는 자신에 할당된 32개의 레지스터만을 사용할 수 있다. 이 레지스터들은 많은 자료를 프로세서 내부에서 유지하도록 하여 연산량이 큰 쉐이딩에서 실행 성능을 높인다. 쉐이딩에 많이 사용되는 정수 곱하기를 위한 파이프라인으로 구성된 연산장치를 가정하였다. 다중쓰레딩 지원을 위해서 쓰레드 생성과 제어용 명령으로 setthrdst(set thread status), readthrdid(read thread ID), suicid(suicide)등의 명령을 추가하였다. setthrdst는 수행 중인 쓰레드들의 상태를 제어하는 쓰레드 상태 레지스터의 값을 바꾸는 명령이다. 쓰레드의 생성은 쓰레드 상태 레지스터에 생성시킬 쓰레드의 상태에 해당하는 위치의 값을 활성값으로 바꾸어 수행한다. readthid는 각 쓰레드가 서로 다른 자료를 처리하도록 자료를 구분하는데 쓰이는 쓰

레드 식별자(thread ID)를 가져오는 명령이다. suicid는 쓰레드를 정지 상태로 만들어서 더 이상 수행되지 않도록 한다.

프로세서 모형의 기본으로 사용된 그림 8의 파이프라인 구조는 파이프라인 장애의 완전한 제거를 위해서 5개 이상의 쓰레드를 요구한다. 모의 실험기의 구성에서는 기본적으로 최대 8개의 쓰레드를 유지하도록 하였다. 명령어 패치 대역폭이 커서 단일 쓰레드가 수행되는 경우에 분기 명령에 인접한 명령들이 과다하게 프로세서로 유입되어 장애를 발생시킬 것을 고려하여 분기 명령의 지연틈을 8로 고정했기 때문에 쓰레드 수가 8보다 큰 것이 스케쥴링에 유리하다.

2. 사이클 수준 시뮬레이션

모의 실험은 프로세서의 사이클 단위 행동을 기초로 이루어진다. 시뮬레이터는 다음의 그림 10과 같은 프로세서 내부 동작을 한 사이클에 한번 씩 수행하고 그 결과를 결과 저장소에 보관하는데, 이것은 파이프라인에서 각 스테이지의 결과가 스테이지 사이의 레지스터 또는 레치에 저장되는 것과 같다. 현 사이클의 결과는 이전 사이클의 결과를 계산 혹은 프로세서 처리과정의 결과를 참조하여 이루어진다. 모의 실험기는 모형 프로세서의 바이트 코드 명령어를 수행한다. 모의 실험을 위한 프로그램은 C 언어로 작성하고 컴파일하여 얻은 어셈블리어 코드에 다중스트리밍을 수행하는데 필요한 몇 가지 명령을 첨가하여 최종 바이트 실행 코드를 만든다. 이렇게 만들어진 코드는 시뮬레이터에 계산 대상이 되는 그림에 대한 정의 자료와 함께 적재되어서 실행 개시한다. 프로세서는 수행을 시작하면 특정 위치에서부터 코드의 실행을 시작한다. 이때는 하나의 쓰레드가 수행 중이다. 이때 수행되는 쓰레드를 master라한다. Master는 적절한 초기화 동작과 실제 계산을 수행하는 쓰레드들을 생성하는 작업을 수행한다. 이렇게 master에 의해 생성된 쓰레드들은 수행을 마치면 소멸된다. 실행 결과는 발생 시점에서 시뮬레이터가 메모리나 레지스터의 값을 파일로 저장하도록 구성하여 얻는다. 프로세서 모의 실험기는 실제 실행할 쓰레드 수만큼의 레지스터 집합을 가지도록 실행 전에 조정된다. 레지스터의 수와 최대 유지가능 쓰레드의 수는 같고 이 수는 프로세서의 설계 요소다.

실험에서 캐쉬와 가상 메모리에 대한 고려가 단순화 되었다. 실제의 캐쉬와 가상 메모리는 접근에서 100% 적중하는 이상적인 경우를 가정하고 캐쉬와 가상 메모리에 대한 접근을 위해서 소비되는 사이클 소비만이 고려되었다.

