# 루프를 효과적으로 처리하는 PASC 프로세서 구조

지 승 현[†] · 박 노 광[††] · 전 중 남[†††] · 김 석 일[†††]


## 요 약

본 논문에서는 연산처리기와 스케쥴러를 쌍으로 구성하고 이들을 여러 개 나열하여 만든 PASC(PArtitioned SCheduler) 프로세서를 제안하였다. PASC 프로세서의 스케쥴러는 단위명령어를 연산처리기에게 제공할 것인지, 또는 자료종속성이나 연산자원의 충돌 등으로 인하여 다음 사이클 동안 연산처리기의 실행유니트를 정지시킬 것인지를 연산처리기별로 독립적으로 결정한다. PASC 프로세서는 자원충돌이나 자료종속성이 있는 연산처리기만 NOP(No OPeration)을 수행하기 때문에 목적 코드를 압축하는 효과를 얻을 수 있다. 즉, 비록 루프의 마지막 명령어와 다음 반복을 시작하는 명령어가 하나의 긴명령어에 포함되지 않더라도 명령어의 스케쥴링 과정에서 명령어간에 자료종속성이나 연산자원의 충돌이 존재하지 않는다면 두 개의 명령어가 동시에 스케쥴링될 수 있다. 따라서 루프에서 자료종속성이나 연산자원의 충돌이 없는 어떤 반복(iteration)의 마지막 명령어와 다음 반복의 처음 명령어를 하나의 긴명령어로 구성하는 것과 동일한 효과를 얻을 수 있으므로 이들을 순서대로 수행되어야 하는 기존의 VLIW 구조나 SVLIW 구조에 비하여 루프처리에 따른 실행사이클을 줄일 수 있다. 모의실험에서도 PASC 프로세서 구조가 VLIW 프로세서 구조나 SVLIW 구조에 비하여 루프를 빠르게 처리함을 확인할 수 있었다.


# PASC Processor Architecture for Enhanced Loop Execution

Sung-Hyun Jee[†] · No-Kwang Park[††] · Joong-Nam Jeon[†††] · Suk-Il Kim[†††]


## ABSTRACT

This paper proposes PASC(PArtitioned SCheduler) processor architecture that equips with a number of functional unit and an individual scheduler pairs. Every scheduler of the PASC processor can determine whether a unit instruction can be issued to the associated functional unit or it is to be waited until next cycle caused by a resource collision or data dependencies.

In the PASC processor, only the functional unit with a resource collision or data dependencies waits by executing a NOP(No OPeration) instruction and the other functional units execute their own instructions. Therefore we can expect the code compaction effect on the PASC processor. Thus, the last instruction of a loop at certain iteration and the very first instruction of the loop at the next iteration can be scheduled simultaneously if the two instructions do not incur any resource collision or data dependencies. Therefore, we can expect that such two instructions without any resource collision and data dependencies are packed into the same very long instruction word and thus, the two instructions are executed concurrently at run time. As a result, we can shorten execution cycles of a loop comparing to the execution of the loop on a traditional VLIW or SVLIW processor architecture. Simulation result also promises faster execution of loops on a PASC processor architecture than those on a VLIW and SVLIW processor architecture.


※ 이 연구는 정보통신부의 정보통신 우수시범학교 지원사업에 의하여 수행된 것입니다.

† 정 회 원 : 천안외국어대학 사무자동화과 교수
†† 준 회 원 : 천안외국어대학 교수
††† 종신회원 : 충북대학교 전자계산학과 교수

논문접수 : 1998년 8월 22일, 심사완료 : 1999년 3월 16일

## 1. 서 론

보다 빠른 계산이 가능한 고성능 프로세서를 제작하기 위해서는 단위시간당 처리하는 명령어의 수를 증가시켜야 한다. 전통적으로 고성능 프로세서를 개발하기 위한 방법으로 프로세서의 클록 속도를 높이는 방법이 사용되었다. 그러나 이 방법에 의한 프로세서의 성능 향상은 이미 프로세서의 집적도가 물리적 한계에 도달함에 따라 새로운 프로세서를 제작하는 기술이 개발되기 전에는 획기적인 성능 향상이 어렵게 되었다. 이에 따라 고성능 프로세서를 구현하기 위한 새로운 돌파구로 동시에 여러 개의 명령어를 중첩하여 실행시킴으로써 단위시간당 처리하는 명령어의 수를 증가시키는 방법이 대두되었고, 응용 프로그램 속에 내재한 명령어 수준의 병렬성(ILP)을 추출하여 가능한 한 많은 수의 명령어를 동시에 처리하도록 하는 ILP(Instruction Level Parallelism) 프로세서 기술이 광범위하게 연구되었다[1-11].

명령어 수준의 병렬성을 이용하여 프로세서의 성능을 향상시킨 대표적인 프로세서인 슈퍼스칼라 프로세서는 상업적으로 가장 성공한 구조이다. 그러나 슈퍼스칼라 프로세서는 제한된 윈도 사이즈로 인하여 충분한 명령어 수준의 병렬성을 추출하지 못하는 문제점이 있다[15-17]. 이러한 문제를 해결하기 위해서도 컴파일러가 응용 프로그램내의 병렬성을 최대한 추출하여 여러 개의 단위명령어를 하나의 긴명령어로 구성한 목적코드를 생성하도록 하고, 실행 시에는 목적 코드로부터 긴명령어들을 순서대로 인출하여 실행하도록 하는 VLIW 프로세서 구조에 대한 연구가 더욱 활발히 진행되고 있다[12-15].

지금까지 VLIW 프로세서와 관련한 연구들은 주로 컴파일러 상에서의 최적화 기법들이 대부분을 차지한다. 트레이스 스케쥴링(trace scheduling)등의 전역 스케쥴링(global scheduling) 기법이나 소프트웨어 파이프라이닝(software pipelining)등의 최적화 기법은 NOP (No OPeration)명령어가 들어갈 긴명령어에 다른 유용한 명령어를 이동시키거나 복사하는 작업을 수행하여 기본 블록(basic block)의 경계를 넘어 루프가 없는 코드를 대상으로 하여 병렬화를 수행하는 기법이다[9-13, 24,25]. 그 중에서도 소프트웨어 파이프라이닝 기법은 강력한 스케쥴링 기법의 하나로 루프(loop)를 명령어 수준에서 최적화 한다. 이 기법은 자원(resource)과 종속성(dependence)이라는 제약 조건하에서 각 반복이 수행 완료되기 전에 다음 반복을 실행하도록 루프 몸체(loop body)를 재구성한다[12,13,24]. 이 결과, 각 반복이 다른 반복과 중첩 실행되도록 허용하여 루프의 실행 속도가 크게 개선되는 특징이 있다.

최근에는 기존의 VLIW 프로세서 구조를 변형하여 컴파일러 기법에 주로 의존하던 병렬성 추출 과정을 하드웨어로 처리하여 컴파일 과정을 단순하게 할 뿐 아니라 프로세서의 성능을 향상시키기 위하여 동적 스케쥴러를 추가한 구조에 관한 연구도 활발히 이루어지고 있다[20-23]. 예를 들어, SVLIW 프로세서 구조[21, 23]는 VLIW 프로세서 구조에 명령어 인출 단계에서 캐시미스로 인하여 명령어 인출 사이클이 연장되어야 하는 경우에도 명령어 파이프라인을 진행시키도록 명령어 스케쥴링 단계를 추가하여 자료종속성이나 자원충돌가능성이 있는 긴명령어 사이에 삽입되어야 하는 LNOP(Long NOP : NOP으로만 구성된 긴명령어)을 제거하도록 하였다. 그러나 이 구조도 기존의 VLIW 구조에서와 마찬가지로 루프처리시에는 각 반복의 중첩 실행을 가능하게 하는 논리회로를 가지고 있지 않아 루프처리를 컴파일러에 일임하고 있다.

한편, VFPE 프로세서 구조[20]는 연산처리기의 종류를 BU(Branch Unit), TU(Transfer Unit) 및 AU (Arithmetic login Unit) 등과 같이 명령어의 종류별로 구분하고, 목적 코드를 생성하는 과정에서 단위명령어를 명령어의 종류별로 연산처리기에 배치한 긴명령어를 생성한다. 또한, 연속한 두 개의 긴명령어가 순서대로 실행되는 경우에 먼저 실행되는 긴명령어가 사용하지 않는 연산처리기를 다음 차례의 긴명령어에서 사용할 수 있도록 연속한 두 개의 긴명령어를 하나의 긴명령어로 압축시키고, 각각의 명령어에 자료종속성 정보를 포함시켜 명령어가 실행되는 과정에서 동적으로 긴명령어를 구성하는 단위명령어간의 동기가 이루어지도록 한다. 따라서 루프의 경우에 루프의 마지막 부분과 처음 부분이 되는 단위명령어가 하나의 긴명령어로 구성되도록 하고, 두 개의 단위명령어가 동시에 수행되도록 하여 루프를 빠르게 수행할 수 있다. 그러나 VFPE 구조는 프로세서를 구성하는 연산처리기를 VLIW 구조나 SVLIW 구조와 같이 동일한 종류의 연산처리기를 다수 이용하여 프로세서를 구성할 수 없으며, 또한, 응용 프로그램의 명령어 분포가 연산처리기의 종류별로 고르게 분포되지 않았을 경우 프로세서의

연산자원을 충분히 활용할 수 없는 단점이 있다.

