論文95-32B-3-2

병렬 파이프라인 프로세서 아키텍처의 설계

(Design of a Parallel Pipelined Processor Architecture)

李 相 靜 *, 金 光 駿 **

(Sang-Jeong Lee, and Kwang-Jun Kim)

요 약

본 논문에서는 작은 VLIW 아키텍처와 같이 동작하는 병렬 파이프라인 프로세서 모델과 이 프로세서 상에서 병렬 처리되는 명령어를 추출하고 스케쥴링하는 스케쥴링 알고리즘을 제안한다. 제안된 프로세서 모델은 이중 구조의 명령어 형식을 갖고 최대 4개의 기본 오퍼레이션이 병렬로 수행하면서 파이프라인 처리되며, 기본 오퍼레이션의 조합으로 응용목적에 따라 다양한 기능을 하는 명령어의 설계가 가능하다. 스케쥴링 알고리즘은 수행되는 프로그램의 데이타 종속관계 및 자원 상충관계를 조사하여 병렬처리되는 오퍼레이션들을 스케쥴하고 파이프라인 해저드를 제거한다. 제안된 프로세서 모델의 동작 검증을 위하여 C 컴파일러와 시뮬레이터를 개발하여 다양한 예의 프로그램을 적용하고 시뮬레이션하여 설계된 프로세의 동작 특성 및 성능을 측정한다.

Abstract

In this paper, a parallel pipelined processor model which acts as a small VLIW processor architecture and a scheduling algorithm for extracting instruction-level parallelism on this architecture are proposed. The proposed model has a dual-instruction mode which has maximum 4 basic operations being executed in parallel. By combining these basic operations, variable instruction set can be designed for various applications. The scheduling algorithm schedules basic operations for parallel execution and removes pipeline hazards by examining data dependency and resource conflict relations. In order to examine operation and evaluate the performance,a C compiler and a simulator are developed. By simulating various test programs with the compiler and the simulator, the characteristics and the performance result of the proposed architecture are measured.

* 正會員, 順天鄕大學校 電算學科

(Dept. of Computer Science, Soonchunhyang University)

** 正會員, 大宇 半導體硏究所 製品開發 1室

(Products Development Div. 1, Semiconductor R&D Center, Daewoo Co.)

※ 이 논문은 1993년도 한국학술진흥재단의 공모과제 연구비에 의하여 연구되었음

接受日字 : 1994年 9月 5日

I. 서 론

오늘날 반도체 제조기술의 발달과 발전된 컴파일러 최적화 기법의 등장으로 명령어 수준에서 병렬성(instruction-level parallelism)을 추출하여 한 클럭 사이클에 다수의 명령어를 실행함으로써 처리속도를 고속화하는 고성능 프로세서의 개발이 활발이 이루어지고 있다. 초기의 표준 RISC(Reduced Instruction Set Computer) 프로세서가 간소화된 명령어를 갖고 아키텍처를 단순화하고 명령어의 파이프라인 처

리를 효율적으로 수행하여 이전의 CISC(Complex Instruction Set Computer)와 비교하여 어느 정도는 프로세서의 성능을 향상 시켰다. 그러나 한 클럭 사이클에 한개의 명령어 만을 실행하기 때문에 명령어를 파이프라인 처리하여도 프로세서가 도달할 수 있는 최대 성능 향상은 명령어 당 소요되는 클럭인 CPI (Clock Per Instruction)를 1 이하로 줄일 수 없고, 실제 프로그램 환경에서는 데이타 종속관계, 분기명령 및 프로세서의 구조적인 문제로 발생하는 파이프라인 해저드(pipeline hazard)로 인하여 지속적인 파이프라인의 처리가 어려워서 CPI가 1 이상이 된다. 이러한 단순 RISC 아키텍처의 한계를 극복하고자 프로세서에 여러개의 기능장치(functional units) 등의 하드웨어를 구비하고,이들 복수개의 기능장치들을 효율적으로 이용할 수 있도록 컴파일러가 프로그램 내에서 존재하는 명령어들 간의 병렬성을 최대한으로 추출하고 스케쥴함으로써 한 사이클 당 하나 이상의 명령어를 중첩 실행하여 성능을 향상시키고자 하는 연구가 대두되었다. 이러한 시도의 대표적인 아키텍처로는 VLIW (Very Long Instruction Word) 아키텍처와 슈퍼스칼라(superscalar) 아키텍처가 있다.

VLIW 아키텍처[2-3][7-8][13][16]는 종래 CISC의 수평 마이크로코드 프로세서의 개념을 확장 발전시킨 기법으로 다수의 중복된 기능 장치들의 동작이 하나의 긴 명령어의 고정된 필드에 의해 기술되어 병렬처리되는 아키텍처이다. 즉,VLIW 아키텍처는 다수의 기능장치들이 하나의 광역 제어장치(global control unit)로 제어되는 단일 명령어 흐름을 갖고 사이클 당 여러 오퍼레이션들을 동시에 실행하고, 최적화 컴파일러가 전적으로 병렬처리되는 명령어의 스케쥴링 및 동기화를 책임진다. 따라서 명령어의 병렬수행을 위한 하드웨어 로직과 명령어 간의 인터록(interlock) 하드웨어를 요구하지 않아 아키텍처의 구성이 규칙적이고 간단하다. 그러나 VLIW 아키텍처의 성능 향상을 위해서는 병렬처리 되는 명령어 간의 데이타 종속관계, 자원 상충관계 등을 실행하기 전에 컴파일러가 스케쥴해야 하므로 보다 정교한 최적화 컴파일러가 필수적인 요소가 된다. 이러한 최적화 컴파일러에 적용되는 대표적인 스케쥴 기법으로는 VLIW 프로세서 상에서 처음으로 기본블럭을 넘어 코드를 스케쥴링하는 Trace Scheduling[11][11]이 있다. 또한 Percolation Scheduling[6]은 다양한 스칼라 코드로 부터 사이클 당 여러 오퍼레이션을 스케쥴링 하는데 효과적인 기법이고, 소프트웨어 파이프라이닝(software pipelining)[9][14]은 반복적인 오퍼레이션을 강력하게 스케쥴하기 위해 프로그램 루프에 적용되는 기법이다.

슈퍼스칼라 프로세서[10][15][19-21]는 다중 기능장치를 갖고 한 클럭 사이클에 여러 명령들이 병렬처리 된다는 점에서는 VLIW 머신과 많은 공통된 특성을 갖는다. 그러나 슈퍼스칼라 프로세서는 컴파일 시가 아닌 실행 시에 하드웨어에 의해 직렬의 명령어 열로 부터 병렬로 수행될 수 있는 명령어들을 이슈하여 동적으로 스케쥴하기 때문에 VLIW 아키텍처와 비교하여 컴파일러의 부담이 줄어들고 현존하는 아키텍처와의 목적코드 호환성이 가능하다는 장점이 있는 반면에 명령어 디코딩,이슈(issue),분기예측 및 인터록을 위한 복잡한 하드웨어가 필수적이라는 단점이 있다.