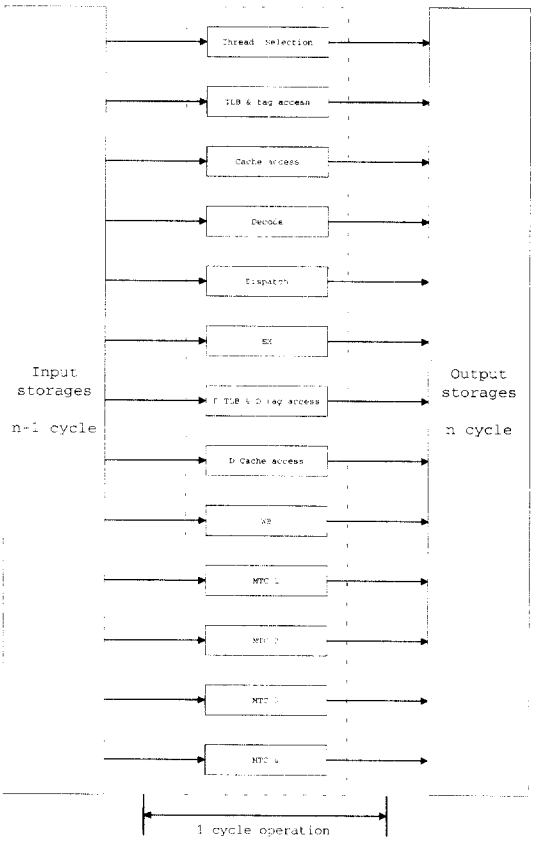

그림 10. 사이클 단위 동작들
Fig. 10. Cycle-wise functions.

그림 10에서 cache access와 Tag & TLB access로 나타난 처리가 소비되는 사이클을 나타낸다. 본 실험에서는 작은 규모의 동일 코드가 서로 다른 자료의 집합에 적용된다. 따라서, 코드 메모리에 대한 캐쉬 적중 확률은 매우 높게 유지되므로 다중스트리밍이 캐쉬에 미치는 영향은 크지 않다. 자료 메모리의 경우 쓰레드 사이에 공유되지 않는 메모리로 캐쉬 크기를 증가시키는 요인이 될 수 있다. 연속한 nop을 encode하여 하나의 단어(word)로 표현하여 nop의 사용에 의한 캐쉬에 대한 영향은 최소화할 수 있다.

3. 실험 결과

3차원 그래픽을 위한 다중스트리밍 프로세서의 성능을 평가하기 위해서 3차원 공간의 물체를 Gouraud

쉐이딩(shading)하는 프로그램을 실행하였다. 제안한 설계를 3차원 그래픽에 적용하여 프로세서 자원의 변화에 따른 프로세서의 성능 변화, 쓰레드수의 변화에 따른 성능의 변화 등을 조사하였다. 그리고, 그림 8의 파이프라인을 사용한 프로세서와 비교하여 성능 향상 정도를 측정하였다.

다중스트리밍 프로세서에서 쓰레드는 상호 독립적으로 실행되는 프로그램의 흐름이다. 모든 프로세서는 자원이 파이프라인으로 구성되어 선점성이 없는 것으로 가정하였다. 쓰레드 선택에 의해서 발생하는 사이클 단위의 문맥 전환(Context switching)은 상호 배제의 소프트웨어적인 구현을 불가능하게 한다. 실험 프로세서는 상호 배제를 위한 하드웨어적인 지원을 포함하지 않았다. 따라서, 프로세서가 실행하는 프로그램은 모든 경계 지역(critical section)을 제거한 형태로 작성되었다. 실제 모의 실험에서 수행한 Gouraud 쉐이딩은 경계 지역이 되는 Z-버퍼가 외부의 하드웨어로 구현됨을 가정하였다. 각 쓰레드가 그림 상의 물체(Object)를 이루는 삼각형을 한번에 하나씩 전담하여 다중스트리밍에 의해 실행하도록 프로그램을 작성하였다. 각 쓰레드는 담당한 삼각형을 계산한 다음 아직 계산되지 않은 삼각형을 선택하여 계산을 계속한다. 이러한 동작은 화면을 구성하는 모든 삼각형들이 계산될 때까지 반복된다. 따라서, 모든 쓰레드는 동일한 코드를 수행한다. 이러한 실행 코드의 생성은 그래픽의 특성에 맞도록 사전에 코드에 병렬성과 자원 이용에서 쓰레드간의 구별을 위한 장치가 고려되어 작성된 것으로 병렬화 컴파일러가 사용된 것은 아니다. 각 쓰레드는 프로세서 내부에서 처리될 때 쓰레드간의 구별을 위해 할당되는 쓰레드 식별자를 참조하여 각각 다른 메모리 영역을 참조하고, 각각 다른 삼각형들을 선택하도록 작성되었다. 쓰레드 식별자는 명령어에 의해서 쓰레드에 할당된 레지스터로 적재가 가능하다.