본 논문에서는 루프처리에 효과적인 VFPE 프로세서의 장점과 동일한 여러 개의 연산처리기로 프로세서를 구성할 수 있는 SVLIW 프로세서의 장점을 혼합하여 동일한 여러 개의 연산처리기로 프로세서를 구성하되 빠른 루프처리가 가능한 PASC(PArtitioned SCheduler) 프로세서 구조를 설계하였다. 본 논문에서 제안한 PASC 프로세서용 목적 코드는 루프 실행시 각 반복(iteration)의 마지막 명령어와 다음 번 반복(iteration)의 첫 명령어를 하나의 긴명령어로 압축하고 긴명령어를 구성하는 단위명령어간의 자료종속성을 긴명령어에 포함시켜 긴명령어가 동시에 수행되더라도 단위명령어간의 동기가 유지되도록 하였다. 이를 위하여 SVLIW 구조와 같이 실행시 단위명령어간의 동기를 유지하기 위한 하드웨어를 포함시켰다. 또한, PASC 구조는 기존 SVLIW 구조용 목적 코드내 자료종속성이 있는 긴명령어들을 하나의 긴명령어로 압축시키는 방법으로 목적 코드를 생성하므로 목적 코드 내에 NOP 명령어가 차지하는 비율이 기존 SVLW 구조의 경우에 비하여 적어져서 프로세서의 캐시 효율을 개선하는 효과도 얻을 수 있다.

본 논문의 구성은 다음과 같다. 제 2절에서는 기존의 ILP 프로세서 구조를 개괄하고, 본 논문에서 제안하려고 하는 동적 스케쥴러를 포함하는 프로세서 구조인 SVLIW 구조와 VFPE 구조를 분석하였다. 제 3절에서는 PASC 프로세서용 긴명령어 실행 패턴과 긴명령어 코드의 구성에 대하여 설명하였고, 제 4절에서는 프로세서를 구성하는 연산처리기와 이 연산처리기에 명령어를 제공하는 스케쥴러를 하나의 쌍으로 구성하고 이들 쌍을 여러 개 이용하여 프로세서를 구성한 PASC 구조에 대하여 설명하였다. 제 5절에서는 제안한 PASC 프로세서 구조, 기존의 VLIW 프로세서 구조와 SVLIW 프로세서 구조를 비교하여 성능이 얼마나 향상되었는지 시뮬레이션 결과를 토대로 비교하였다. 그리고 마지막으로 제 6절에서는 본 논문의 결론을 맺고 향후 계획을 기술하였다.

## 2. ILP 프로세서

### 2.1 VLIW 프로세서

VLIW 프로세서 구조에서는 컴파일러가 생성한 명령어 스트림의 순서에 따라 긴명령어를 수행한다. 즉, 어떤 긴명령어와 이어서 수행되는 긴명령어가 자원충돌이나 자료종속성으로 인하여 일정한 사이클 만큼 수행을 기다려야 하는 경우에는 컴파일러가 이를 정확히 계산하여 필요한 만큼의 NOP으로만 구성된 긴명령어(LNOP : Long NOP)를 긴명령어 사이에 삽입해 주어야 한다. 따라서 컴파일러의 입장에서는 긴명령어 흐름을 구성하는 과정에서 각 명령어가 수행되는데 필요한 연산처리기와 연산처리기의 자원사용 상태 및 사이클의 수를 정확히 예측하여야 하며, 이 값을 정확히 예측할 수 없을 경우에는 오류가 발생하여 계산 결과를 사용할 수 없게 된다. 이러한 특성으로 인하여 응용 프로그램에서 실행사이클의 최악시간을 계산하는 기법에 관한 연구도 진행된 바 있다[14,15].

긴명령어에 삽입된 NOP은 목적 코드의 크기를 늘일 뿐 프로그램의 유용한 부분으로 사용되지 않는다. 이러한 이유로 NOP을 제거하여 목적 코드의 크기를 줄이려는 연구도 시도되고 있다. 특히 실수연산이 많이 포함된 경우에는 실수연산에 필요한 사이클이 정수연산과는 달리 실행사이클이 길고 불규칙적이어서 연속한 긴명령어간에는 LNOP이 많이 포함되게 된다. 그런데 LNOP이 목적 코드에 많이 포함되면 목적 코드의 크기를 증가시켜서 캐시에 적재된 목적 코드 중에서 실행될 가능성이 높은 코드가 차지하는 비율이 적어지게 되며, 프로그램이 실행되는 과정에서 그만큼 캐시미스율이 높아져 메모리참조로 인한 파이프라인의 빈번한 정지상태가 초래되기도 한다[23]. 뿐만 아니라 ILP(Instruction Level Parallelism)가 적은 경우에도 목적 코드 내에서 NOP이 차지하는 비율이 증가하여 캐시미스가 빈번히 일어난다. 이러한 문제점을 목적 코드를 생성하는 과정에서 해결하기 위하여 기본 블록(basic block)을 확장하기 위한 다양한 기법들이 연구되었다[9-13,24]. 그러나 이들 기법의 경우에도 목적 코드의 수행 중 캐시미스 등의 이유로 인하여 메모리 참조가 늦어짐으로 인하여 명령어 파이프라인을 정지시키는 문제를 해결할 수는 없다.

VLIW 프로세서에서 루프를 빠르게 처리하는 기법은 주로 컴파일러 측면에서 다양한 기법들이 연구되고 있다[12,13,24,25]. 이들 방법은 루프를 몇 차례 전개하고, 그 가운데 존재하는 병렬성을 루프내 ILP로 활용하는 루프 전개(loop unrolling) 기법[12,24]과 루프 전개 기법에서 발생하는 각 반복의 마지막과 다음 반복의 시작간의 지연시간을 줄이기 위하여 루프를 대상으

로 하여 여러 개의 반복들이 중첩 수행될 수 있도록 루프를 재구성하는 루프변환 기법인 소프트웨어 파이프라이닝(software pipelining) 기법[13,24]이 대표적이다. 그러나 하드웨어 구조의 측면에서 루프를 빠르게 처리하는 구조에 관한 연구는 별로 수행된 바가 없다.

### 2.2 슈퍼스칼라 프로세서

실행 중에 명령어를 스케쥴하는 슈퍼스칼라 프로세서 구조의 경우에는 목적 코드가 VLIW 프로세서 구조와는 달리 순수한 명령어로만 구성된다. 따라서 VLIW 프로세서 구조와 비교하여 목적 코드의 길이가 짧으므로 캐시의 이용률이 좋다. 뿐만 아니라 이론적으로는 응용 프로그램에서 추출할 수 있는 ILP의 최대 크기에 맞추어 연산처리기의 수를 결정하고, 동적 스케쥴링 장치로 하여금 응용 프로그램 전체로부터 ILP를 찾아내어 이들을 연산처리기에 할당하도록 한다면 가장 좋은 성능을 발휘할 수 있다. 그러나 실제로는 동적 스케쥴러가 찾아낼 수 있는 ILP의 크기가 한번에 찾아낼 수 있는 병렬성의 길이 또는 윈도 길이에 따라 결정되므로 윈도 길이를 늘림에 따른 하드웨어의 구성 비용은 급격히 증가한다. 따라서 슈퍼스칼라 프로세서 구조에서 연산 처리기의 수를 늘리는 것은 결국 동적 스케쥴러를 구성하는 비용이 증가하는 단점이 있다[14-19]. 또한 이 구조의 경우에도 하드웨어 구조 측면에서 루프를 빠르게 처리하는 기법에 대한 연구보다는 VLIW 구조의 경우와 같이 루프 전개 기법[12,24]과 소프트웨어 파이프라이닝 기법[13,24]과 같이 컴파일러 측면에서 더 많은 연구가 진행되고 있다.

### 2.3 하이브리드(Hybrid) 프로세서

#### 2.3.1 SVLIW 프로세서

SVLW 프로세서 구조는 VLIW 프로세서에 슈퍼스칼라 프로세서의 동적 스케쥴러의 일부분을 추가한 구조로 컴파일러는 응용 프로그램 전체로부터 ILP를 찾아내어 긴명령어로 구성된 목적 코드를 구성한다. 한편, 긴명령어간의 자원충돌이나 자료종속성은 실행 중 스케쥴러가 처리하도록 한다. 따라서 SVLIW 구조에서는 LNOP 명령어가 목적 코드에 삽입되지 않으므로 목적 코드의 길이가 VLIW 프로세서 구조용 목적 코드에 비하여 크기가 작아진다. 그러나 SVLIW 구조용 목적 코드에서도 응용 프로그램내의 ILP가 충분하지 않은 경우에는 NOP이 많이 존재할 가능성이 있다.

또한 VLIW 프로세서 구조에서 캐시미스에 따라 명령어를 메모리로부터 참조하는 경우에 메모리 사이클 길이만큼 명령어 파이프라인을 중지시킨다. 그러나 SVLIW 프로세서 구조에서는 캐시미스가 발생하더라도 명령어 스케쥴러가 명령어 파이프라인을 진행시킬 수 있다[23,27]. 따라서 캐스미스 기간 중에도 자원충돌이나 레지스터 파일의 자료종속성 문제가 해소될 수 있으므로 응용 프로그램의 실행사이클이 단축될 수 있는 장점이 있다.

한편, SVLIW 프로세서 구조에서도 긴명령어를 구성할 때 자료종속성이 없는 명령어들을 긴명령어로 구성하기 때문에 긴명령어 내에 NOP이 삽입되는 것을 방지할 수 없다. 뿐만 아니라 하나의 긴명령어의 실행이 종료되어야 다음 긴명령어가 실행되므로 루프의 경우에 루프의 마지막에 실행되는 긴명령어와 루프의 다음번 반복이 시작되는 첫 번째 긴명령어 간에 자료종속성이나 자원충돌이 존재하지 않더라도 순차적으로 실행되어야 하므로 루프의 처리는 전통적인 VLIW 구조와 그 성능이 동일하다.