본 논문에서는 VLIW 아키텍처의 개념을 도입하여 단일칩이 작은 VLIW 프로세서와 같은 동작을 하는 병렬 파이프라인 구조의 프로세서 모델을 설계 제안한다. 제안된 프로세서 모델은 이중 구조의 명령어 형식을 갖고 최대 4개의 기본 오퍼레이션이 병렬로 수행하면서 파이프라인 처리되며, 기본 오퍼레이션의 조합으로 응용목적에 따라 다양한 기능을 하는 명령어의 설계가 가능하다. 제안된 모델은 대규모 응용 프로그램의 병렬처리를 위하여 여러 개의 프로세서를 쉽게 확장 연결이 가능하도록 설계함으로써 멀티프로세서 구성 시 많은 명령어들을 병렬처리하여 전형적인 VLIW 아키텍처로 동작한다. 또한 다중 오퍼레이션을 병렬처리하고 파이프라인 해저드의 효율적인 처리를 스케쥴하는 병렬 스케쥴링 알고리즘을 제안한다. 제안된 프로세서 모델의 동작 검증을 위하여 C 컴파일러와 시뮬레이터를 개발하였다. 개발된 C 컴파일러에 다양한 예의 프로그램을 적용하여 제안된 프로세서의 목적 코드를 생성하고, 이를 입력으로 받아 시뮬레이션하여 설계된 프로세의 동작 특성 및 성능을 측정한다.

Ⅱ. 병렬 파이프라인 프로세서 모델

1. 프로세서 아키텍처

설계 제안된 병렬 파이프라인 프로세서 아키텍처는 한 사이클 당 두개 이상의 오퍼레이션을 동시에 수행하고 각 오퍼레이션 간의 4단 파이프라인을 지원하기 위해 그림 1과 같이 2개의 정수처리부(IU,integer unit),하나의 실수처리부(FPU,floating-point unit) 기능장치와 레지스터 화일 및 제어장치로 구성된다.

정수처리부는 덧셈,뺄셈 및 논리연산을 수행하는 OP1 오퍼레이션과 시프트,로테이션 기능을 수행하는 OP2 오퍼레이션 두 부분의 조합으로 동작하도록 그림 2와 같이 구성되었다. 이러한 구조는 기존의 RISC 프로세서들이 두 개의 명령으로 처리해야 했던 명령의

일부를 하나로 압축 시킴으로써 두 개의 정수 처리부를 가지면서도 최대 4개의 오퍼레이션들을 병렬처리할 수 있도록 구성하였다.

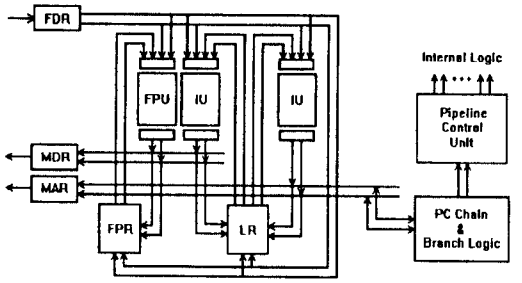

그림 1. 병렬 파이프라인 프로세서 모델
Fig. 1. The model of the parallel pipelined processr.

그림 2와 같은 구조로 정수처리부를 구성하면 시프트 연산과 덧셈,뺄셈 등의 산술 연산을 한 명령어로 실행하여 오퍼레이션의 병렬처리 기회를 높일 수 있다. 이를 이용하는 대표적인 연산으로는 배열의 주소를 계산하는 예를 들수 있다. 정수형 자료가 4 바이트이고 배열의 시작주소를 a라 할 때 배열 a[i]의 주소는 a + i * 4 로 계산된다. 이때 i*4의 연산은 2 비트 만큼의 왼쪽시프트(left shift)로 대체할 수 있으므로 다음과 같이 레지스터 R4에 배열의 주소가 놓이도록 계산될 수 있다.

R2 ← a, R3 ← i : 배열의 시작주소, 인덱스
R4 ← R2 + (R3 ⟨⟨ #2) : a[i]의 주소

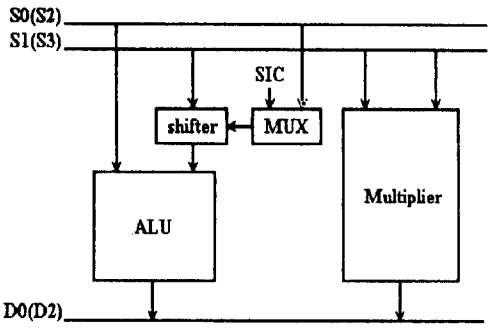

그림 2. 정수 처리부 기능장치
Fig. 2. Integer functional unit.

제안된 아키텍처에서는 아래와 같은 형식의 slad(shift left and addition) 명령이 이를 지원한다.

slad R4, R2, R3, #2 : R4 ⟨- R2 + (R3 ⟨⟨ #2)

레지스터는 각각 32개의 32비트 정수 레지스터(LR,local register)와 실수형 레지스터(FPR,floating-point register)가 있다. 또한 프로세서 외부의 메모리 또는 다른 프로세서와의 통신을 위해서 FDR (front-door register), MDR(memory data register),MAR(memory address register)가 있다. FDR 레지스터는 외부로 부터 데이타의 수신 시에 사용되는 버퍼이고, MDR은 프로세서에서 외부로 데이타의 송신 시에 사용되는 레지스터이다.

2. 명령어 아키텍처

제안된 아키텍처의 명령어 형식은 그림 3과 같이 각각 32비트인 IA,IB의 오퍼레이션 형식으로 구성된 이중 명령어 형식을 갖는다. IA 오퍼레이션은 정수 연산 및 실수 연산을 지정하고, IB 오퍼레이션은 정수 연산 및 분기 오퍼레이션을 지정한다.

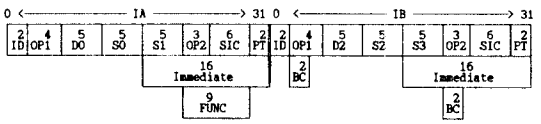

그림 3. 명령어 형식
Fig. 3. Instruction format.

IA,IB 각 형식은 ID(IDentification) 필드의 기술에 따라 표 1과 같이 각각 4개의 동작 모드를 가지며 각 명령어 형식은 그림 4와 같이 표현된다.

표 1. ID 값에 따른 동작 모드
Table 1. Operation modes identified by ID field.

ID	IA Operation	IB Operation
0	IRRA, Integer Operations*	IRRB, Integer Operations*
1	IRIA, OP1 Immediate Operations	IRIB, OP1 Immediate Operations
2	FRRA, Floating Point Operations	BRRB, RR type Branch Operations
3	Reserved	BRIB, RI type Branch Operations

IRRA,IRRB 형식과 IRIA,IRIB 형식은 각각 IA, IB 오퍼레이션의 정수형 레지스터-레지스터 오퍼레이션(Integer Register-Register)과 정수형 레지스터-이미디어트(Integer Register-Immediate) 형식이다. 또한 BRRB,BRIB 형식은 모두 IB 오퍼레이션으로 OP1의 레지스터-레지스터,레지스터-이미디어트 연산의 결과로 계산된 주소로 BC 필드에 기술된 조건에 따라 분기하는 오퍼레이션이다. FRRA 형식은 실수 연산을 위한 레지스터-레지스터 형식이다. 이들 형식에서 OP1, OP2 필드는 각각 병렬처리가 가능한 정수형 산술 및 논리 연산과 시프트 연산을 나타내는 오퍼레이션 코드(OP code) 필드이며 수행되는 연산은 표 2와 같다. FRRA 형식에서 수행되는 실수 연산의 동작은 OP 필드와 FUNC 필드의 결합으로 표현된다. D0,D2는 IA,IB 형 오퍼레이션을 위한 데스티네이션 레지스터(destination register)를 나타내고, S0,S1

과 S2,S3는 각각 IA,IB 형 오퍼레이션의 소스 레지스터(source register) 필드이다.

