論文 90-27-8-9

RISC 아키텍춰의 코드 생성기 설계

(The Design of A Code Generator for RISC Architectures)

朴 鍾 得*, 林 寅 七*

(Jong Deuk Park and In Chil Lim)

要 約

본 논문에서는 컴파일러의 설계에 있어서 RISC 머신을 위한 코드 생성 기법과 정수형 상수 곱셈의 효율적인 처리 알고리듬을 제안한다.

RISC 아키텍춰는 CISC에 비해 실행시간이 빠르고 형식이 간단한 명령어들을 사용하며 대부분의 RISC 프로세서가 정수 곱셈처리 명령어를 내장하지 않고 있으므로 정수형 곱셈을 효율적으로 처리할 수 있는 알고리듬이 요구된다.

제안된 코드 생성기의 구성을 위해 Portable C Compiler(PCC)를 RISC 머신에 적합하도록 설계하고 코드 생성 과정에서 발생빈도가 높은 정수 곱셈의 일부인 상수에 의한 곱셈에 대하여 보다 빠른 처리가 가능한 덧셈 체인을 형성한다.

Abstract

This paper presents a code generation method and an effective handling algorithm of integer constant multiplication for RISC machines in compiler design.

As RISC Architectures usually use faster and more simply formed instructions than CISC's and most RISC processors do not have an integer multiplication instruction, it is required an effective algorithm to process integer multiplication.

For the proposed code generator, Portable C Compiler (PCC) is redesigned to be suitable for an RISC machine, and composed an addition chain is built up to allow fast execution of constant multiplication, a part of integer one which appears very frequently in code generation phase.

I. 서 론

오늘날 컴퓨터 설계자는 반도체 집적회로 기술의 급속한 발전에 힘입어 프로세서 설계시에 명령어 세트의 복잡도를 더욱 증가시키고 있다. 그러나 명령어 세트의 다양화만으로는 컴퓨터 시스템의 성능 향상에 그다지 도움이 못된다는 사실이 차츰 인식되고 있다. 이에 때맞춰 등장한 RISC 프로세서는 코드의 실행속도를 10MIPS이상으로 끌어 올리는 현저한 성능 향상을 보여 주고 있다. 최근에 RISC 프로세서 IC는 실시간 화상 처리, 디지탈 신호 처리, 슈퍼컴퓨터에의 응용을 위한 핵심적인 CPU로서 각광을 받고 있다[1],[2]

RISC는 하드웨어의 단순화를 지향하여 대부분의 명령어가 단일 사이클내에 실행되고, 명령어 형식을

*正會員, 漢陽大學校 電子工學科
(Dept. of Elec. Eng., Hanyang Univ.)

接受日字 : 1990年 5月 14日

고정시킴으로써 명령어 디코드 시간이 줄어들며, 단일 어드레싱 모드로 메모리를 액세스한다. 따라서 RISC는 기존의 방식보다 많은 스테이지를 갖는 강력한 파이프라인(pipeline)의 실현을 가능케 하여 컴퓨터의 성능을 대폭 향상시킬 수 있게 된다.

RISC 기술의 주요한 목표중의 하나는 수행 성능을 높이는데 있다. 이의 실현을 위해서는 프로세서 아키텍춰의 설계뿐만 아니라 고성능 컴파일러의 설계가 필요하다. 이러한 컴파일러 설계시, 다음과 같은 문제를 고려해야 한다. 첫째로, 기존에 사용하던 머신 아키텍춰와 비교해 볼때, 하드웨어적으로 제한(예를 들면, 명령어 사이즈가 일정하고 단순화된 적은 수의 명령어 집합을 갖고 있다는 점 등)이 많다는 점에서 큰 차이를 보이지만, 기존 아키텍춰에서 사용되어 오고 있는 고급 언어의 연산이나 명령어의 수행과 동일한 결과가 나올 수 있도록 몇개의 단순한 명령어들로 조합 구성하여 수행하여야 하고, 둘째로는 기존의 아키텍춰에는 없지만 수행 성능을 뛰어나게 하기 위하여 특별한 기능(예를 들면, 레지스터 윈도우 등)을 가진 하드웨어가 첨가된 머신 아키텍춰에는 없었던 특수한 기능을 잘 이해하여 효율적으로 프로그램을 수행할 수 있도록 해야한다. 따라서, 새로운 아키텍춰에서 프로그램 수행 성능을 높이려면, 컴파일러의 전반부(front-end)인 파서(parsor)와 후반부(back-end)인 코드 생성기중에서 기계 독립적인 파서도 중요하지만 기계 종속적인 부분인 코드 생성기의 설계에 중점을 두어야 한다.