#### 2.3.2 VFPE 프로세서

VFPE 프로세서 구조[20]는 VLIW 프로세서와 슈퍼스칼라 프로세서 구조의 특징을 혼합한 구조이다. 즉, VFPE 프로세서는 SVLIW 프로세서와 같이 긴명령어가 한번에 하나씩 인출되고, 인출된 긴명령어는 각 단위명령어로 나뉘어 단위명령어가 수행될 연산처리기의 명령어 큐에 저장된다. 따라서 각각의 연산처리기는 자신의 큐에 적재된 단위명령어를 다른 연산처리기의 상태와 관계없이 수행하므로 자료의존성이 존재하는 여러 개의 단위명령어들간의 동기를 위하여 동기제어장치가 필요하다. 이러한 동기제어장치의 도움으로 자료의존성이 있는 단위명령어를 하나의 긴명령어로 구성하더라도 연산처리기에서 실행될 때에는 각 단위명령어들간의 계산순서가 정확하게 유지된다. 여기서 단위명령어란 긴명령어를 구성하는 각각의 명령어를 지칭한다. 따라서 본 논문에서는 명령어와 단위명령어를 혼용하여 사용하기로 한다. 다만 혼동을 방지할 필요가 있을 때에는 이를 명확히 구분하도록 하였다.

VFPE 프로세서는 자료의존성을 지닌 인접한 명령어들을 하나의 긴명령어로 구성하므로 생성되는 목적 코드내의 NOP이 SVLIW 프로세서용 목적 코드에 비하여 더 줄 수 있다. 이러한 특징은 루프의 경우에 어

떤 반복(iteration)의 마지막 긴명령어와 다음번 반복의 첫 번째 긴명령어를 구성하는 단위명령어들이 사용하는 연산처리기가 상이할 때 이들을 하나의 긴명령어로 구성하도록 허용함으로써 빠른 루프처리가 가능한 장점이 있다.

그러나 VFPE 프로세서는 그 구성이 매우 제한적이다. 즉 이 구조에서는 목적 코드를 생성하는 과정에서 응용 프로그램을 구성하는 명령어들을 프로세서를 구성하는 연산처리기별로 유형을 구분하고 각 연산처리기별로 쓰레드(thread)를 만든 후에 연산처리기별 각 쓰레드의 명령어를 취하여 하나의 긴명령어를 구성한다. 따라서 PASC 프로세서를 구성하는 연산처리기는 종류별로 하나만 존재하여야 한다. 참고문헌 [20]에서는 BU(Branch Unit), TU(Transfer Unit)와 AU(Arithmetic Logic Uint)의 세 가지 연산처리기로 프로세서를 구성하고 있다. 따라서 VFPE 구조는 프로세서를 구성하는 연산처리기의 수를 VLIW 구조나 SVLIW 구조와 같이 다양하게 구성할 수 없다.

VLIW 프로세서나 SVLIW 프로세서용 코드는 긴명령어 형태를 지니며 인출, 분석, 실행단계 및 쓰기 단계의 모든 파이프라인 과정을 지나는 동안 긴명령어를 구성하는 단위명령어들이 동기적으로 처리된다. 이에 반하여 VFPE 구조에서는 긴명령어 단위로 인출 및 분석작업이 이루어진 후, 단위명령어로 분해되어 이들이 실행될 연산처리기의 명령어 큐로 전달된다. 명령어 큐에 적재된 명령어는 다른 연산처리기의 실행여부에 관계없이 결정되므로 비록 하나의 긴명령어를 구성하였던 단위명령어라고 하더라도 서로 다른 사이클에 실행될 수 있다.

본 논문에서 제안하는 PASC 프로세서에서도 인출단계가 종료되는 시점에서 인출된 긴명령어를 단위명령어로 구분하여 명령어 큐에 삽입하도록 하고 분석단계에서는 각각의 개별적인 명령어 큐로부터 명령어를 인출받아 이를 분석하여 실행유니트에 할당하여 실행할 것인지 또는 다음 사이클까지 기다릴 것인지를 결정하도록 하였다. 이때 명령어가 실행유니트로 제공될 수 있는지 여부는 다른 연산처리기에서 수행중인 명령어와의 자료종속관계나 자원충돌 여부에 의하여 결정되므로 이 과정을 분석단계에서 수행하도록 하기 위하여 명령어에 다른 명령어와의 관계를 나타내는 정보를 포함하도록 목적 코드를 구성하여 한 사이클의 분석단계에서 명령어의 분석과 이 명령어의 스케쥴링 여부를 결정할 수 있도록 하였다.

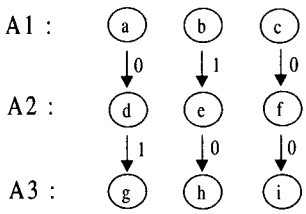
## 3. PASC 프로그램

### 3.1 프로세서 구조에 따른 명령어 실행 패턴

본 논문에서는 프로세서를 매 사이클마다 인출(fetch), 분석(decode), 실행(execute), 쓰기(write back) 단계를 수행하는 파이프라인 구조의 프로세서라고 가정한다. 정수명령어의 경우에는 실행단계가 한 사이클을 차지하며, 실수명령어의 경우에는 명령어에 따라서 여러 사이클이 필요하다고 가정하였다. 또한 명령어 종류에 관계없이 실행단계가 종료하자마자 실행단계의 계산 결과를 다음 명령어의 실행단계에서 사용할 수 있도록 by-pass 선로[26]가 존재한다고 가정하였다. By-pass 선로가 존재하는 구조는 쓰기단계를 거치지 않고 실행단계의 계산 결과를 다음 사이클에서 이용할 수 있으므로 실행단계가 한 사이클을 차지하는 정수명령어의 경우에는 명령어간의 자료종속성이 제거된다. 그러나 실수명령어의 경우에 실행단계가 $m$사이클을 점유하는 두 개의 명령어간에 자료종속성이 존재하는 경우에는 두 번째 명령어는 첫 번째 명령어가 실행된 후 $m-1$ 사이클만큼 지연된 후에야 실행단계에 진입할 수 있다[27]. 본 논문에서는 두 개의 명령어간에서 $k$ 사이클이 지연된 후에 시작되는 경우를 명령어 그래프 $I_i \xrightarrow{k} I_j$로 표시하기로 한다. 이때 $k$는 $m-1$ 사이클이다. 여기서 $I_i$와 $I_j$는 같은 연산처리기의 실행단계를 거쳐야 하는 연속한 두 개의 명령어이다.

(그림 1)은 9개의 명령어로 구성된 명령어 그래프의 한 예로 3개의 연산처리기로 구성된 ILP 프로세서용으로 생성된 목적 코드의 형태이다. (그림 1)에서 명령어 e는 b가 실행단계로 진입한 후 두 사이클 후에야 실행단계로 진입할 수 있음을 의미하며, 명령어 g도 명령어 d가 실행단계로 진입한 후 두 사이클 후에야 실행단계로 진입할 수 있다. 나머지 명령어들은 한 사이클이 지난 다음 번 사이클에 실행단계로 진입할 수 있음을 보여준다. 따라서 (그림 1)의 명령어 그래프를 VLIW용 목적 코드로 변환하면 (그림 2) (a)와 같다. 즉, 명령어 그래프에서 A1과 A2 사이에는 명령어 b와 e간의 자료종속성으로 인하여 하나의 LNOP(long NOP) 명령어가 삽입되며, A2와 A3 사이에도 하나의 LNOP

이 삽입된다. 한편 (그림 2) (a)에 보인 목적 코드의 실행과정은 (그림 3) (a)와 같으며, 계산을 완료할 때까지 총 8 사이클이 필요하다.




(그림 1) 명령어 그래프

(그림 1)에 보인 명령어 그래프를 토대로 SVLIW용으로 생성된 목적 코드는 (그림 2) (b)와 같이 VLIW용 목적 코드에서 LNOP이 제거된 형태이며, LNOP은 분석단계에서 자원충돌 및 자료종속성 존재여부에 따라 삽입여부가 결정된다. 따라서 실행 패턴은 (그림 3) (b)와 같다. 즉, A1과 A2를 구성하는 명령어 b와 e간에 자료종속성으로 인한 자원충돌이 발생하므로 e가 1사이클 후에 실행단계로 진입할 수밖에 없으므로 명령어 d와 f 명령어들도 1사이클 후에 실행단계로 진입한다. 즉, 하나의 명령어 e가 야기하는 자원충돌로 인하여 긴명령어 A2의 실행단계 진입이 늦어지게 된다. 마찬가지로 A3도 A2가 실행단계에 진입한 후 한 사이클이 지연된 후에야 실행단계로 진입할 수 있다. 따라서 SVLIW 구조에서도 계산이 종료되기 위해서 필요한 사이클의 수는 VLIW의 경우와 마찬가지로 총 8사이클이다.

| A1 | a | b | c |
|---|---|---|---|
| A1-1 | 0 | 0 | 0 |
| A2 | d | e | f |
| A2-1 | 0 | 0 | 0 |
| A3 | g | h | i |