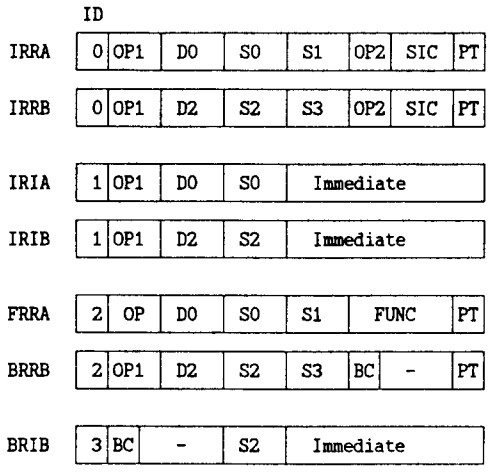

그림 4. 오퍼레이션 모드의 명령어 형식
Fig. 4. Instruction formats of operation modes.

표 2. OP1, OP2 연산
Table 2. Operations of OP1,OP2.

연산	의미(mnemonic)
OP1 연산	NOP, ADD, ADDC, ADDU, ADDUC, SUB, SUBB, SUBUB, MUL, MULU, AND, OR, XOR, MASK
OP2 연산	NOP, SLL, SRL, SLA, SRA

표 3. SIC 필드 구분
Table 3. Identification of a SIC field.

SIC	Meaning
0xxxxx	Shift Amount Immediate Value
110000	S, Sign Condition
101000	Z, Zero Condition
100100	C, Carry Condition
100010	V, Overflow Condition
100001	Reserved

SIC(Shift amount Immediate and Condition code) 필드는 시프트 오퍼레이션일 때는 시프트 되는 상수 값을 표시하며, 비교 오퍼레이션일 때는 비교 조건을 표시하는 필드이다. SIC 필드의 구분은 표 3과 같이 필드의 최상위 비트가 0 일 때는 OP2 연산에 의해 시프트 되는 상수 값을 나타내며, 1 인 경우에는 비교 조건을 표시한다. 비교 조건인 경우에는 S,Z,C,V의 각 조건 비트를 세트되며 각 조건 비트는 SIC의 최상위 비트가 1 인 경우에 OR 결합으로 조합되며 0 인 경우이거나 IRIA,IRIB 형식인 정수 오퍼레이션의 경우에는 자동적으로 Z 조건 비트를 선택하여 등가 비교만이 가능하도록 설계하였다. 각 조건은 분기 오퍼레이션 시에 표 4와 같이 정의된 분기 조건인 BC (Branch Condition) 필드에 의해 기술되는 분기 조건 참,거짓과 조합함으로써 대소 비교의 조건을 선택할 수 있다. BC 필드에서 멀티프로세서 동작 지원을 위해 조건을 프로세서 내부에서만 분기하는 국소조건(local condition)과 전체 프로세서가 동기되어 분기하는 광역조건(global condition) 으로 구분하였다.

표 4. BC 필드의 구분
Table 4. Identification of a BC field.

BC	Condition
0	Global True Branch
1	Global False Branch
2	Local True Branch
3	Local False Branch

PT(PorT) 필드는 ID 필드와 결합하여 레지스터 필드 D0,D2,S0,S1,S2,S3에 대응되는 범용 레지스터와 특수 레지스터의 매핑 관계를 표 5와 같이 나타내는 필드이다.

표 5. PT 필드 매핑
Table 5. Mapping of a PT field.

		IA			IB		
ID	PT	DO	SO	S1	D2	S2	S3
0	0	LR	LR	LR	LR	LR	LR
	1	LR	LR	FDR	LR	LR	FDR
	2	ET	LR	LR	ER	LR	LR
	3	LR	FDR	FDR	LR	FER	FDR
1	-	LR	LR	-	LR	LR	-
2	0	FPR	FPR	FPR	LSR	LSR	LSR
	1	FPR	FPR	FDR	LSR	LSR	LSR
	2	ET	FPR	FPR	ER	LSR	LSR
	3	FPR	FDR	FDR	LSR	FDR	FDR
3	-	-	-	-	-	LSR	-

표 5에서 LR(Local Register)은 32개의 32 비트 범용 레지스터 R0-R31을 나타내고 LSR(Local & Special Register)은 R0-R15 까지의 15개의 32비트 범용 레지스터와 프로그램 카운터 PC, 플래그 레지스터, BTAR(Branch Target Address Register)등과 같은 특수 레지스터를 표시한다. FPR은 32개의 32비트 실수 레지스터를 나타낸다. FDR은 전방참조 레지스터(Front Door Register)로써 64비트 레지스

터이다. FDR은 외부 메모리의 데이타를 읽을 때 버퍼로 사용된다. 또한 멀티프로세서 시스템 구성 시 다른 프로세서로 전송될 데이타의 버퍼로 사용되는 레지스터이다. ET(External Transmission)는 외부 프로세서나 메모리에 데이타 전송 시에 수신할 프로세서의 번호(또는 메모리 뱅크 번호)를 기술한다. 단일 프로세서 시스템에서는 메모리에 저장될 데이타를 MDR에 전송하기 위해서 이 필드에 자기 프로세서의 번호를 기술하면 된다. ER(External Reception)이 선택되면 MAR에 연산 결과가 저장되고, ER의 내용은 메모리와 프로세서 간의 데이타 전송 방식을 표 6과 같이 표시한다. 표 6에서 bit 0는 워드 값이 바이트 단위로 메모리에 저장되는 방식을 나타내고, 비트 1은 메모리 읽기,쓰기를 나타내며 비트 2는 메모리,프로세서 간의 전송 여부를 표시하는 비트이다. 비트 3,4는 전송될 데이타의 크기를 표시한다. 각 비트는 OR로 결합되어 의미를 갖는다. 예를들어 BWM2 는 Big endian으로 4 바이트 데이타를 메모리에 저장(write)하는 것을 의미한다.

표 6. ER 비트 값의 정의
Table 6. Definition of bits of a ER field.

비트	기호	의미
bit 0	L!B	1 : Little endian, 0 : Big endian
bit 1	R!W	1 : Read, 0 : Write
bit 2	P!M	1 : Processor, 0 : Memory transfer data size,
bit 3	TDSZ	00 : 1 byte, 01 : 2 byte, 10 : 4 byte(W),
bit 4	TDSZ	11 : 8 byte

명령어 형식에서 제공되는 대부분의 연산이 OP1, OP2에서 기술되는 연산의 동작에 의해 지시된다. 따라서 이 두 연산과 각 필드의 조합에 의하여 다양하고 새로운 명령이 정의될 수 있다. 이는 기존의 마이크로 프로그램 제어 시스템에서 마이크로오퍼레이션의 조합으로 기계어 명령이 정의되는 것과 유사하고, 프로세서가 적용되는 응용 목적에 따라 적합한 명령을 정의할 수 있는 장점이 있다. 명령어 정의 일부 예를 표 7에 나타내었다.

표 7에서 mov 명령은 OP1의 기본 오퍼레이션 ADD를 이용하여 정의하였고 IRR 형식으로 지정되어 IA,IB 오퍼레이션 양쪽에 적용될 수 있다. lw(load word) 명령은 s1+s2의 주소로 지정된 메모리를 읽는 명령으로 IRRB 형식으로 레지스터 매핑 필드 PT의 값이 2이므로 IB 오퍼레이션에서 D2 필드가 ER이 되고 그 값은 B(Big Endian),R(Read), M(Memory), 2(4 바이트)가 된다.(표 6 참조) sb(store byte) 명령은 IA 오퍼레이션으로 저장될 데이타를 fdr에 저장하고, IB 오퍼레이션으로 메모리의 주소를 계산한다.

표 7. 명령어 정의 예
Table 7. Examples of instruction definitions.