실험에서 사용된 물체는 624개의 삼각형으로 이루어진 서양 장기말이다. 실험된 프로세서 모형은 네 가지로 각각 내부 자원의 수가 다르다. 추가 자원에 대해서는 표 1에 보였다. 실험된 프로세서 모델 별로 전체 처리 과정에 소요되는 사이클과 nop으로 낭비되는 사이클을 측정하여 표 2에 보였다. RISC 프로세서 MIPS R3000의 실행 사이클에 비해 sMIPS로 표시된 제안된 스케쥴링 알고리즘과 그림 8의 파이프라인을 사용한 프로세서는 48% 정도 적은 수행 사이클을 보였다. 이는 MIPS의 1.94배의 성능이다. 사이클 낭비는 sMIPS가 1129010 사이클 적게 나타났다. 이는 사이클의 낭비를 84% 줄인 것이다.

표 1. 추가 자원
Table 1. Added resources.

Processor	Added resources
MC	ALU pipeline 1 unit
MMC	Memory access pipeline 1 unit
MMMAC	ALU & Memory access pipeline each 1 unit

MC는 프로그램의 연산 요구량에 기초하여 요구가 많은 ALU 연산을 보강하기 위해 ALU를 2개로 늘린 프로세서다. 전체 실행 사이클이 sMIPS에 비해 2796853 사이클 줄었고 실행 속도가 MIPS에 비해 3배 이상으로 증가하였다. 자원 요구에 대한 자료로 각 자원을 한 개씩 보유한 다중스트리밍 프로세서의 자원별 동시 요구량에 대한 사이클 수의 분포를 그림 11에 보였다. 가로축은 특정자원이 동시 요구된 량이고, 수직 축은 전체 실행에서 특정 량의 동시 요구가 발생한 사이클 수다.

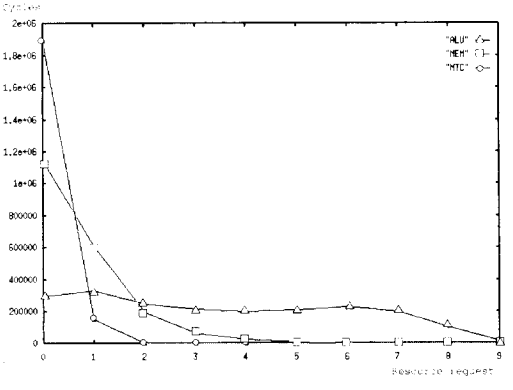

그림 11. 자원 요구량에 따른 사이클의 분포
Fig. 11. Distribution of cycle for resource request.

MMC는 MEM 파이프라인을 추가한 모형이다. MC에 비해 실행 사이클의 수가 9.6% 줄었다. ALU에 비해 요구가 적은 메모리에 대한 자원 추가는 성능을 크게 개선시키지 못하고 있다. MMMAC는 MMC에 사용 쓰레드의 수를 늘리고 이를 보조하도록 패치 대역폭을 50% 증가시킨 프로세서다. MIPS에 비해 3.3배 이상의 성능을 보인다. 낭비되는 사이클은 95.5% 줄었다. 단일 사이클에 평균 2.23개의 명령을

실행하고 있는 것이다.

표 2. 프로세서에 따른 실행 시간과 낭비된 사이클

Table 2. Run cycle and idled cycle for each processor.