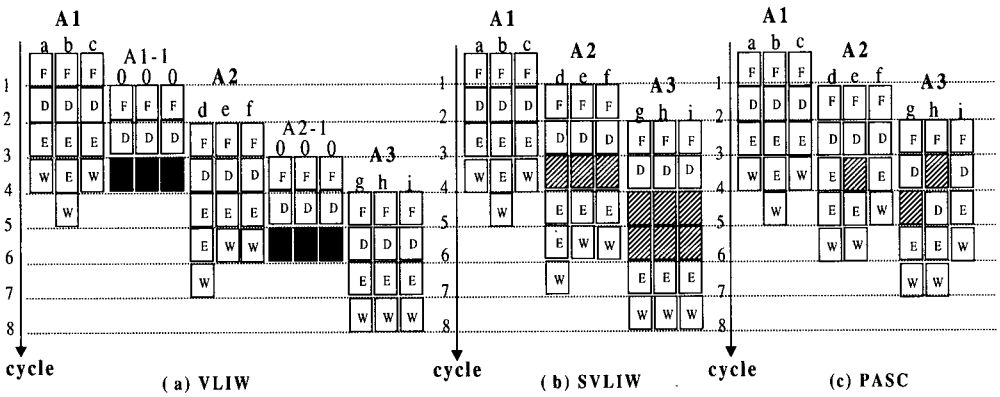
(a) VLIW code

| A1 | a | b | c |
|---|---|---|---|
| A2 | d | e | f |
| A3 | g | h | i |

(b) SVLIW code

(그림 2) 목적 코드의 형태

만일 긴명령어를 구성하는 명령어가 독립적으로 그 실행 순서가 결정될 수 있는 프로세서 구조- 예를 들면 본 논문에서 제안하려는 PASC 구조-에서 실행된다고 가정하자. (그림 3) (c)에 보인 것과 같이 긴명령어 A1은 VLIW나 SVLIW의 경우와 동일하게 실행될 것이다. 이어서 긴명령어 A2는 SVLIW에서와 같이 다음 사이클에서 인출된 후 분석되어 실행단계로의 진입여부가 결정될 것이다. 그런데 SVLIW 에서는 명령어 e와 명령어 b가 실행단계를 공유할 수 없으므로 A2는 한 사이클 지연된 후에야 실행단계로 진입된다. 그러나 PASC구조에서는 각 명령어가 실행단계로 진입할 수 있는지 여부를 독립적으로 결정할 수 있으므로 비록 명령어 e는 명령어 b가 실행단계를 빠져 나오기까지 대기하더라도 명령어 d와 f는 (그림 3) (c)와 같이 실행단계로 진입할 수 있다. 마찬가지로 긴명령어 A3를 구성하는 명령어들도 앞서의 명령어의 실행에 지장을 주지 않으면 실행단계로 진입할 수 있으므로 총 7




(그림 3) 실행 이미지

사이클만에 계산을 종료할 수 있다. 따라서 PASC는 VLIW나 SVLIW 구조에 비하여 한 사이클 줄어드는 효과를 얻을 수 있다. 그러나 만일 명령어 e와 i간에 자료종속성이 있다면 명령어 e가 실행단계를 빠져 나올 때까지 명령어 i를 실행하지 않도록 해야한다. 즉, 긴명령어를 구성하는 명령어간에 자원충돌관계나 자료종속성이 존재하는 경우에는 긴명령어간의 동기나 긴명령어내의 명령어간의 동기를 유지할 수 있는 방안이 강구되어야 한다. 본 논문에서 제안하려는 PASC용 목적 코드를 구성하는 각 명령어에도 명령어간의 동기를 위한 정보가 포함된다.

### 3.2 PASC 프로세서용 긴명령어

PASC 프로세서를 위하여 생성되는 목적 코드에서 긴명령어를 구성하는 명령어간의 동기를 위한 정보로는 선행종속성(pre dependencies) 정보와 후행종속성(post dependencies) 정보로 구성된다. 선행종속성 정보는 하나의 명령어와 자료의존관계가 있는 먼저 수행된 명령어들이 어떠한 연산처리기에 할당되어 있는지를 나타내고 후행종속성 정보는 현재의 명령어와 자료종속관계가 존재하는 다음에 수행될 명령어들이 어떠한 연산처리기에 할당되어 있는가에 대한 정보를 지니고 있다. 따라서 각 명령어들은 이와 같은 자신의 선행종속성 정보를 토대로 명령어를 실행단계로 진행시킬 것인가의 여부를 판단하고 실행의 진입이 가능한 경우 후행종속성 정보를 이용하여 다음에 수행될 명령어들의 실행여부를 결정한다.

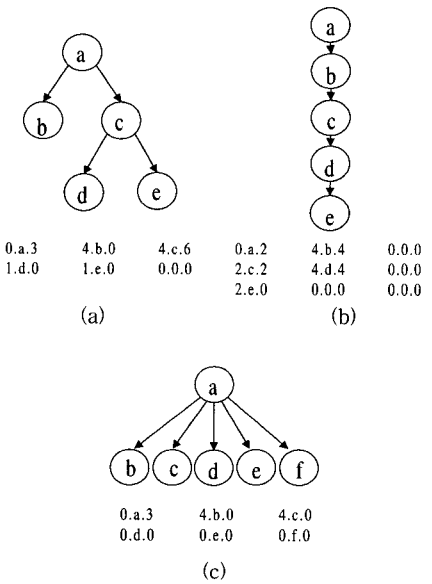
선행종속성 정보는 현재 수행되고 있는 자신의 연산처리기의 명령어가 앞서 수행된 다른 연산처리기에 존재하는 명령어들과의 자료종속성 관계여부를 알고 있어야 하므로 연산처리기 수만큼의 비트가 필요하다. 또한 후행종속성을 표현하는 정보도 다른 연산처리기에서 다음에 수행될 명령어들과의 종속성 여부를 표현해야 하므로 총 연산처리기의 수만큼 비트가 필요하다. 이와 같이 연산처리기의 수만큼 비트 정보를 갖도록 긴명령어를 구성한 것은 스케줄러에서 각 비트 패턴을 한 비트만 세트하여 하나의 연산처리기에 대하여 자료종속성이 존재하는 다른 모든 연산처리기의 자료종속성을 제거시켜 주기 위함이다. 본 논문에서는 긴명령어를 구성하는 단위명령어를

*nn.OP.rr*

형태로 표현하기로 한다. 여기서 *OP*는 명령어 및 연산자를 의미하며, *nn*과 *rr*은 각각 선행종속성 정보와 후행종속성 정보를 의미한다.

본 논문에서는 긴명령어를 구성할 때 단위명령어간의 자료종속성의 길이가 1 이하인 명령어들로 구성하였다. 그 이유는 목적 코드내 명령어간에 존재하는 자료종속성이 대부분 이웃한 명령어들 사이에 존재하므로 이렇게 제한함으로써 대부분의 자료종속성을 제거할 수 있을 뿐 아니라, 자료종속성의 길이가 긴 명령어들로 동일한 긴명령어를 구성할 경우 실행 시 발생하는 복잡성을 최소화할 수 있기 때문이다. 또한, 이와 같이 구성으로 자료종속성은 위에 보인 명령어 형식의 선행종속성 및 후행종속성을 표시하는 영역을 단위명령어당 각각 하나의 비트로 구성할 수 있다.

(그림 4)는 여러 가지 형태의 명령어 그래프에 대하여 3개의 연산처리기로 구성된 PASC 프로세서를 위한 PASC형 긴명령어의 구성방법이다.




(그림 4) 여러 형태의 PASC 프로세서용 긴명령어 코드

(그림 4) (a)에서 명령어 a, b, c는 하나의 긴명령어로 구성될 수 있으며, 첫 번째 긴명령어에서 a의 후행종속성 정보는 a가 b와 c에 자료종속관계가 있다는 정보를 유지한다. 이때 명령어 a는 명령어 b, c와 자료종속관계임을 나타내기 위하여 비트 패턴이 011로 세트되므로 후행종속성 정보는 3이며, 다른 명령어의 실행

여부에 자신의 실행이 영향을 받지 않으므로 선행종속성 정보는 0이다. 그러나 명령어 그래프로부터 명령어 b와 c는 a의 실행단계가 종료된 후에 실행되어야 하므로 이들 명령어는 각각 a가 수행되는 첫 번째 연산처리기의 실행단계가 종료될 때까지 실행이 지연되어야 함을 나타내기 위하여 비트 패턴이 100으로 세트되며, 선행종속성 정보는 각각 4이다. 그러므로 (그림 4) (a)의 명령어 그래프는 아래와 같이

| | | |
|---|---|---|
| 0.a.3 | 4.b.0 | 4.c.0 |
| 1.d.0 | 1.e.0 | 0.0.0 |

의 형태로 구성된다.

(그림 4) (b)는 극단적으로 모든 명령어들이 자료의 존관계를 가지고 있어서 순차적으로 수행되어야 하는 프로그램이다. 첫 번째 긴명령어를 a, b, c로 구성하고, 두 번째 긴명령어를 d, e로 구성할 수 있으나 본 논문에서는 긴명령어를 구성하는 명령어간 자료종속성의 길이를 1 이하로 제한하였으므로 긴명령어 코드가 아래와 같이 구성된다.

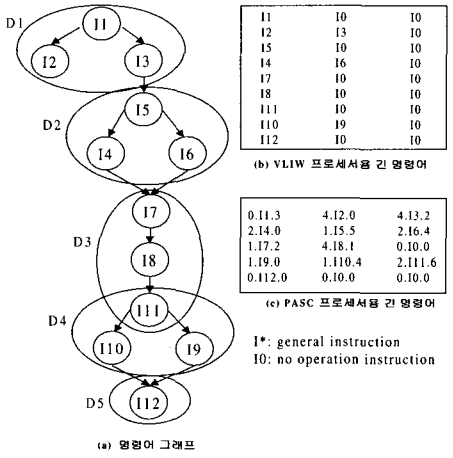
| | | |
|---|---|---|
| 0.a.2 | 4.b.4 | 0.0.0 |
| 2.c.2 | 4.d.4 | 0.0.0 |
| 2.e.0 | 0.0.0 | 0.0.0 |

(그림 4) (c)는 동시에 수행 가능한 명령어들의 수가 연산처리기의 수를 초과할 경우의 예를 보여준다. 예와 같이 첫 번째 긴명령어를 구성하는 명령어와 동일한 레벨의 명령어들로 두 번째 긴명령어가 구성될 경우에는 첫 번째 긴명령어와 두 번째 긴명령어간에는 선행종속성이 존재하지 않으므로 선행종속성 정보는 0이며, 그 구성형태는 아래와 같다.

| | | |
|---|---|---|
| 0.a.3 | 4.b.0 | 4.c.0 |
| 0.d.0 | 0.e.0 | 0.f.0 |

(그림 5)는 명령어의 자료종속성이 (그림 5) (a)와 같이 표현될 때 각 명령어들이 프로세서별로 어떻게 긴명령어로 구성되는가를 보여주고 있다. 연산처리기가 세 개인 PASC 프로세서에서 긴명령어는 (그림 5) (a)에서 타원으로 묶여있는 명령어의 그룹으로 구성된다. 즉, 그룹 D1에 포함된 I1 명령어와 I2, I3 명령어는 서로 한 레벨 차이로 자료종속성이 존재한다. 긴명령어 생성시 I2와 I3 명령어는 I1과 자료종속성이 있기 때문에 VLIW 프로세서에서는 I1과 I2, I3 명령어를 두 개의 그룹으로 구분지어서 긴명령어를 생성해야 하나 PASC 프로세서에서는 하나의 긴명령어로 구성할 수 있다. 그 이유는 각 명령어가 선행종속성 정보와 후행종속성 정보를 이용하여 동일한 긴명령어를 구성하는 다른 명령어간의 관계를 표현할 수 있기 때문이다.