코드 생성기 설계와 관련된 기존의 설계 방식은 원시언어에 있는 각 연산자와 피연산자에 대하여 계산을 수행하는 일련의 대상 명령어를 출력하는 루틴들의 집합을 구성하는 것이었다. 그러나 새로운 아키텍춰의 계속적인 출현으로 각 아키텍춰를 대상으로 한 테이블-유도 분석 방법(table-driven analysis method)과 같이 아키텍춰 의존 부분의 정보를 따로 갖도록 하는 이식성이 높은 컴파일러에 대한 설계 방식이 연구되어 왔다. 현재는 코드생성기를 자동화하려는 노력의 일환으로 코드 생성기-생성기 (code generator-generator)에 관한 연구가 활발히 진행되고 있다.[5],[6],[7]

본 논문에서는 RISC 머신 코드 생성기에서 코드 생성 수행이 가능하도록 CISC 컴파일러인 PCC를 RISC머신에 적합하게 재설계하여 사용자 프로그램이 수행될 수 있도록 하고, 프로그램 수행시 내재된 연산들 중 특히 발생빈도가 높은 정수형 상수 곱셈에 대하여 보다 빠르게 처리할 수 있는 알고리듬을 제안한다. 또한 제안된 알고리듬을 상용 RISC 머신인 SUN SPARC station의 정수 곱셈 처리부와 비교 검토하여 그 효율성을 입증한다. 제안된 코드 생성 기법과 정수형 상수 곱셈 처리 알고리듬은 SUN SPARC station(UNIX BSD 4.3)상에서 C언어로 구현한다.

Ⅱ. 코드생성의 개요

컴파일러는 특정 원시 언어로 쓰여진 프로그램을 대상 머신에서 실행가능한 코드로 바꾸어 주는 번역기이다. 컴파일러의 설계가 지난 수 십년동안 연구되어 왔고, 많은 컴파일러가 만들어졌지만 아직도 만족스럽게 해결되지 않는 설계 문제들이 많은 실정이다. 이들 중의 하나가 코드 생성 과정이다. 코드 생성기는 컴파일러의 마지막 단계로서 그림 1 처럼 원시 프로그램의 중간 표현을 입력으로 받아 대상 머신 프로그램을 출력한다. 코드 생성 기법은 그 이전 단계에서 기계 독립적인 최적화의 수행여부에 관계없이 사용될 수 있다.

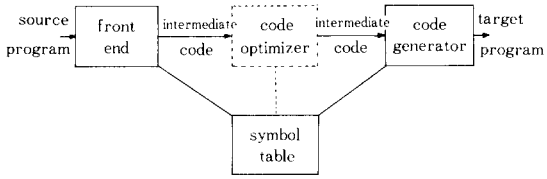


그림 1. 컴파일러의 구조
Fig. 1. The structure of compiler.

코드 생성기의 출력 코드는 원시 언어의 의미가 그대로 전달되어야 하며, 효율적인 시퀀스로 구성되어야 한다. 이것은 대상 머신의 자원이 효과적으로 사용되도록 설계되어야 하고 코드 생성 과정이 효율적으로 실행되어야 함을 의미한다.