명령	의미, 명령어 정의
mov d,s	d <- s+0 IRR ADD dDL[LR d] LR s LR 0 NOP SIC[0,0] PT[0]
add d, s1, s2	d <- s1+s2 IRR ADD dDL[LR d] LR s1 LR s2 NOP SIC[0,0] PT[0]
add d, s1, im	d <- s1+IM IRI ADD dDL[LR d] LR s1 LR s2 NOP SIC[0,0] PT[0]
s11 d, s1, s2	d <- s1<<s2 IRR NOT dDL[LR d] LR s2 LR s1 SLL SIC[1,0] PT[0]
lw fdr, s1, s2	fdr <- M[s1+s2] IRRB ADD MAR|BRM2 LR s1 LR s2 NOP SIC[0,0] PT [2]
sb s1, s2, s3	M[s1+s2] <- s3 IRRA ADD dDL[ET 0] LR s3 LR 0 NOP SIC[0, 0] PT[2], IRRB ADD MAR|BWMO LR s1 LR s2 NOP SIC[0,0] PT[2]
jmp lb	PC <- cp+lb IRRA SUB dDL[LR 0] LR 0 LR 0 NOP SIC[0,0] PT[0], BRIB Gtr - LR cp IMME : lb
bltt s1, s2, lb	IRRA SUB dDL[LR 0] LR s1 LR s2 NOP SIC[1,16] PT[0], BRIB Gtr - LR cp IMME : ib
slad d, s1, s2, im	d <- s1+(s2<<IM) IRR ADD dDL[LR d] LR s1 LR s2 SLL SIC[1,im] PT[0]
push s1	M[sp] <- s1 ; sP <- sp-4 IRRA ADD dDL[ET 0] LR s1 LR 0 NOP SIC[0, 0] PT[2], IRRB ADD MAR|BWM2 LR sp LR 0 NOP SIC[0, 0] PT[2] ; IRI SUB dDL[LR sp] LR sp IMME : 4

무조건 분기 명령 jmp는 IA 오퍼레이션에서 조건을 계산한다. 이 경우 0에서 0을 빼주므로 항상 참이 되어 무조건 분기가 된다. IB 오퍼레이션에서 BC 필드가 Gtr(global true branch)의 분기조건을 지정하고 계산된 주소로 분기한다. bltt(branch less than true) 명령은 IA 오퍼레이션에서 조건을 테스트 한다. 즉, SIC 값에 최상위 비트가 1 이므로 조건을 표시하고 조건값이 16(sign condition)이므로 s1이 s2 보다 작으면 참이 된다. IB 오퍼레이션에서 이 조건을 판정하여 분기한다. slad(shift left & addition) 명령은 OP1과 OP2 연산을 병렬처리하여 정의된 연산이다.

push 명령은 두개의 명령어로 구성된다. 첫번째 명령으로 s1을 sp가 지정하는 메모리에 저장하고, 두번째 명령으로 sp를 감소시키도록 구성한다.

현재 이와같은 형식으로 모두 105개의 명령을 정의하였고 C 컴파일러가 이들 명령들에 대하여 정의된 기본 오퍼레이션들을 생성하였다. 이 오퍼레이션들을 입력받아 스케쥴러가 병렬처리 될 수 있는 오퍼레이션들을 스케쥴하고, 이를 어셈블하여 목적코드를 생성한 후에 이를 입력받아 시뮬레이션하였다.

Ⅲ. 병렬 파이프라인 처리

1. 파이프라인 구조

제안된 프로세서는 그림 5와 같이 두개의 IA,IB 오퍼레이션이 병렬로 동시에 이슈되어 4단 파이프라인으로 처리된다. 각 단은 명령어 페치(IF) 스테이지, 명령어 디코드(ID) 스테이지, 실행(EXE) 스테이지와 저장(WB) 스테이지로 구분된다.

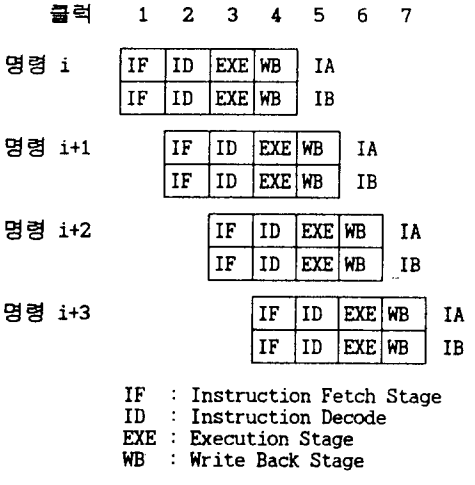

그림 5. 4단 파이프라인 처리
Fig. 5. 4-Stage pipeline processing.

IF 스테이지에서는 명령어를 IR(Instructuon Register) 레지스터에 페치한다. 명령어 페치 시 상위 워드는 IA 오퍼레이션, 하위 워드는 IB 오퍼레이션으로 간주되어 각각 IRA,IRB 레지스터에 페치된다. ID 스테이지에서는 명령어를 해독하고 레지스터에 저장된 소스 오퍼랜드를 소스 래치(source latch)로 읽어들인다. EXE 스테이지에서는 ALU에서 소스 래치를 읽어들여 연산을 수행한 후 그 결과를 데스티네이션 레지스터로 전송하거나 또는 forwarding인 경우에는 직접 ALU의 소스 래치로도 공급한다. 조건분기 오퍼레이션인 경우에는 IA 오퍼레이션은 조건을 결정하고, IB 오퍼레이션은 분기할 타겟 주소를 계산하여 BTAR (Branch Target Address Register) 레지스터에 저장한다. 메모리 참조 오퍼레이션인 경우에는 유효주소(effective address)를 계산한다. WB 스테이지에서는 ALU 연산의 결과를 레지스터에 저장한다. 조건분기 오퍼레이션인 경우에는 조건을 만족하면 BTAR의 타겟 주소를 읽어 PC 저장한다. load 오퍼레이션인 경우에는 메모리의 데이타를 읽어들여 fdr에 저장된다. store 오퍼레이션인 경우에는 메모리에 저장될 데이타가 메모리로 전송된다.

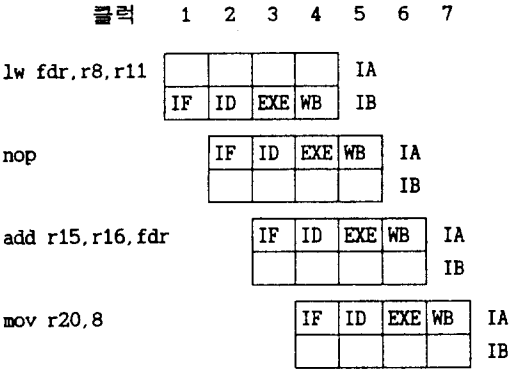

그림 6. 데이타 해저드 예
Fig. 6. An example of data hazard.

파이프라인의 처리 성능은 연속적인 파이프라인 처리를 방해하는 파이프라인 해저드(pipeline hazard)에 저하된다. 파이프라인 해저드는 데이타 해저드(data hazard),제어 해저드(control hazard) 및 구조 해저드(structural hazard)로 나누어진다. 데이타 해저드는 오퍼랜드의 정상적인 액세스 순서가 파이프라인 처리에 의해 변경될 때 발생하는 것으로 RAW (Read After Write),WAR(Write After Read), WAW(Write After Write) 해저드가 있다.[12] 제안된 아키텍처에서는 메모리 load 명령의 결과를 참조하고자 하는 경우에 RAW 해저드만 발생한다. 즉, 그림 6과 같이 메모리에서 읽어들인 fdr 레지스터의 결과를 참조하는 경우에 발생한다. 그림 6의 예에서 lw 명령으로 메모리의 데이타를 읽어들이면 클럭 4의 WB 스테이지에서 fdr에 저장된다. 따라서 다음 명령의 ID 스테이지의 클럭 3을 수행할 때에는 아직 fdr 값이 가용하지 않게된다. 또한 클럭 4의 EXE 스테이지에서도 아직 fdr에 값이 저장되지 않았으므로 forwarding에 의해 ALU로 직접 공급할 수 없다. 따라서 nop(no operation)를 삽입하여 한 사이클 지연시킨다. 한 사이클 지연시키면 클럭 4의 WB 스테이지의 결과가 forwarding으로 직접 ALU로 공급되어 add 명령의

EXE 스테이지에서 사용할 수 있게 된다.