한편 그룹 D3에 포함되어 있는 I7, I8, I11의 명령어는 두 레벨 이상의 자료종속성이 있기 때문에 두 레벨의 차이가 있는 I11 명령어를 다음 그룹인 D4로 넘겨주고 그룹 D3의 I11 명령어 대신 빈 명령어를 나타내는 I0을 삽입시켜 준다. 명령어 I1, I2, I3로 구성된 D1과 I4, I5, I6로 구성된 D2간에는 각각의 긴명령어에 존재하는 I3 명령어와 I5 명령어가 자료종속관계에 있다. 이처럼 긴명령어간 존재하는 자료종속성은 I3의 후행종속성 정보와 I5의 선행종속성 정보를 이용하여 즉, I3가 수행을 완료한 후 자신의 후행종속성 정보를 이용하여 I5의 선행종속성 정보를 해제시킴으로써 I5가 실행 가능토록 해준다. 이와 같은 방식으로 구성된 자료종속성 정보는 각각의 연산처리기에 결합된 스케쥴러에 의해서 스케쥴 됨으로써 종속성 정보를 포함하고 있는 긴명령어들의 자료종속성을 실시간에 해결할 수 있다.




(그림 5) 두 가지 프로세서의 긴 명령어 코드

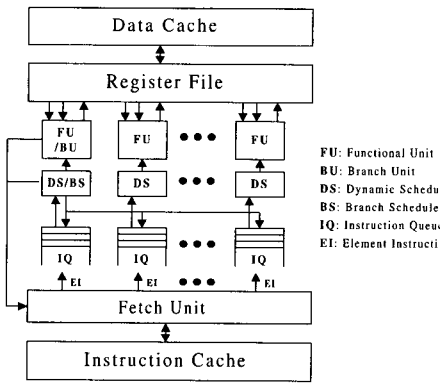
(그림 5) (b)와 (c)는 각각 VLIW 프로세서용 긴명령어 코드와 PASC 프로세서용 긴명령어 코드를 나타낸 것이다. (그림 5) (b)와 (c)에서 PASC 프로세서용 긴명령어 코드는 자료종속성 정보를 명령어 코드 내에 표현하고 있기 때문에 VLIW 프로세서용 긴명령어 코

드에서 빈 명령어로 채워진 부분을 훨씬 더 축소한 긴 명령어 코드를 만들 수 있다.

## 4. PASC 프로세서

### 4.1 PASC 프로세서 구조

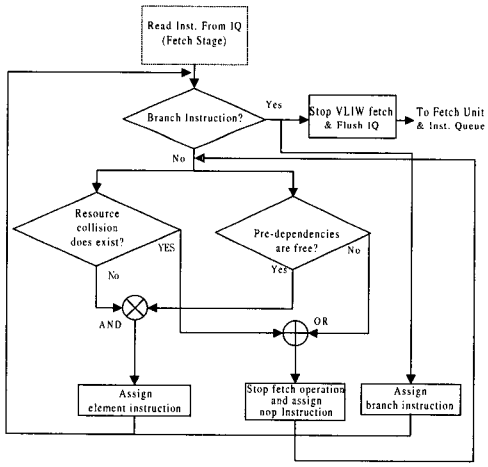
(그림 6)은 PASC 프로세서의 구조를 나타낸 것이다. (그림 6)에서 PASC 프로세서의 명령어 인출기는 명령어 캐시로부터 매 사이클마다 하나의 긴명령어를 인출하여 이를 명령어(EI)별로 구분하여 연산처리기별로 구분된 명령어 큐(IQ)에 삽입한다. 스케쥴러(DS)는 명령어 큐로부터 명령어를 하나 인출하여 명령어에 포함된 자료종속관계를 가지고 이 명령어를 실행할 것인지 아니면 한 사이클 동안 기다릴 것인지를 결정하고, 그 결과에 따라 명령어를 연산처리기(FU)로 진행시키거나 기다리도록 한다.



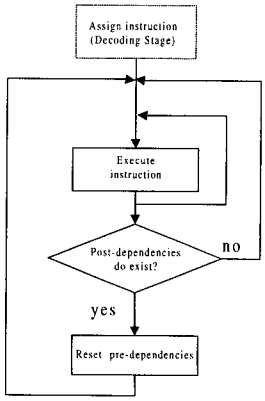

(그림 6) PASC 프로세서 구조

명령어 분석단계과 실행단계에서 각각의 독립적인 스케쥴러가 명령어 큐로부터 읽은 명령어의 자료종속성 정보를 처리하는 과정은 (그림 7)과 같다. (그림 7) (a)는 분석단계에서 각각의 독립적인 스케쥴러가 명령어 큐로부터 읽은 명령어의 선행종속성 정보를 처리하는 과정과 분기문의 경우 명령어 인출기 및 명령어 큐를 제어하는 과정을 나타낸 것이다. 또한, (그림 7) (b)는 실행단계에서 명령어를 수행한 후 후행종속성 정보를 이용하여 다음에 수행될 명령어의 선행종속성 정보를 해제하여 명령어의 실행여부를 결정하는 제어 흐름을 표현한 것이다.

만일 분기문 예측법 및 분기문 타겟 버퍼 등을 이용하여 분기문을 처리하는 경우에는 분기문으로 인해서 발생되는 지연을 해결할 수 있다. 이 경우 새로운 긴명령어를 인출하기 위해 지연되는 페널티(penalty)가 모두 제거되므로 PASC 프로세서에서는 자료종속성이 존재하는 명령어를 동시에 인출할 수 있기 때문에 루프의 경우 매 반복 시에 루프의 첫 긴명령어를 개별적으로 각각의 명령어 큐에 미리 적재할 수 있다. 즉, 기존의 VLIW 프로세서에서는 각각의 연산처리기에 할당된 명령어들이 서로 다른 사이클에 실행되어야 한다. 그러나 PASC 프로세서에서는 루프의 마지막 긴명령어의 일부와 첫 번째 긴명령어의 일부가 동시에 수행될 수 있으므로 매 루프시 한 사이클씩 수행 시간이 단축되는 효과가 있다.




(a) Decoding stage




(b) Execution stage

(그림 7) 명령어 흐름 제어

### 4.2 루프와 PASC 구조

파이프라인의 실행 이미지를 고찰하기 위하여, 본 절에서는 순차적인 MIPS 코드를 입력으로 받아서 생성된 목적 코드가 각 프로세서 구조별 명령어 파이프라인 상에서 어떠한 실행 흐름을 보이는 지를 비교·분석하였다.

```
$32:  lw     $14, 20($sp)
      mul    $15, $14, 4
      addu   $24, $sp, 0
      addu   $25, $15, $24
      sw     $14, 0($25)
      lw     $8, 20($sp)
      addu   $9, $8, 1
      sw     $9, 20($sp)
      blt    $9, 5, $32
```

위의 순차 프로그램은 0부터 4까지의 수를 배열에 초기화하는 프로그램으로서 위의 코드를 VLIW 프로세서용 코드와 PASC 프로세서용 코드로 각각 표현하면 각각 (그림 8) (a) 및 (b)와 같다.

```
$32:  lw      addu    lw
      mul     nop     addu
      addu    nop     sw
      sw      nop     nop
      nop     blt     nop
```

(a) VLIW code

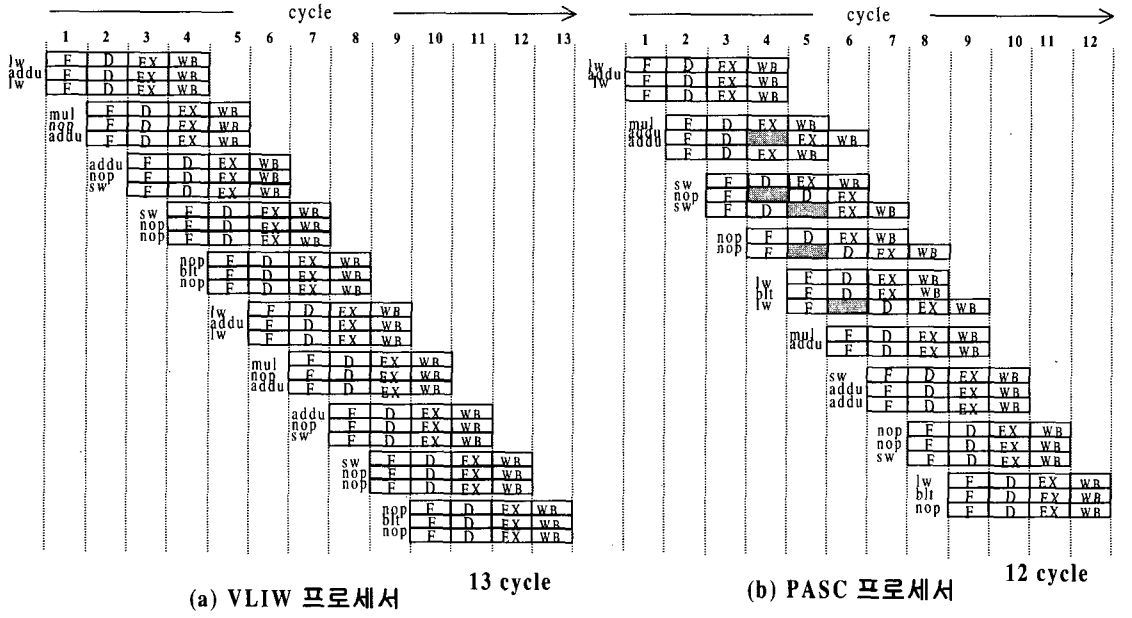
```
$32:  0.lw.0     0.addu.0    0.lw.0
      0.mul.2    4.addu.1    0.addu.0
      0.sw.0     0.nop.0     2.sw.2
      0.nop.0    1.blt.0     0.nop.0
```