코드 생성에 대한 기존의 설계 방식은 원시 언어에 있는 각 연산자와 피연산자에 대해 계산을 수행하는 일련의 대상 명령어(target instruction)를 생성하는 루틴들의 집합을 제공하는 것이었다. 코드 생성 과정에 포함된 부분으로는 중간 결과를 위한 기억영역 할당, 레지스터 관리(register management), 코드 최적화(code optimization) (길고 느린 계산을 동등한 기능을 가진 더 짧고 빠른 계산으로 바꾸는 것), 그리고 명령어의 선택등이 있다. 원시 언어로 부터 대상 머신으로 매핑하는데 있어 복잡성과 다양한 효

율성의 필요때문에 코드 생성기는 방대하고 복잡한 프로그램이 된다. 게다가 코드 생성기의 많은 부분이 임의의 형식으로 씌여졌기 때문에 이를 디버그(debug)하고 수정하고 관리하기 어렵다. 컴파일러는 원시 언어와 대상머신 양쪽의 구조와 의미에 대해서 상당한 지식을 포함하므로, 원시 언어와 대상 머신이 새로 나올때 마다 새로운 컴파일러가 만들어져야 한다. 이에 따라 컴파일러 설계를 자동화하려는 시도가 계속되어 왔다.

최근에 시스템 소프트웨어 설계자들은 컴파일러의 후반부로 관심을 돌리고 있으며 특히, 코드 생성을 자동화 하는 도구를 만들기 위해 테이블 유도 방법에 주목하고 있다. 코드 생성에 대한 이러한 접근방식은 테이블 유도구분 분석에서와 같이 많은 잇점들을 가지고 있다. 즉, 신뢰성이 있고, 사용하기 쉬우며, 컴파일러 작성 도구에서 요구되는 융통성이 있다는 것이다.

자동화 코드 생성에 대한 연구는 대략 다음의 3종류로 분류할 수 있다.

- interpretive코드 생성
- pattern-matched 코드 생성
- table-driven 코드 생성

Interpretive 코드 생성은 가상(virtual) 머신을 위한 코드를 생성하고 나서 실제 대상 코드로 확장한다. pattern-matched 코드 생성은 코드 생성 알고리듬으로부터 대상 머신을 기술한 머신 표현을 분리한다. table-driven코드 생성은 pattern-matched 코드 생성 방식을 향상시켜 자동화시키는데 중점을 두었다. 그것은 형식을 갖춘 머신 표현을 쓰고, 자동적으로 코드 생성기들을 만들기 위해 코드 생성기-생성기를 사용한다.[5],[6],[7]

테이블 유도 구문 분석기를 사용함으로써 새로운 머신에 대한 명령어를 기술한 문법과 아울러 대상 머신의 표현을 줌으로써 새로운 머신에 대한 코드를 생성하는데 있어 기존의 코드 생성기를 재 대상화(retargetting) 할 수 있다.

Ⅲ. 템플리트(template)의 구성

제안된 코드생성기는 VAX 머신에서 사용되는 PCC의 템플리트-매칭 알고리듬을 기초로 하여 설계된다. PCC는 UNIX 시스템에서 사용하는 프로그래밍 언어 C를 번역하기 위하여 만들어 졌다. PCC는 S.C. Johnson에 의하여 쓰여졌는데, 이식이 용이하여 12개 이상의 CISC 머신에서 성공적으로 이식되었다.

PCC는 테이블-유도 방식 코드 생성 방법을 사용한다. 코드 생성기의 후반부에서는 입력단위가 하나의 트리 표현이 되는데, Sethi-Ullman 값을 계산하여 중간 계산값이 필요하면 전체 트리 중에서 중간값이 필요한 지점을 루트로 하는 부트리들에 대한 코드를 먼저 생성한다.