이와같이 데이타 해저드가 발생하면 제안된 아키텍처는 파이프라인 인터록(pipeline interlock) 하드웨어를 사용하지 않고 컴파일러가 nop를 삽입하여 한 클럭 지연시키는 코드를 생성한 후 파이프라인 스케줄시 nop 위치의 슬롯(slot)에 이들 명령과 데이타 종속관계가 없는 mov 명령을 스케줄하여 수행시간을 줄인다.

```
lw  fdr,r8,r11      ; fdr <- M[r8+r11]
mov r20,8
add r15,r16,fdr     ; r15 <- r16 + fdr
```

제안된 알고리즘을 이용하여 스케줄한 결과 데이타 해저드로 인해 생성된 nop 중 약 90%가 파이프라인 스케줄에 의해 제거할 수 있었다.(IV 장 그림 13 참조)

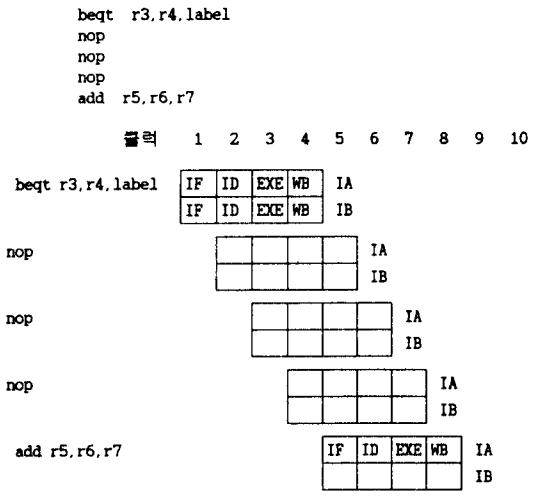

그림 7. 지연분기 예
Fig. 7. A Example of a delayed branch.

제어 해저드는 분기 명령에서 분기 조건의 결과 판정 지연과 분기 타겟 주소의 계산 지연 등으로 다음에 수행할 명령이 결정되지 않았을 때 발생하는 해저드이다. 제안된 프로세서의 조건분기 명령은 EXE 스테이지에서 조건을 비교하고 타겟 주소를 계산한 후 WB 스테이지에서 PC에 조건을 판정하여 분기이면 PC에 타겟 주소를 저장한다. 따라서 다음에 수행될 명령은 WB 스테이지 이전의 클럭에서는 페치할 수 없고 3 클럭 사이클 지연되어야 한다. 이와같은 제어 해저드가 발생하는 경우 대부분의 RISC 프로세서는 그림 7과 같이 지연분기(delayed branch) 방식을 채택하여 지연슬롯(delay slot)에 nop를 삽입하여 코드를 생성한 후 파이프라인 스케줄에 의해 분기 명령과 데이타 종속관계가 없는 명령들을 nop의 슬롯에 채워넣어서 제어 해저드로 인한 성능의 저하를 보상한다.

그러나 제어 해저드의 파이프라인 스케줄 결과 3개의 지연슬롯에 채워지는 경우는 약 37%로 데이타 해저드 보다는 다소 떨어졌다.(IV 장 그림 13 참조)

2. 병렬 스케줄링

제안된 아키텍처는 IA,IB의 2개 기본 오퍼레이션의 병렬처리가 가능하기 때문에 컴파일러가 각 오퍼레이션들의 코드를 생성한 후 병렬처리될 수 있는 오퍼레이션들을 스케줄해야 한다. 전형적인 VLIW 아키텍처에서는 수십개의 오퍼레이션들 병렬처리해야 하므로 Trace Scheduling이나 Percolation Scheduling과 같은 기법을 사용한 광역 스케줄링(global scheduling)을 한다. 광역 스케줄링은 스케줄되는 대상이 기본블럭(basic block)을 넘어서 전체 코드를 대상으로 하기 대문에 많은 오퍼레이션들을 병렬처리할 수 있다는 장점이 있다. 기본블럭은 코드의 처음과 끝을 제외하고는 분기의 타겟이 되는 오퍼레이션이나 분기 오퍼레이션을 포함하지 않는 코드의 시퀀스이다. 그러나 기본블럭을 넘어서 스케줄하기 위해서는 조건분기 오퍼레이션을 포함하여 스케줄해야 한다. 분기 조건은 실행시에 판정되어 프로그램의 수행 흐름이 결정되기 때문에 컴파일러가 스케줄할 때 분기의 결과를 예측하고 스케줄해야 하고 예측이 틀렸을 경우에는 이를 보상해야 하는 어려움이 있다. 제안된 프로세서에서는 병렬처리되는 오퍼레이션이 2개이므로 스케줄 대상을 기본블럭 내의 오퍼레이션으로 국한한 국소 스케줄링(local scheduling)을 적용하여도 성능이 크게 저하되지 않는다.

국소 스케줄링 알고리즘은 이전에 마이크로코드 컴팩션(microcode compaction) 시에 제안하여 사용하였던 알고리즘[5]을 수정하여 적용한다. 즉, 마이크로코드 컴팩션 알고리즘은 오직 병렬처리되는 마이크로 오퍼레이션 만을 스케줄하였지만 제안된 아키텍처의 알고리즘은 명령어의 병렬처리 뿐만 아니라 파이프라인 처리 스케줄하여야 하므로 파이프라인 해저드도 고려하여 스케줄하도록 수정하였다.

제안된 알고리즘은 스케줄될 오퍼레이션을 한번에 하나씩 선택하여 backtracking 없이 스케줄하는 heuristic algorithm이다. 알고리즘은 먼저 기본블럭 내의 각 오퍼레이션의 데이타 종속관계(data dependency relation)를 조사하여 각 오퍼레이션이 노드가 되고 각 에지가 데이타 종속관계를 나타내는 비순환형(acyclic) 그래프인 DDG(Data Dependency Graph)를 구성한다. 데이타 종속관계는 각 오퍼레이션의

```
schedule()
{
  S = 기본블럭의 오퍼레이션 집합;
  last_cycle = 0;
   while (집합 S가 공집합이 아니면) {
   op = 집합 S에서 가장 높은 우선순위의 오퍼레이션;
   S = S - { op };
   if (op가 데이타 해저드 발생 오퍼레이션이면) {
      op1 =  데이타 해저드 결합 오퍼레이션;
      S = S - { op1 };
   }
   cycle = last_cycle;
   while (cycle이 0이 아니면서 op가 INST[cycle]의 오퍼레이션들과
         데이타 종속관계가 없으면) {
         if (op가 데이타 해저드 발생 오퍼레이션이면서
              op1이 INST[cycle+2]와 데이타 종속관계가 있으면)
               break;
         --cycle;    /* 스케줄될 가장 빠른 클럭 사이클 결정 */
   }
   ++cycle;
   while (cycle != last_cycle+1) {
                        /* 마지막 스케줄 명령어 도달 때 까지 */
    if (op가 INST[cycle]의 오퍼레이션과 자원 상충관계가 없으면) {
       if (op가 데이타 해저드 발생 오퍼레이션이 아니면) {
          INST[cycle] = INST[cycle] + { op };  /* 스케줄 */
          break;
       }
       else if (op1이 INST[cycle+2]의 오퍼레이션과 자원 상충관계가
                없으면) { /* 데이타 해저드 발생 오퍼레이션 */
                   INST[cycle] = INST[cycle] + { op };
                   INST[cycle+2] = INST[cycle+2] + { op1 };
                   break;
          }
          else /* 자원 상충관계가 있으면 다음 사이클 조사 */
               ++cycle;
    }
}
if (cycle == last_cycle + 1) {
       /* op가 이미 스케줄된 명령어와 병렬처리할 수 없으면 */
       if (op가 데이타 해저드 발생 오퍼레이션이 아니면) {
              INST[cycle] = { op };   /* 새로운 명령어 생성 */
              last_cycle = cycle;
        }
        else { /* 데이타 해저드 발생 오퍼레이션 */
              INST[cycle] = INST[cycle] + { op };
              INST[cycle+2] = INST[cycle+2] + { op1 };
              last_cycle = cycle + 2;
        }
     }
  }
}
```

그림 8. 스케줄링 알고리즘
Fig. 8. Scheduling algorithm.