(b) PASC code

(그림 8) PASC 프로세서와 VLIW 프로세서용 코드

또한, (그림 9) (a)와 (b)는 각각의 프로세서 구조에서 수행되는 파이프라인 실행 이미지를 첫 번째 반복까지 나타낸 것이다. (그림 9) (a)와 (b)에서 루프의 첫 반복 시 PASC 프로세서가 VLIW 프로세서보다 한 사이클 단축된 파이프라인 실행 이미지를 보여준다. (그림 9) (b)에서 검은 부분은 파이프라인의 실행 유니트가 정지(stall)된 상태를 나타낸다.

본 프로세서의 연산처리기들은 자료종속성이 존재



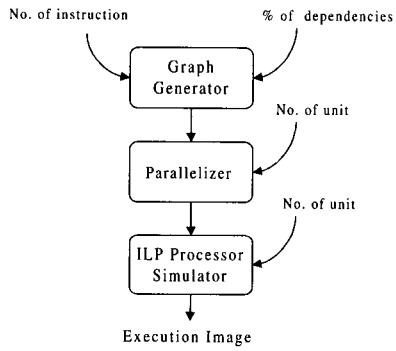

(그림 9) 명령어 실행 이미지

하는 긴명령어를 동시에 인출하여 명령어 큐에 적재하기 때문에 VLIW 프로세서에서는 서로 다른 사이클에서 수행되어야만 하는 마지막 긴명령어 코드의 일부와 처음의 긴명령어 코드의 일부가 동시에 수행될 수 있다. (그림 9)의 (a)와 (b)에서 네 번째와 다섯 번째의 파이프라인 실행 이미지를 비교해 볼 때, VLIW 프로세서에서는 모든 유니트들이 동시에 수행되어야 하는 특징이 있어서 마지막 긴명령어가 수행된 후에야 첫 번째 긴명령어가 수행되는 반면, PASC 프로세서에서는 루프의 첫 번째 긴명령어와 마지막 긴명령어가 동시에 실행되는 파이프라인 이미지 형태가 가능하다. 그러므로 위의 프로그램에서 VLIW 프로세서는 한 번 루프 반복 시 13 사이클이 걸리는 반면 본 프로세서는 12 사이클만에 첫 반복 루프를 수행한다. 결국 위의 프로그램은 5번의 반복이 있어야 하므로 PASC 프로세서는 VLIW 프로세서와 비교하여 총 5 사이클의 실행 시간을 단축할 수 있다.

## 5. 실험 및 고찰

### 5.1 시뮬레이션 시스템 및 실험 환경

제안한 PASC 프로세서의 동작 및 성능을 평가하기 위하여, 각 ILP 프로세서들의 실행 모델을 확인할 수 있는 시뮬레이션 시스템을 구현하였다. 시뮬레이션 시스템은 임의로 생성된 자료종속성 그래프를 입력으로 받아 가상적으로 명령어 어셈블리 코드를 생성하고 이 코드는 각각의 구조를 위하여 구현된 컴파일러를 경유하여 긴명령어 코드를 생성한 후 시뮬레이터에 직접 입력이 된다. 특히 루프에서 병렬성을 추출하는 기본 영역인 기본 블록(basic block)이 작음으로 인해서 추출 가능한 병렬성의 정도가 적어지는 문제점을 해결하기 위하여 루프 몸체를 구성하는 명령어들의 수를 변경하도록 하였다. 또한, 루프 내에 포함하고 있는 명령어들을 다양한 형태로 구성하기 위하여 루프의 횟수, 명령어간 자료종속성을 다양하게 발생시켜서 시뮬레이션을 하였다. 시뮬레이션 시스템의 구성은 (그림 10)과 같다.




(그림 10) 시뮬레이션 시스템

(그림 11)은 15개의 명령어와 80%의 자료종속성을 가지고 있다고 가정하고 연산처리기의 수를 3개로 설정하였을 경우 시뮬레이션 실행 예를 보여주고 있다. (그림 11) (a)는 각 명령어 상호간에 자료종속성을 표

```
I1: I2 I3 I4 I5 I6 I7 I8 I11 I12 I14 I15
I2: I3 I4 I5 I6 I7 I8 I9 I10 I11 I12 I13 I14 I15
I3: I4 I5 I6 I7 I8 I10 I11 I13 I14 I15
I4: I5 I6 I7 I8 I9 I10 I11 I12 I13 I14 I15
I5: I6 I7 I8 I9 I10 I12 I13 I14 I15
I6: I7 I9 I10 I11 I12 I13 I14 I15
I7: I8 I9 I11 I12 I13 I14 I15
I8: I9 I10 I11 I12 I13 I15
I9: I10 I11 I12 I13 I15
I10: I12 I14 I15
I11: I12 I13 I14 I15
I12: I13 I14 I15
I13: I14 I15
I14:
I15:
```

(a) 명령어 수행순서

```
0.I1.2   4.I2.4   0.0.0
2.I3.2   4.I4.4   0.0.0
2.I5.2   4.I6.4   0.0.0
2.I7.2   4.I8.4   0.0.0
2.I9.3   4.I10.4  4.I11.4
3.I12.2  4.I13.4  0.0.0
2.I14.0  0.I15.0  0.0.0
```

(b) 긴 명령어 코드

```
I1      nop     0
nop     I2      nop
I3      nop     0
nop     I4      nop
I5      nop     0
nop     I6      nop
I7      nop     0
nop     I8      nop
I9      nop     nop
nop     I10     I11
I12     nop     0
nop     I13     nop
I14     I15     0
I1      nop     0
nop     I2      nop
I3      nop     0
nop     I4      nop
I5      nop     0
```

(c) 실행 이미지

(그림 11) 실행 예

현한 것으로 ';'를 기준으로 오른쪽에 있는 명령어들은 왼쪽에 있는 명령어가 수행한 후에야 실행할 수 있는 자료종속관계를 표현하고 있다. 여기서 자료종속성 80%는 각 명령어간에 자료종속성이 존재하는 쌍의 수가 전체 명령어의 쌍에 대한 비를 의미한다. (그림 11)(b)는 3.2절에서 설명한 기법을 적용하여 각 명령어에 선행종속성 정보와 후행종속성 정보를 삽입하여 구성한 긴명령어 코드 형태를 표현하고 있다. (그림 11)(c)는 각 명령어가 실제 실행단계에서 수행되는 이미지를 표현하고 있다.

실험에서는 PASC용 프로세서용 시뮬레이터가 VLIW 및 SVLIW용 시뮬레이터로 동작할 수 있도록 하기 위하여 SVLIW와 VLIW 긴명령어 코드는 가상적으로 선행종속성 정보와 후행종속성 정보를 추가하여 목적코드를 구성하였다. 또한 캐시 효율 및 파이프라인 실행이미지 측면을 고려하기 위하여 목적 코드의 크기와 루프의 반복 회수를 성능 평가 요소로 설정하여 기존의 VLIW 프로세서, SVLIW 프로세서 및 제안한 PASC 프로세서를 비교·분석하였다. 이를 위하여 연산처리기의 수를 3개로 고정시킨 상태에서 입력되는 그래프의 명령어 수와 자료종속성 및 루프의 반복회수를 변화시키면서 그래프를 생성하고 이를 입력으로 하여 시뮬레이션을 수행하였다.

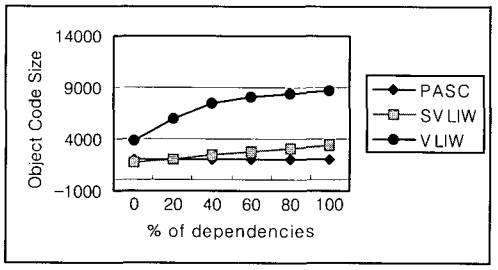
### 5.2 실험 결과

(그림 12)는 연산처리기의 수를 변화시키면서 VLIW 및 SVLIW 프로세서용 목적 코드와 PASC 프로세서용 목적 코드의 크기를 비교한 것이다. PASC의 경우에는 선행종속성 정보와 후행종속성 정보로 구성된 자료종속성 정보를 목적코드에 포함시켜 비교하였다. (그림 12) (a), (b) 및 (c)는 각각 연산처리기의 수가 2개, 3개 및 4개일때로 자료종속성의 비율을 20%에서 100%까지 변화시키면서 각각의 프로세서에 구조를 위하여 생성한 목적 코드의 크기를 측정한 것이다. 그 결과 PASC 프로세서가 생성하는 목적 코드의 크기는 VLIW 프로세서와 SVLIW 프로세서에서 발생시킨 목적 코드의 크기보다 작은 것을 알 수 있다. 이것은 PASC 프로세서용 목적코드가 자료종속성 정보를 포함하고 있더라도 VLIW 프로세서나 SVLIW 프로세서용 목적 코드내에 포함되는 NOP으로 인한 목적코드의 증가보다 그 비율이 낮기 때문인 것으로 보인다.