Processor	Run (cycles)	Idle (cycles)
MIPS	4187491	1339039
sMIPS	2160053	210029
MC	1390638	68058
MMC	1257313	12071
MMMAC	1240901	9318

그림 12와 13은 쓰레드의 공급이 원활하지 않은 상황에서 프로세서의 성능 변동을 관찰하기 위한 12개 삼각형으로 이루어진 물체를 Gouraud 쉐이딩하는 실험 결과다. 6개 이상의 쓰레드가 있을 때에 실행 사이클과 낭비되는 사이클이 늘어나는 경향이 있다. 이는 프로그램의 실행 초기에 공급된 쓰레드들이 비슷한 시간에 종료하지 않기 때문이다. 쓰레드에 따라 처리해야하는 자료의 양적 차이로 실행 후반부에 소수 쓰레드가 남기 때문에 프로세서의 성능이 떨어진다.

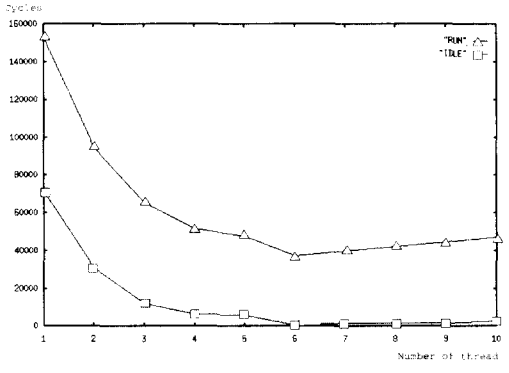

그림 12. 쓰레드 수의 변화에 따른 실행 사이클과 낭비되는 사이클

Fig. 12. Idled cycle and run cycle under varying thread number.

그림 12에서 쓰레드수가 프로그램 실행 시간 전체에 걸쳐 균형을 유지할 수 있는 6개 쓰레드가 최적이다. 프로그램이 최대로 생성 가능한 쓰레드가 12개이므로, 12의 약수인 6, 4, 3 개의 쓰레드를 유지하면 실행 시간 전체에 걸쳐 쓰레드수가 균형을 이루기 때문이다. 그러나, 쓰레드의 수가 3개 또는 4개인 경우에는 스케줄링을 수행하기에 쓰레드 수가 부족하여 프로세서의 성능이 감소하였다.

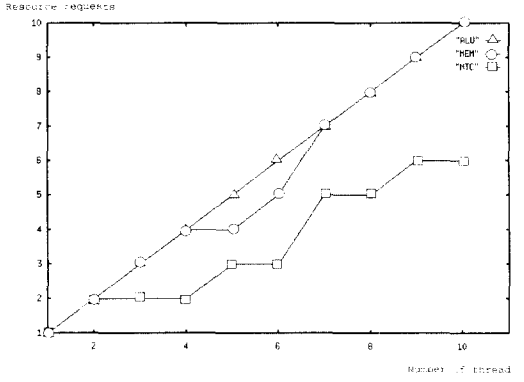

그림 13. 쓰레드 수에 따른 최대 자원 요구량 변화

Fig. 13. Peak resource request under varying thread number.

그림 13에 쓰레드 수에 따른 최대 자원 요구량을 보였다. ALU의 경우 쓰레드 수에 비례하여 증가하고 메모리 접근인 MEM도 비슷하다. ALU와 MEM가 많이 사용됨을 알 수 있다. 곱셈기에 해당하는 MTC에 대한 요구는 많지 않다.

그림 14는 MMMAC의 자원별 동시 요구량에 대한 사이클 수의 분포다. 그림 11과 같이 가로축은 특정 자원이 동시 요구된 량이고, 수직 축은 특정 량의 동시 요구가 발생한 사이클 수다. 그림 11에 비해서 ALU, MEM에 대한 동시 요구가 적은 쪽에 편중되었다. 자원의 공급에 따라 명령 소비 형태가 달라졌다.