소스와 데스티네이션 레지스터를 조사하여 같은 레지스터인 경우에 발생한다. DDG 구성 시 데이타 해저드 발생 오퍼레이션(load 오퍼레이션) 노드와 다음에 지연될 오퍼레이션(데이타 해저드 결합 오퍼레이션) 노드 간에는 결합에지(coupled edge)로 표시한다. 또한 fdr 레지스터의 상충은 데이타 종속관계로 표시하지 않고 자원 상충관계로 간주하고, nop는 노드로 표시하지 않는다. DDG를 구성한 후에 각 노드에 대하여 자손 노드(descendant nodes)의 수를 세어서 자손 노드의 수가 많은 순으로 각 오퍼레이션에 대하여 스케줄링 우선순위(scheduling priority)를 부여한다. 자손 노드의 수가 같으면 소스 코드 상에서 먼저 기술된 순서대로 우선순위를 부여한다. 우선순위 부여 후에는 우선순위가 높은 오퍼레이션 부터 스케줄될 오퍼레이션으로 선택하고 각 명령어의 오퍼레이션과 데이타 종속관계 및 자원 상충(resource conflict) 관계를 조사하여 그림 8의 알고리즘을 적용하여 스케줄한다. 자원 상충관계는 각 오퍼레이션 간의 IA,IB 형의 같은 필드를 점유하거나 fdr 래지스터를 같이 사용하는 경우에 발생한다. 그림 8에 나타난 바와 같이 현재 스케줄될 오퍼레이션과 함께 스케줄되어 병렬처리될 명령어가 여러 개인 경우에는 가장 빠른 클럭 사이클에서 실행되는 명령어와 함께 스케줄하여 차후에 현재 오퍼레이션과 데이타 종속관계가 있는 오퍼레이션들의 스케줄 시에 병렬처리의 가능성을 높인다.

Ⅳ. 시뮬레이션 및 성능 측정

1. C 컴파일러 및 시뮬레이터

구현된 C 컴파일러는 yacc와 C 언어를 사용하여 그림 9와 같이 구성하였다.

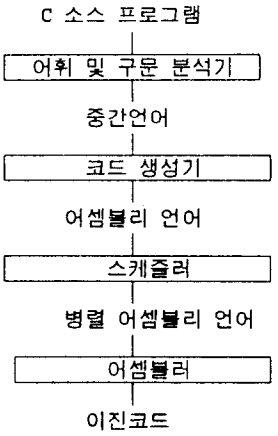

그림 9. C 컴파일러 구조
Fig. 9. Structure of a C compiler.

C 소스 프로그램이 어휘분석(lexical analysis)과 구문분석(syntax analysis)되어 3 주소 코드 형식의 중간언어를 생성한다. 생성된 중간언어는 코드 생성기(code generator)에 입력되어 레지스터 할당(register allocation)이 된 후에 어셈블리 코드가 생성된다. 스케줄러는 기본 오퍼레이션으로 구성된 어셈블리 코드를 입력받아 스케줄하고 어셈블러가 스케줄된 어셈블리 코드에 대한 목적코드를 생성한다.

```
int a = 2,b = 3,c = 4;
int d,e;
main()
{
   d = a + b;
   e = d + c * a;
}
```

(a)

```
$00006000  _a       .dw    2                          .00000002
$00006004  _b       .dw    3                          .00000003
$00006008  _c       .dw    4                          .00000004
$0000600c  _d       .dw    ?                          ?
$00006010  _e       .dw    ?                          ?
           main     .proc
$00000000           mov    r5   ,0
                    mov    r2   ,24572                .44a00000 .44405ffc
$00000008           mov    r4   ,24576
                    nop                               .44806000 .00000000
$00000010           push   r3                         .04030002 .04420002
$00000018                                             .54420004 .00000000
$00000020           add    r3   ,r2   ,4
                    nop                               .44620004 .00000000
$00000028           sub    r2   ,r2   ,0
                    nop                               .54420000 .00000000
$00000030           mov    r8   ,_a
                    mov    r10  ,_b                   .45000000 .45400004
$00000038           mov    r16  ,_c
                    lw     fdr  ,r8   ,r4             .46000008 .05482002
$00000040           mov    r13  ,_d
                    lw     fdr  ,r10  ,r4             .45a0000c .054a2002
$00000048           mov    r22  ,_e
                    mov    r9   ,fdr                  .46c00010 .05200001
$00000050           mov    r11  ,fdr
                    nop                               .05600001 .00000000
$00000058           add    r12  ,r9   ,r11
                    lw     fdr  ,r16  ,r4             .05895800 .05502002
$00000060           sw     r13  ,r4   ,r12            .040c0002 .044d2002
$00000068           mov    r17  ,fdr
                    nop                               .06200001 .00000000
$00000070           mul    r20  ,r17  ,r9
                    nop                               .26914800 .00000000
$00000078           add    r21  ,r12  ,r20
                    nop                               .06aca000 .00000000
$00000080           sw     r22  ,r4   ,r21            .04150002 .04562002
$00000088  @1
$00000088           add    r2   ,r2   ,0
                    nop                               .44420000 .00000000
$00000090           pop    r3                         .04420908 .0542090a
$00000098                                             .00000000 .00000000
$000000a0                                             .04600001 .00000000
$000000a8           stop                              .14000000 .cfe0ffff
$000000b0                                             .00000000 .00000000
$000000b8                                             .00000000 .00000000
$000000c0                                             .00000000 .00000000
           main     .endp
```

(b)

그림 10. 컴파일러 코드 생성 예
(a) 소스 프로그램 (b) 목적코드

Fig. 10. An example of code generation by compiler.
(a) source program (b) object code

그림 10은 (a)의 소스 프로그램에 대해 스케쥴되어 생성된 코드의 예 보여주는 그림이다.

시뮬레이터는 그림 11과 같이 컴파일러가 생성한 이진코드를 입력받아 로더가 메모리 화일을 생성한다. 메모리 화일은 제안된 아키텍처의 메모리를 시뮬레이션한 화일이다. 메모리 편집기 메모리 화일의 내용을 화면에 출력하거나 메모리의 내용을 편집한다. 시뮬레이터는 메모리 화일로 부터 명령과 데이타를 읽어들여 시뮬레이션하고 시뮬레이션 결과를 출력한다. 그림 12는 그림 10의 프로그램에 대하여 각 클럭 별로 시뮬레이터가 동작되는 상태 중 일부를 보여주는 그림이다.