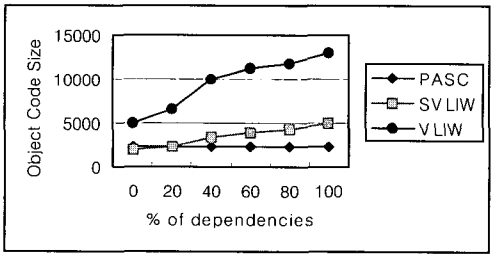
PASC 프로세서에서 목적 코드에 포함되는 자료종속성 정보는 자료종속성의 비율과 관계없이 연산처리기의 수($n$)의 함수로 결정된다. 즉, 단위명령어가 32비트 길이라고 하고 선행종속성 정보 및 후행종속성 정보가 각각 연산처리기의 수만큼의 비트가 필요하다고 하면 목적 코드내에 포함되는 자료종속성 정보가 $2n$이므로 그 비율은 $\frac{n}{16+n}$이다. 예를 들어 연산처리기가 두 개인 경우에 그 비율은 11%이며, 4개인 경우에는 20%가 된다. 따라서 PASC용 목적코드에서 자료종속성 정보가 차지하는 비율이 큰 편이나 (그림 12)에서 알 수 있듯이 SVLIW 구조나 VLIW 구조용 목적코드에 비해서는 그 크기가 매우 작다. 또한, PASC 프로세서의 경우에는 다른 프로세서의 경우와 달리 자료종속성의 비율이 증가해도 목적 코드의 크기는 증가하지 않는다. 한편, VLIW 프로세서나 SVLIW 프로세서는 연산처리기의 수가 증가할수록 그에 비례하여 명령어 병렬성을 추출하기 어렵고 자료의존성의 비율이 증가할수록 목적 코드내에 삽입되는 NOP의 수가 증가한다.

자료종속성이 20%일 경우, SVLIW 프로세서용 목적 코드의 크기가 PASC 프로세서용 목적 코드의 크기보다 작은 경우가 발생한다. 이는 자료종속성의 비율이 매우 낮은 경우 SVLIW용 목적 코드 내에 존재하는 NOP 명령어의 크기가 매우 적음으로 인하여 PASC용 명령어에 추가되는 자료종속성 정보의 크기가 제거되는 NOP이 차지하는 영역보다 크기 때문인 것으로 판단된다. 자료종속성이 20%이상인 경우에는 PASC 프로세서 구조가 SVLIW 구조보다도 목적 코드의 압축률이 더 좋은 것을 보여준다. 즉, PASC용 목적 코드는 단위명령어간의 자료종속성을 표시하는 정보가 긴명령어에 포함되더라도 SVLIW 구조용 목적 코드에 비하여 그 크기가 늘어나지 않는다. 따라서 PASC에서의 캐시효율도 SVLIW 구조에서 성취할 수 있는 캐시 효율과 유사할 것으로 보인다.

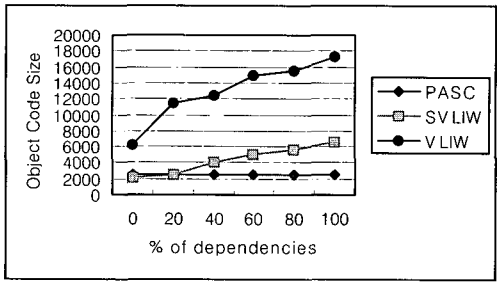
PASC 프로세서는 자료의존성이 있는 여러 개의 명령어를 동시에 인출할 수 있기 때문에 루프 내에 존재하는 명령어들에 대하여 파이프라인 실행사이클이 VLIW 및 SVLIW 프로세서에서의 실행사이클과 비교하여 매 사이클마다 한 사이클씩 줄일 수 있다. 이를 확인하기 위하여 프로그램 내 루프의 블록 수와 루프의 반복회수 및 자료종속성을 변화시키면서 루프의 경우에 대하여 각 프로세서별로 성능을 측정하였다.




(a) Unit 2개




(b) Unit 3개




(c) Unit 4개

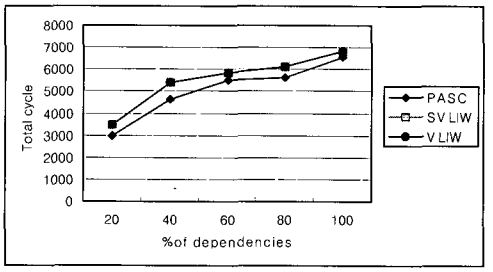
(그림 12) 유니트별 목적 코드 크기 비교

<표 1>은 연산처리기의 수가 세 개로 고정된 VLIW, SVLIW, PASC 프로세서 구조에서 루프의 반복회수를 100으로 고정시키고 자료종속성 비율을 0%에서 100%까지 변화시켰을 경우 각 프로세서 구조에서 요구되는 총 실행사이클을 측정한 것이다. 그리고 (그림 13)은 <표 1>을 도식화한 것이다. 이때 $\delta$는 타 프로세서에서 요구되는 총 실행사이클에서 PASC 프로세서에서 요구되는 총 실행사이클의 차로써, $\delta$가 양수이면 PASC 프로세서가 다른 프로세서에 비하여 $\delta$ 만큼 성능이 향상된 것이고 $\delta$가 음수이면 그렇치 않음을 의미한다.

〈표 1〉 자료종속성에 따른 총 실행사이클

| data dependency \ Processor | VLIW | SVLIW | PASC | $\delta$ |
|---|---|---|---|---|
| 20 | 3470 | 3470 | 2990 | 480 |
| 40 | 5390 | 5390 | 4630 | 760 |
| 60 | 5870 | 5870 | 5530 | 340 |
| 80 | 6110 | 6110 | 5630 | 480 |
| 100 | 6830 | 6830 | 6590 | 240 |

<표 1>의 결과를 통하여 루프내의 자료종속성의 변화에 관계없이 항상 PASC 프로세서가 기존의 VLIW 프로세서나 SVLIW 프로세서에 비하여 항상 우수함을 입증하였으며 $\delta$가 고정된 루프횟수인 100 이상으로 성능향상을 보인 것은 루프 수행시 매 반복마다 한 사이클씩 절약하여 얻은 100 사이클과 목적 코드 압축을 통하여 줄어든 실행사이클의 합 때문이다. (그림 12)으로부터 PASC용 목적 코드는 다른 프로세서용 목적 코드에 비하여 높은 코드압축율을 가짐을 알 수 있는데 이와같은 코드압축은 결국 캐시미스를 적게 발생시키므로 평균 메모리참조 시간을 줄여서 총 실행사이클을 단축시켜서 성능을 향상시킨다.

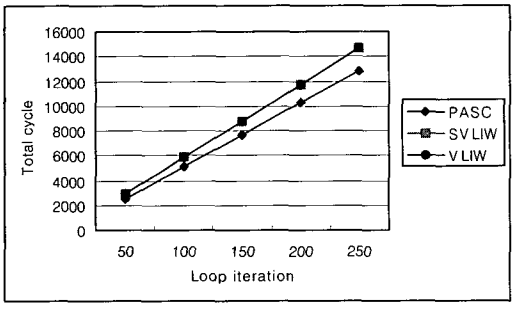



(그림 13) 자료종속성에 따른 총 실행사이클

<표 2>는 연산처리기의 수가 세 개인 VLIW, SVLIW, PASC 프로세서 구조에서 자료종속성을 60%로 고정시키고 루프의 반복회수를 50에서 250까지 변화시켰을 경우 각 프로세서 구조에서 요구되는 총 실행사이클을 측정한 것으로 (그림 14)는 이를 도식화한 것이다. 이때 $\beta$는 타 프로세서에서 요구되는 총 실행사이클에서 PASC 프로세서에서 요구되는 총 실행사이클의 차로써, $\beta$가 양수이면 PASC 프로세서가 다른 프로세서에 비하여 $\beta$만큼 성능이 향상된 것이고 $\beta$가 음수이면 그렇치 않음을 의미한다.

〈표 2〉 반복회수 변화에 따른 총실행사이클

| Loop iteration \ Processor | VLIW | SVLIW | PASC | $\beta$ |
|---|---|---|---|---|
| 50 | 2935 | 2935 | 2765 | 170 |
| 100 | 5870 | 5870 | 5530 | 340 |
| 150 | 8805 | 8805 | 8295 | 510 |
| 200 | 11740 | 11740 | 11060 | 680 |
| 250 | 14675 | 14675 | 13825 | 850 |




(그림 14) 반복회수 변화에 따른 총 실행사이클

<표 2>의 결과를 통하여 루프내의 반복회수의 변화에 관계없이 PASC 프로세서가 기존의 VLIW 프로세서나 SVLIW 프로세서에 비하여 항상 우수함을 입증하였으며 $\beta$가 변화하는 루프내의 반복회수 이상으로 성능향상을 보인 것은 <표 1>의 결과와 동일하게 루프 수행 시 반복회수 만큼 절약하여 얻을 수 있는 사이클 수와 코드압축으로 인해서 절약된 실행사이클 때문이다. 예를 들어 루프 반복회수 $\Delta L = 100$일 때 $\beta$= 340인 경우, PASC 프로세서는 타 프로세서에 비하여 루프수행을 통하여 줄어든 실행사이클 $\Delta L$와 코드압축으로 인해서 줄어든 실행사이클 240의 합만큼의 성능향상을 갖는다.