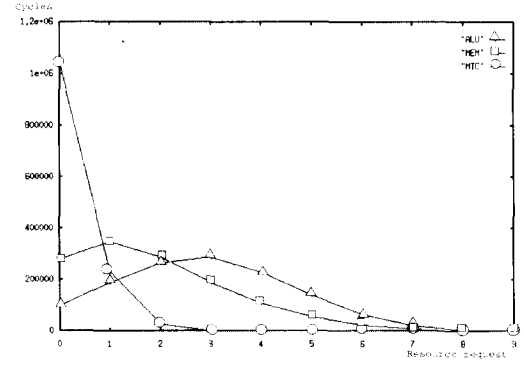

그림 14. 자원 추가 시의 자원 요구량

Fig. 14. Resource request with added resources.

VI. 결 론

본 논문에서는 3차원 그래픽 가속을 위한 명령어 프로세서의 설계를 위하여 다중 명령 실행이 가능한 다중스트리밍 프로세서 구조와 쓰레드 스케쥴링 알고리즘을 제안하였다. 제안된 프로세서의 3차원 그래픽 수행에서의 성능을 측정하기 위해서 모의 실험을 수행하였다. 제안한 다중 명령 실행 다중스트리밍 프로세서는 다중스트리밍의 지연틈 활용에 의해서 자원 활용도가 우수하고, 다중쓰레딩에 의한 병렬성의 이용으로 처리 속도면에서 뛰어난 성능을 보인다. 다중스트리밍은 분기 예측과 명령간 의존도 검사 등을 행하는 복잡한 하드웨어를 사용하지 않고 프로세서의 자원 활용도를 높인다. 다중스트리밍 프로세서는 쓰레드 간의 프로세서 내부 연산 자원의 공유가 가능하여 다중프로세서에 비해 같은 성능을 얻기 위해서 더 적은 연산 자원을 요구한다. 이는 곱셈기와 같은 전용 파이프라인 연산 장치에 의한 칩 면적에 대한 요구를 줄여서 실제 제작을 유리하게 한다. 다중스트리밍은 병렬성을 쓰레드 사이에서 차는 반면 슈퍼스칼라는 한 쓰레드 내에서 병렬성을 찾는다. 한 쓰레드 내의 병렬성은 명령간의 의존 관계에 의해서 제한을 받게 된다. 따라서, 슈퍼스칼라는 병렬성 자체의 제한이 크다. 반면 다중스트리밍은 쓰레드 간의 독립성에 기초하여 병렬성을 찾기 때문에 병렬성의 탐색이 쉽고 쓰레드 공급에 따라 더 큰 병렬성을 가질 수 있다. 따라서, 공급자원의 량이 커질수록 다중스트리밍은 성능 증가를 보이는 반면 슈퍼스칼라는 성능의 증가에 한계가 있게 된다. 그래픽과 같이 쓰레드가 프로세서의 동작 기간 전체에 걸쳐 충분히 공급되는 응용 분야에서는 다중 명령 투입 다중스트리밍 프로세서가 적합한 명령어 프로세서 설계가 될 것이다. LIW 형태의 프로세서는 컴파일 단계에서 슈퍼스칼라 프로세서가 활용하는 명령어 흐름내의 병렬성을 활용하기 때문에 다중스트리밍과 함께 적용하는 경우에 더 넓은 병렬성 활용을 가져올 것으로 보인다.

Hirata의 경우 다중컴퓨터를 위한 처리 단위의 설계인 반면 본 연구는 단일 프로세서로 실시간 그래픽의 빠른 구현에 중점을 두고 수행되었다. Hirata의 연구에서는 지연틈이 필요하지 않은 쓰레드로의 문맥 전환과 명령간 의존도 제거 방법으로 interlocking을 사용하여 주로 응용 자체 특성인 메모리 접근에서의 프로세서의 성능 저하를 제거하였다. 여기서 interlocking에 의한 명령간 의존성 제거는 하드웨어 복잡도를 증가시키고, 코드의 재배치에 의한 최적화를 어렵게 한다. 반면 본 연구에서는 명령 실행장치를 모두 파이프라인으로 구성하고 nop에 의한 명령별 지연틈 표현 방식을 사용하여 간단한 하드웨어로 구성이 가능하고, 코드의 재배치에 의한 최적화가 가능하여 소프트웨어적인 프로세서의 성능 향상을 가능하게 한다. 파이프라인에 의한 실행 장치 구성은 한 명령이 실행 장치를 여러 사이클 동안 독점하는 것을 막아서 각 사이클마다 다른 명령의 투입이 가능하도록 하여 모든 지연틈의 프로세서 성능 저하를 제거 가능하도록 한다.