2. 성능 측정 및 시뮬레이션

제안 설계된 병렬 파이프라인 프로세서의 동작을 검증하고 성능을 평가하기 위해서 다음과 같이 모두 7개의 벤치마크(benchmark) 테스트 프로그램의 3가지

측정유형에 대해서 코드를 생성하여 시뮬레이션하였다.

binsch 50개 정수의 이진탐색(binary search)
fib 30개 피보나치 급수 계산(Fibobacci series)
sieve 20개 소수 계산(Sieve of Erastosthenes)
short 5 노드 그래프의 최단경로 계산(shortest path)
fact 5 팩토리알 계산(factorial)
hanoi 5개 디스크 이동 하노이 탑(towers of hanoi)

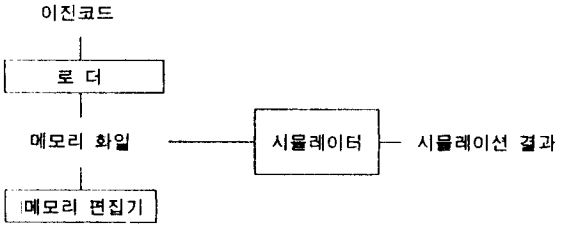

그림 11. 시뮬레이터 구성도
Fig. 11. Block diagram of simulator.

```
Initialization...
     open 'v_m_p.xxx'

clock 0
     SL0[0] SL1[0] : DL00[0] DL10[0] -> #0
     SL2[0] SL3[0] : DL20[0] DL30[0] -> #0 F_I[]
     IRRA NOP1
     IRRB NOP1
     PC=00000000, IRA=44A00000, IRB=44405FFC
clock 1
     SL0[0] SL1[0] : DL00[0] DL10[0] -> constant0
     SL2[0] SL3[0] : DL20[0] DL30[0] -> constant0 F_I[]
     IRIA #0 ADD #00000000h -> R5
     IRIB #0 ADD #00005FFCh -> R2
     PC=00000008, IRA=44806000, IRB=00000000

clock 9
     SL0[0] SL1[8] : DL00[8] DL10[8] -> R16
     SL2[0] SL3[6000] : DL20[4] DL30[6000] -> MAR F_I[]
     IRIA #0 ADD #0000000Ch -> R13
     IRRB R10 ADD R4 -> MAR BRM4
     PC=00000048, IRA=46C00010, IRB=05200001
clock 10
     Local READ : Address=00006000,data=00000002,00000003
     SL0[0] SL1[c] : DL00[8] DL10[c] -> R13
     SL2[4] SL3[6000] : DL20[4] DL30[6004] -> MAR F_I[]
     IRIA #0 ADD #00000010h -> R22
     IRRB #0 ADD FDR0:0 -> R9
     PC=00000050, IRA=05600001, IRB=00000000
clock 11
     Local READ : Address=00006004,data=00000002,00000003
     SL0[0] SL1[10] : DL00[10] DL10[10] -> R22
     SL2[0] SL3[2] : DL20[4] DL30[2] -> R9 F_I[]
     IRRA #0 ADD FDR0:0 -> R11
     IRRB NOP1
     PC=00000058, IRA=05895800, IRB=05502002
clock 12
     SL0[0] SL1[3] : DL00[10] DL10[3] -> R11
     SL2[0] SL3[0] : DL20[2] DL30[2] -> constant0 F_I[]
     IRRA R9 ADD R11 -> R12
     IRRB R16 ADD R4 -> MAR BRM4
     PC=00000060, IRA=040C0002, IRB=044D2002

program normal terminate at Processor 0..
                    estimated clock 25..
```

그림 12. 시뮬레이션 결과
Fig. 12. The results of simulation.

quick 10개 정수의 퀵정렬(quick sort)

유형 I 스케줄하지 않음
유형 II 파이프라인 스케줄만 적용
유형 III 파이프라인,병렬 스케줄 모두 적용

표 8. 테스트 프로그램 측정
(a) 명령어 수 정적측정 (b) 수행시간의 동적측정
Table 8. Test programs measurements.
(a) static measurements of instruction counts (b) dynamic Measurements of execution times.

(a)

프로그램	I 명령어			II 명령어			III 명령어		
	dnop	cnop		dnop	cnop		dnop	cnop	
binsch	127	14	30	107	1	19	86	3	13
fib	106	11	18	91	0	11	71	0	8
sieve	210	20	51	175	2	34	148	1	28
short	334	38	72	276	7	45	248	7	45
fact	71	4	18	61	2	10	52	1	10
hanoi	105	11	12	94	1	11	77	1	11
quick	318	40	54	271	2	45	231	1	45

(b)

(단위 : clocks)

프로그램	I	II	III
binsch	472	378	312
fib	2146	1678	1290
sieve	4943	4107	3505
short	2466	2302	1886
fact	263	232	205
hanoi	4007	3664	3221
quick	3662	3134	2820

테스트 프로그램은 일반적으로 널리 알려진 프로그램을 사용되는 프로그램을 선택하였다. 이 중 fact, hanoi, quick은 재귀함수(recursive function)로 작성된 프로그램이다. 이들 프로그램에 대해서 컴파일하여 코드를 생성한 후에 3가지 유형으로 시뮬레이션하였다. 유형 I는 생성된 코드에 대하여 어떤 스케줄도하지 않은 유형이고, 유형 II는 데이타 해저드,제어 해저드로 인하여 삽입된 nop를 제거하기 위해서 파이프라인 스케줄만을 적용하였다. 유형 I,II는 병렬처리 기능이 없는 기존의 전형적인 RISC 프로세서의 유형이다. 유형 III은 유형 II의 결과에다가 최대 4개의 오퍼레이션들이 병렬처리되도록 스케줄한 유형이다. 이들 유형에 대하여 스케줄하고 시뮬레이션한 측정결과를 표 8에 보였다. 표 8에서 (a)는 컴파일러에 의해 생성된 코드에 대하여 스케줄하여 명령어의 수와 nop의 수를

정적측정(static measurements)한 결과이다. 여기서 dnop 데이타 해저드로 인하여 삽입된 nop이고 cnop는 제어 해저드로 인하여 삽입된 nop이다. (b)는 이를 시뮬레이션하여 수행시간을 클럭 수로 동적측정(dynamic measurements)한 결과이다. 표 8에서와 같이 파이프라인 스케줄만을 한 II의 경우에는 nop의 감소로 명령어의 수는 약 15%가 감소하고 병렬 스케줄을 함께한 III의 경우에는 약 30% 감소하였다.

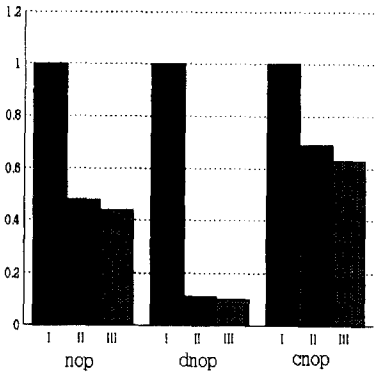

그림 13. nop 수의 비
Fig. 13. Ratio of nop numbers.

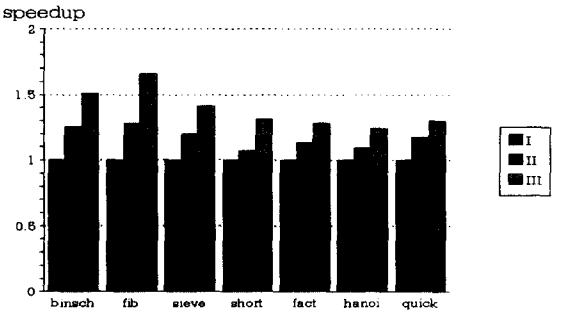

그림 14. 성능 비
Fig. 14. Ratio of performance.

nop의 수는 그림 13과 같이 II의 경우 약 58%가, III의 경우에는 63%가 감소하였다. II,III의 경우 약간 차이가 나는 이유는 III의 병렬 스케줄 시 코드의 상향 이동으로 이들 이동한 명령과 데이타 종속관계가 있는 명령들이 nop 위치에 채워질 가능성이 증가하였기 때문이다. 그러나 nop 제거의 대부분은 데이타 해저드인 경우이고(약 90%) 제어 해저드인 경우에는 약 37% 정도 밖에 스케줄되지 않았다. 이는 3개의 분기 지연슬롯을 데이타 종속관계가 없는 명령들로 모두 채워지지 않았기 때문이었다.