## 6. 결론 및 향후 계획

본 논문에서는 프로세서의 성능 향상을 위하여 각각의 연산처리기와 결합한 개별적인 스케쥴러를 쌍으로 여러 개 구성하여 만든 PASC 프로세서를 제안하고 대표적인 VLIW 프로세서와 비교·분석하였다. PASC 프로세서는 각각의 연산처리기에 자료종속성을 해결할 수 있도록 스케쥴러를 두어 컴파일러가 해야만 하는 처리를 하드웨어가 일부 처리하게 함으로써 결국 목적코드의 크기가 VLIW 프로세서용 목적 코드의 크기보다 훨씬 더 압축된 목적 코드를 구성할 수 있다. 루프의 경우에는 프로그램의 형태에 따라 루프가 매 반복할 때마다 실행사이클이 한 사이클씩 더 적게 수행되므로 전체적으로 성능이 향상되는 효과를 얻을 수 있다. 본 논문에서는 PASC 프로세서를 모의할 수 있는 시뮬레이션 시스템을 구현하여 루프의 처리에 따른 성능 향상의 정도를 측정하였다. 시뮬레이션에서는 명령어의 수, 자료종속성 정도 및 반복의 횟수 등을 변화시키면서 다양한 형태의 그래프를 생성하고 이를 입력자료로 사용하여 VLIW, SVLIW 및 PASC 구조하에서 프로그램의 총 실행사이클을 측정하였다. 그 결과, 반복횟수가 증가할수록 많은 성능향상을 얻을 수 있었다. 이것으로 보아 본 논문에서 제안한 구조가 루프수행에 특히 효과적인 것을 알 수 있었다.

또한, PASC용으로 생성된 목적 코드 내에서 NOP 명령어가 차지하는 크기가 VLIW나 SVLIW 프로세서용으로 생성된 목적 코드의 경우보다 작은 것으로 보아 비록 PASC용 목적 코드는 SVLIW나 VLIW용 목적 코드와는 달리 자료종속성 정보가 목적 코드에 포함됨에도 불구하고 전체적으로 목적 코드의 길이가 증가하지 않는다는 것도 확인되었다. 그리고 목적 코드의 길이가 늘어나지 않기 때문에 캐시의 효율이 SVLIW 구조의 효율과 비슷할 것으로 기대된다. 앞으로 이 분야에 관한 연구를 계속할 예정이며, PASC 프로세서를 구성하는 연산처리기 수의 변화에 따른 성능향상 여부에 관한 연구도 계속할 계획이다.

## 참 고 문 헌

[1] Kevin W. Rudd and Michael J. Flynn "Instruction-level parallel processors-dynamic and static scheduling tradeoffs," *Proc. Inter. Symp. Para. Algo. Arch. Synth.*, pp.74-80, March 1997.

[2] Roger Espasa and Mateo Valero, "Exploiting instruction-and data-level parallelism," *IEEE Micro*, Vol.17, No.5, Sept/Oct 1997.

[3] Rau B. R., "Dynamically scheduled VLIW processors," *Proc. 26th Inter. Symp. Micro.*, pp. - 80-90, 1993.

[4] G. Slaveburg, S. Rathnam, and H. Dijkstra, "The

TriMedia TM-1 PCI VLIW media processor," *Symp. High-Performance Chips,* August 1996.

[5] Thomas M. Conte and Sumedh W. Sathaye, "Dynamic Rescheduling : A technique for object code compatibility in VLIW architecture," *Proc. 28th Inter. Symp. Micro.,* 1995.

[6] Arthur Abnous and Nader Bagherzadeh, "Pipelining and bypassing in a VLIW processor," *Trans. Para. Dist. Sys.,* Vol.5, No.6, pp.658-664, June 1994.

[7] Joseph A. Fisher, "The VLIW machine : multiprocessor for compiling scientific code," *Proc. Inter. Conf. Para. Pro.,* pp.45-53, July 1984.

[8] Alexandru Nicolau, Joseph A. Fisher, "Measuring the parallelism available for VLIW architectures," *Trans. Comp.,* Vol.C-33, No.11, pp.968-976, November 1984.

[9] Robert P. Colwell, Robert P. Nix, and etc "A VLIW architecture for a trace scheduling compiler," *Proc. Trans. Comp.,* Vol.37, No.8, pp.967-979, August 1988.

[10] Joseph A. Fisher, "Trace scheduling : A technique for global microcode compaction," *Trans. Comp.,* Vol.C-30, No.7, pp.478-490, July 1981.

[11] Arthur Abnous, Roni Potasman and Alex Nicolau, "A percolation based VLIW architecture," *Proc. Inter. Corf. Para. Pro.,* pp.I144-I148, 1991.

[12] Eisenbeis, C., "Optimization of horizontal microde generation for loop structures," *Inter. Conf. Supercomputing,* pp.453-465, July 1988.

[13] R. Govindarajan, Erik R. Altman and Guang R. Gao, "A framework for resource-constrained rate-optimal software pipelining," *IEEE Trans. Dist. Sys.,* Vol.7, No.11, November 1996.

[14] Soo-Mook Moon, Kemal Ebcioglu, "On performance and efficiency of VLIW and superscalar," *Proc. Inter. Conf. Para. Pro.,* pp.II 283-287, 1993.

[15] Shyh-Kwei Chen, W. Kent Fuchs and Wen-Mei W. Hwn, "An analytical approach to scheduling code for superscalar and VLIW architectures," *Proc. Inter. Conf. Para. Pro.,* pp.I258-I292, 1994.

[16] Ashok Kumar "The HP PA-8000 RISC CPU" *IEEE Micro,* Vol.17, No.2, March/April 1997

[17] A. Leung, K. Palem, and C. Ungureanu "Run-time versus Compile-time Instruction Scheduling in Superscalar (RISC) Processors : Performance and Tradeoffs," Technical report 699, New York University, Computer Science, July 1995.

[18] Chriss Stephens, Bryce Cogswell, and etc, "Instruction level profiling and evaluation of the IBM RS/6000," *Proc. Inter. Symp. Comp. Arch,* pp.180-189, 1991.

[19] Dezso Sima, "Superscalar instruction issue," *IEEE Micro,* Vol.17, No.5, Sept/Oct 1997.

[20] Shusuke Okamoto and Masahiro Sowa, "Hybrid processor based on VLIW and PN-Superscalar," *PDPTA'96 Inter. Conf.,* pp.623-632, 1996.

[21] Boyoun Jeong, Joongnam Jeon and Sukil Kim, "Design of VLIW architectures minimizing dynamic resource collisions," *Journal KISS,* Vol.24, No.4, pp.357-368, April 1997.

[22] A.B. Ruighaver "From A very long instruction word architecture to a decoupled multicomputer architecture," *Proc. Inter. Conf. Para. Pro.,* pp. I188-191, 1992.

[23] Sung-Hyun Jee, No-Kwang Park and Sukil Kim, "Performance analysis of caching instructions on SVLIW processor and VLIW processor," *Journal IEEE Korea Council,* Vol.1, No.1, December 1997.

[24] Zhizhong Tang, Chihong Zhang, Sifei Lv and Tao Yu, "A new architecture for branch-intensive loops," *Proceedings Advances in Para. Dist. Comp.,* pp.241-246, March 1997.

[25] Mamoru Sakamoto and Toyohiko Yoshida, "Micro support for reducing branch penalty in a superscaler processor," *Proc. Comp. Design,* pp.208-226, October 1996.

[26] Arthur Abnous and Nader Bagherzadeh, "Pipelining and bypassing in a VLIW processor," *Trans. Para. Dist. Sys.,* Vol.5, No.6, pp.658-664,

June 1994.

[27] Sung-Hyun Jee, No-Kwang Park and Sukil Kim, "Performance evaluation of data dependence removable instruction pipelines," submitted to *Journal KISS*, July 1998.

**지 승 현**

e-mail : jsh@mail.chonan-c.ac.kr

1993년 충북대학교 전자계산학과 학사 취득

1995년 충북대학교 전자계산학과 석사 취득

1998년 충북대학교 전작계산학과 박사과정 수료

1999년~현재 천안외국어대학 사무자동화과 교수로 재직 중

관심분야 : 병렬처리 컴퓨터 구조, 병렬처리 네트웍, 멀티미디어 프로세서 등임

**박 노 광**

e-mail : pnk@csca.chungbuk.ac.kr

1997년 한국기술교육대학교 정보통신공학과 학사 취득

1999년 충북대학교 전자계산학과 석사 취득

관심분야 : 병렬처리 컴퓨터 구조, 병렬처리 네트웍, 병렬처리 알고리즘 등임

**전 중 남**

e-mail : joongnam@cbucc.chungbuk.ac.kr

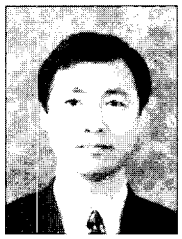
1981년 연세대학교 전자공학과 학사 취득

1985년 연세대학교 전자공학과 석사 취득

1990년 연세대학교 전자공학과 박사 취득

1990년~현재 충북대학교 컴퓨터과학과 부교수로 재직 중

관심분야 : 컴퓨터 구조, 병렬처리 알고리즘, 공장자동화 등임



**김 석 일**

e-mail : ksi@cbucc.chungbuk.ac.kr

1975년 서울대학교 전기공학과 학사 취득

1975년~1990년 국방과학연구소 선임연구원으로 근무

1985년~1989년 미국 North Carolina State University에서 공학박사 취득

1990년~현재 충북대학교 컴퓨터과학과 부교수로 재직 중

관심분야 : 병렬처리 컴퓨터 구조, 슈퍼컴퓨팅, 병렬처리 언어, 이기종 분산처리, 시각장애인 사용자 인터페이스 등임