PCC의 입력단위는 트리이므로 코드를 생성하기 위해서는 레지스터 할당 문제도 염두에 두어야 한다. 즉 트리가 타겟머신의 한 명령어보다 복잡한 표현식일 경우에는 여러개의 코드 시퀀스로 출력되므로 중간계산결과를 저장하는 레지스터의 선택을 고려할 필요가 있다. PCC는 Sethi-Ullman 값을 트리의 각 노드마다 계산하여 중간계산 결과를 필요로 하는 노드를 검출하고 이 노드를 루트로 하는 부트리에 대한 코드를 먼저 생성한 후 그 계산 결과를 이용하는 표현식의 코드를 출력시킨다. 이 코드생성기는 템플리트의 집합으로 구성되는 테이블의 사용한다. 각 템플리트는 입력 트리에 매치될 수 있는 패턴을 가지고 있다. PCC는 두 개의 패스로 구성되어 있는데 첫번째 패스에서 프로그램의 중간표현을 생성하고 나서 두 번째 패스에서 코드를 생성한다. 두 번째 패스는 얼마나 많은 템포러리 레지스터를 필요로 하는지를 계산하고 트리의 어떤 부분이 그 템포러리 레지스터의 사용을 요구하는지를 파악한다. 이 렇게 하는 이유는 머신에 따라서는 충분한 레지스터가 없을 뿐더러 프로시쥬어 호출시 인수를 위한 값을 리턴하는 경우가 존재하기 때문이다. 한 트리의 탐색중에 임시 결과의 저장없이 계산이 가능한 부트리가 발견되면 바로 그 트리를 계산하는 코드를 출력하는 루틴이 호출된다. 코드 출력 루틴에서는 쿠키 값에 따라 매핑될 트리에 합당한 템플리트를 템플리트 테이블을 통해 찾는다. 만일 매치가 이루어지지 않으면 트리가 재구성되는데 테이블은 트리의 재구성 정보를 가지고 있어 이를 넘겨 주어서 다음번에는 매칭이 이루어지도록 한다. PCC와 같이 코드를 생성하는 루틴을 설계하려면 다음 사항을 고려해야 한다.

1. 어떤 트리를 입력으로 받으면 필요한 임시 변수의 레지스터 수를 계산하고 트리의 어떤 위치(노드)에서 필요로 하는 지도 계산한다. 대상 어셈블리 코드의 어드레스 모드를 단항 고려하여 연산자인 '*'의 자손 노드가 레지스터인 경우 두 노드를 하나의 노드로 결합하여야 한다. 또한 '++'나 '--'가 변수 뒤에 나타나는 경우는 이것을 포함하는 전체 트리가 계산되고난 후 증감되는 코드를 생성하도록 코드 나열 순서를 조정할 필요가 있다.

2. 이후 코드를 생성하기 위해서는 코드의 성질

(쿠키)을 고려하여 이 중 한 부류를 선택하여야 한다. 쿠키 값은 다음과 같이 분류될 수 있다.

1) 결과를 레지스터에 저장.
2) 조건 코드를 세트.
3) 계산 결과를 저장하지 않고 계산된 효과만을 이용.
4) 계산 결과를 메모리에 저장
5) 프로시쥬어(procedure)의 인수(argument)로의 취급.

3. 템플리트 테이블 엔트리와 견주어 2)의 쿠키와 맞아 떨어지는 템플리트와 매칭시킨다.

4. 매치되지 않을 경우, 매치가 이루어지도록 입력 트리를 재구성한다.

이와 같이 PCC에서는 코드 생성시 레지스터 할당을 동시에 고려하나 여기에서는 별도의 레지스터 할당과정 수행 후 코드를 생성하므로 레지스터 할당 문제는 고려하지 않는다.

제안된 코드 생성기는 대상 머신의 코드 정보를 템플리트라고 하는 패턴이 기술된 하나의 테이블에 수록한다. 코드 생성기는 트리 표현을 입력으로 받아 템플리트-매칭 방법에 따라서 매치가 이루어지면 코드가 발생된다.