그림 14는 I의 경우를 1로 하였을 때 II,III에 대한 각 프로그램의 성능(수행시간의 역수) 비를 나타낸 그림으로 II의 경우에는 약 17%, III의 경우에는 약 39% 정도의 성능이 향상되었다. 즉 프로세서를 이중 명령어 구조로 설계하고 이를 하드웨어가 아닌 소프트웨어 스케줄만을 하여 39% 정도의 성능개선이 측정되었다.

V. 결 론

본 논문에서는 VLIW 아키텍처의 개념을 도입하여 단일칩이 작은 VLIW 프로세서와 같은 동작을 하는 병렬 파이프라인 구조 프로세서의 기능레벨(functional level)과 명령어 아키텍처를 설계하였고, 이 아키텍처 상에서 병렬 처리되는 명령어를 추출하고 스케줄링하는 스케줄링 알고리즘을 제안하였다.

제안된 병렬 파이프라인 프로세서 아키텍처는 한 사이클 당 두개 이상의 기본 오퍼레이션을 동시에 수행하고 각 오퍼레이션 간에는 4단 파이프라인으로 처리되도록 설계하였다. 또한 이들 기본 오퍼레이션의 조합으로 응용목적에 따라 다양한 기능을 하는 명령어의 설계가 가능하여 광범위한 응용에 적용할 수 있는 융통성을 갖는다. 제안된 스케줄링 알고리즘은 수행되는 프로그램의 데이타 종속관계 및 자원 상충관계를 조사하여 병렬처리되는 오퍼레이션들을 스케줄하고 파이프라인 해저드를 제거하였다.

설계된 병렬 파이프라인 프로세서 아키텍처와 스케줄링 알고리즘의 동작 검증을 위하여 C 컴파일러와 시뮬레이터를 개발하여 널리 사용되는 7개의 테스트 프로그램에 적용하여 시뮬레이션 하였다. 시뮬레이션 결과 스케줄하지 않았을 경우 보다 이중 명령어 구조의 특성을 이용하여 스케줄 하였을 경우에 약 40% 정도 성능이 향상되었다.

앞으로 대규모 응용 프로그램의 병렬처리를 위하여 제안된 프로세서를 여러개 연결하여 전형적인 VLIW 아키텍처로 동작하는 멀티프로세서 시스템을 설계하고, 이를 위한 광역 스케줄링 알고리즘(global scheduling algorithm)을 연구 개발할 예정이다. 또한 이와 병행하여 제안된 프로세서 아키텍처의 게이트 레벨(gate level) 설계를 추진할 계획이다.

참 고 문 헌

[1] J. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction", IEEE Transaction on Computers, Vol.C-30 No.7, pp.478-490, July 1981.

[2] J. Fisher, "Very Long Instruction Word

Architectures and the ELI-512", ACM Proc. 10th International Symposium on Computer Architecture, pp.140-150, June 1983.

[3] J. Ellis, J. Fisher, J. Ruttenberg, and A. Nicolau, "Parallel processing : A smart compiler and dumb machine", ACM Proc. SIGPLAN '84 Symposium on Compiler Construction, pp. 37-47, June 1984.

[4] S. Weiss and J. Smith,"Instruction Issue Logic in Pipelined Supercomputers" , IEEE Transaction on Computers, Vol. C-33 No.11, pp.1013-1022, Nov. 1984.

[5] Y.I. Cho, S.J. Lee, J.D. Park and I.C. Lim, "A Technique for Global Microcode Compaction", IEEE International Symposium on Circuits and Systems(ISCAS), pp.1051-1054, June 1985.

[6] A. Nicolau, " Percolation Scheduling: A Parallel Compilation Technique", Technical Report TR85-678, Department of Computer Science Cornell Univ., May 1985.

[7] M. Annaratone et. al., "The Warp Computer: Architecture,Implementation, and Performance", IEEE Transaction on Computers, Vol.C-36 No.12, Dec. 1987.

[8] R. Colwell, R. Nix, J. O'Donnell, D. Papworth and P.Rodman , "A VLIW Architecture for Trace Scheduling Compiler", IEEE Transaction on Computers, Vol.C-37 No.8, pp. 967-979, August 1988.

[9] M.Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines", ACM Proc. SIGPLAN '88 Conference on Programming Language Design and Implementation, pp.318-328, June 1988.

[10] J. Smith, "Dynamic Instruction Scheduling and the Astronautics ZS-1", IEEE Computer, July 1989.

[11] J. Ellis, Bulldog: A Compiler for VLIW Architectures, The MIT Press, 1986.

[12] J. Hennessy and D. Patterson, Computer Architecture a Quantitative Approach, Morgan Kaufmann Publishers Inc., 1990.

[13] M. Danelutto and M. Vanneschi, "VLI-W-in-the-large: a model for fine grain parallelism exploitation on distributed memory multiprocessors", IEEE 23rd Annual Workshop on Microprogramming and Microarchitecture, pp.7-16, Nov. 1990.

[14] R. Jones and V. Allan, "Software Pipelining: A Comparision and Improvement", IEEE Proc. 23rd Annual Workshop on Microprogramming and Microarchitecture, pp.46-56, 1990.

[15] M. Johnson, Superscalar Microprocessor Design, Prentice Hall, 1991.

[16] A. Wolfe and J. Shen, "A Variable Instruction Stream Extension to the VLIW Architecture", Proc. 4th Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV), pp.2-14, 1991.

[17] A. Fukuda and S. Tomita, "Toward Advanced Parallel Processing: Exploiting Parallelism at Task and Instruction Levels", IEEE Micro, pp.16-61, August 1991.

[18] J. Grout, Instruction-Level Parallelism Research, Technical Report ,Center for Supercomputing Research and Development Univ. of Illinois at Urbana-Champaign, Dec. 1992

[19] D. Alpert and D. Avnon, "Architecture of the Pentium Microprocessor", IEEE Micro, pp.11-21, June 1993.

[20] E. McLellan, "The Alpha AXP Architecture and 21064 Processor", IEEE Micro, pp.36-47, June 1993.

[21] M. Becker et.al., "The PowerPC 601 Microprocessor", IEEE Micro, pp.54-68, Oct. 1993.

저 자 소 개

李 相 靜(正會員)

1960년 6월 3일생. 1983년 2월 한양대학교 전자공학과 졸업(공학사). 1985년 2월 한양대학교 대학원 전자공학과 졸업(공학석사). 1985년 8월 한양대학교 대학원 전자공학과 졸업(공학박사). 1988년 9월 ~ 현재 순천향대학교 전산학과 교수. 주요관심분야는 VLIW/슈퍼스칼라 아키텍처 설계, 최적화 컴파일러 설계, 마이크로프로그래밍 등임.

金 光 駿(正會員)

1966년 9월 24일생. 1990년 2월 한양대학교 전자공학과 졸업(공학사). 1992년 2월 한양대학교 전자공학과 졸업(공학석사). 1992년 3월 ~ 현재 (주)대우 반도체 연구소 연구원. 주요관심분야는 프로세서 아키텍처 설계, 신경회로망 등임.