이상의 다중 명령 투입 다중스트리밍 프로세서는 쓰레드의 대량 공급이 쉬운 응용 분야에서 목표 프로그램의 분석을 통한 응용 분야의 자원 요구에 기초하여 최적화된 ASIP를 자동 생성하기 위한 기반 설계로 쓰일 수 있을 것이다. 앞으로 실제 ASIP의 설계에 이 연구를 적용하기 위해서 다중스트리밍이 실제 캐쉬에 미치는 영향을 시뮬레이션을 통하여 조사하고, 적절한 캐쉬의 형태를 연구한다. 아울러 제안된 프로세서 구조를 다른 형태의 3차원 그래픽 처리 방법에 적용된 경우에 성능을 조사하고, LIW 등의 다른 형태의 병렬 처리와의 동시 적용에 의한 병렬성 이용을 확대하는 방안을 연구한다.

참 고 문 헌

[1] J. Foley, A. van Dam, S. Feiner, and J. Hughes, *Computer graphics prin-ciples and practice*, Addison-Wesley publishing company, New York, p. 855, 1990

[2] N. Gharahorloo, S. Gupta, E. Hokenek, P. Balasubramanian, W. Bogholtz, C. Mathieu, and C. Zoulas Subnaonsecond Pixel Rendering with Million Transistor Chips," *SIGGRAPH 88*, p. 41-49, 1988

[3] R. Alverson, D. Callahan, D. Cummings, and B. Koblenz, "The Tera computer system", *Proc. ACM Int. Conf. Supercomputing*, Amsterdam, The Netherland, pp. 1-6, June, 1990

[4] D. C. McCrackin, "Eliminating interlocks in deeply pipelined processors by delay

enforced multistreaming", *IEEE Transaction on Computers*, vol. 40, no. 10, pp. 1125-1132, Oct. 1991

[5] D. C. McCrackin, "Practical delay enforced multistream (DEMUS) control of deeply pipelined processors", *IEEE Transaction on Computers*, vol. 44, no. 3, pp. 458-462, Mar. 1995

[6] J. Fisher, J. Ellis, J. Ruttenberg, and A. Nicolau. "Parallel processing: a smart compiler and a dumb machine", *In Proc. '84 Sym. on Compiler construction*, Montreal, Canada, pp. 37-47, June, 1984

[7] D. Tabak, *RISC Systems*, Research Studies Press LTD., England, p. 49, 1987

[8] R. L. Sites, "Alpha AXP architecture", *Digital Technical Journal*, vol. 4, no. 4, Special Issue 1992

[9] J. E. Smith, C. Richard, and S. Weiss, "PowerPC 601 and Alpha 21064: A tale of two RISCs", *IEEE Journal Computer*, vol. 27, no. 6, p. 46-58, June, 1994

[10] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa. An elementary processor architecture with simultaneous issuing from multiple threads. In *19th Annual International Symposium on Computer Architecture*, p. 136-145, May 1992

저 자 소 개

朴 鏞 珍(正會員)

1995년 2월 경북대학교 전자공학 학사. 1997년 2월 경북대학교 전자공학 석사. 현재 삼성전자 기간네트웍 사업부 소속. 주관심 분야는 집적회로 설계및 테스트, 병렬 처리, 특수용도용 명령어 프로세서

李 東 浩(正會員)

1979년 2월 서울대학교 전자공학과 졸업. 1981년 2월 KAIST 전산학과 졸업(이학석사). 1981년 3월 ~ 1982년 7월 KAIST 전산연구소 기술원. 1982년 8월 ~ 1992년 7월 ETRI 선임연구원, 1992년 8월 (미) 아이오와 대학 전산학과 졸업(박사). 1992년 8월 ~ 1993년 1월 (미) Motorolar senior CAD engineer. 1993년 3월 ~ 현재 경북대학교 전자공학과 조교수. 주관심 분야는 컴퓨터 구조, 설계자동화, 프로그래밍 언어