각 템플리트는 5가지의 정보를 가지고 있다.
다음 템플리트의 예를 생각하여 보자.

```
ASGPLUS,    INAREG,
SAREG,      TINT,
SNAME,      TINT,
            0,      RLEFT,
            "       add     AL, AR\n",
```

첫 줄의 라인은 연산자("+=")와 쿠키(cookie)를 나타낸다. 연산자는 트리의 각 노드가 가지고 있는 토큰 중의 하나이다. 쿠키는 계산 결과가 어떤 부류의 기억장치(예를 들면 레지스터나 메모리)에 저장해야 하는 지를 나타낸다. 두번째와 세번째 줄은 "+=" 연산자의 각각 왼쪽과 오른쪽자손(descendant)를 나타낸다. 왼쪽 자손은 레지스터 노드를 나타내고, 정수타입(integer type)을 가진다. 마찬가지로 오른쪽 자손은 identifier를 나타내고, 정수 타입이다. 4번째 줄은 자원(resource) 요구에 관한 사항으로 임시변수의 필요 여부를 제시한다. 최종 줄상의 문자열은 어셈블리 출력코드를 표현한다. 이중 따옴표 내의 문자열은 한 글자 씩 순서대로 인쇄되는데, 소문자, tab, space 등은 그대로 출력하며, 대문자는 매크로(macro)처럼 다양하게 확장된다. 대개 AL은 왼쪽 피연산자의 주소를 나타내는데, 레지스터 번호인 경우가 많다. 유사하게 AR은 오른쪽 피연산자의 이름(name)으로 확장된다. 즉 대문자로 읽혀지면 별도의 프로그램이 코드생성부로 넘어온 트리 정보를 참조하여 경우에 따라 합당한 코드를 인쇄한다.

연산자, 쿠키, 타입 등에 따라 모든 조합을 별개의 템플리트로 만들 수도 있다. 이렇게 되면 경우의 수가 너무 많아지게 되므로 단일 템플리트에 의해 가급적 많은 종류의 트리 매칭이 가능하도록 한다.

코드 생성기의 중심부는 템플리트-매칭 알고리듬으로서, 주어진 트리를 쿠키에 그 트리를 번역할 어떤 템플리트와 그 트리를 매치시키려 시도한다. 매치가 성공되면 바로 출력 코드가 생성된다. 템플리트를 트리에 매치하기 위해서는 쿠키와 루트의 연산자 뿐만 아니라 트리의 오른쪽, 왼쪽 자손의 타입과 형태를 매치하는 것도 필요하다.

하나의 트리가 어떤 템플리트에 바로 매치되기는 어려우며 대개는 부트리의 중간 결과를 레지스터에 저장하기 까지의 코드가 발생된 후 이 레지스터의 값을 읽어서 다음 계산을 수행하는 코드를 생성하게 된다. 그러나 본 코드생성기에서는 Sethi-Ullman 값을 사용하여 전체 트리를 취급하지는 않으며 별도의 레지스터 할당 과정 수행 후의 중간코드를 입력으로 한다. 코드생성기에 입력된 중간코드는 바로 SPARC 어셈블리코드를 위하여 설계된 템플리트의 집합과 매치를 시도한다. 템플리트의 테이블에 대한 효과적인 탐색을 위해 연산자와 바로 매치가 될 수 있도록 하였으며 다만 같은 연산자라고 하더라도 오퍼랜드의 종류에 따라 하나의 템플리트로는 완전히 표현하지 못하는 경우가 있으므로 여러 개의 템플리트를 준비하고 있다. 따라서 매치가 이루어지지 않으면 다음 번지에 있는 같은 연산자의 템플리트로 넘어가게 된다. 매치가 이루어지면 바로 어셈블러 코드를 출력하게 되는데 경우에 따라 여러가지 형태로 마크로 확장이 된다. 한편 기계독립적인 부분인 표현식들은 템플리트로 취급이 가능하지만, 기계 종속적인 부분(함수의 시작을 위하여 스택의 번지를 지정하는 경우 등)은 컴파일러의 파서에서 직접 취급하여 프린트 문의 형태로 인쇄가 된다. 그 밖에 고려할 사항들은 CISC 머신에 있는 특별한 동작을 하는 단일 명령어(예를 들면, VAX 명령어중에서 switch 문 처리를 위한 명령 "casel")를 간결한 스텝으로 대치하기 위한 명령어 구성과 메모리 관리(예를 들면, 프레임 사이즈를 계산할 때 레지스터 윈도우의 overflow 발생을 막기 위한 레지스터 back-up 장소도 함께 계산) 등이 있다.

PCC를 RISC 타입 머신에 맞는 컴파일러로 재대상화(retargetable)하려면, 컴파일러의 머신 독립적

인 부분인 전반부보다 머신 종속적인 부분인 후반부를 대상으로 해야 한다. 즉, 템플리트와 그것을 통해 코드를 출력하는 일부 루틴들의 집합체를 전부 수정하여야 한다.

다음 템플리트를 생각하여 보자.

```
ASG LS, INAREG|FOREFF|FORCC,
        SAREG|AWD,      TWORD,
        SAREG|NIAWD,    ANYSIGNED|ANYUSIGNED,
                0,      RLEEF|RESCC,
                "       ashl    AR, AL, AL/n",
```

위 예제에서는 opcode가 ASG LS (C 언어에서 '=≪')인 템플리트를 나타내고 있다. 이 템플리트에서 출력되는 코드는 "ashl operand1, operand2, operand3\n"인데, 이 코드에서 'ashl'이라는 명령어는 'arithmetic shift longword'로 첫번째 피연산자는 count를 나타내고, 두번째, 피연산자는 소스(source)를 나타내고, 세번째 피연산자는 destination을 나타낸다. 여기서 count 값으로 음수나 양수가 나타날 수 있는데, 양수인 경우는 왼쪽으로 쉬프트하고 음수인 경우에 오른쪽으로 쉬프트한다. count 값은 트리와 매치될 때, 음수값인지 양수값인지를 알 수 없기 때문에 다음과 같은 템플리트를 통하여 확장 처리할 수 있다.

```
ASG LS, INAREG|FOREFF|FORCC,
        SAREG|AWD,      TWORD,
        SAREG|NIAWD,    ANYSIGNED|ANYUSIGNED,
                0,      RLEEF|RESCC,
                "       ZQ      AL, AR, AL\n",
```

위의 템플리트와 같이 ZQ라는 대문자를 출력 코드 부분에 삽입하므로써 매크로로 확장 처리될 수 있다. 다음은 ZQ에 대해 확장한 프로그램이다.

```
zzzcode(p, c) register NODE *p; {
        switch(c) {
                case 'Q' :/*determine shift direction*/
                        {
                        int value;
                        value=p->in.right->tn.lval;
                        if (value>0) {
                                putstr("sll");
                                p->in.right->tn.lval=value;
                        }
                        else {
                                putstr("sra");
                                p->in.right->tn.lval = -(value);
                        }
                        return;
                        }
                default;
                        cerror("illegal zzzcode");
        }
}
```

Ⅳ. 정수 곱셈 처리

일반적으로 머신을 설계할 때, 하드웨어를 과잉설계(overdesign)하거나 축소설계(underdesign)하는 경향은 가격/성능 비를 나쁘게 하기 때문에 피하여야 한다. 그러나 부동 소수점 연산, 10진 연산, 대형 블럭(block) 이동, 정수의 곱셈과 나눗셈 연산 등은 명시적 발생빈도가 적을지라도 머신 설계에 있어서 깊이 고려할 필요가 있다. 그 이유는 프로그래머가 정의하지 않은 연산들이 머신에 내재되어 여러번 수행되고 있기 때문이다.

정수 곱셈 처리를 하는데 있어, 곱셈 명령 하드웨어가 없는 RISC 머신을 위해 설계된 코드 생성기에서는 소프트웨어 루틴 호출을 이용하여 곱셈 명령어 대신에 몇개의 덧셈과 뺄셈, 쉬프트 명령어등의 코드로 바꾸어 수행토록 하여야 한다. 따라서 곱셈을 대치하여 생성된 코드의 길이는 프로그램의 실행속도를 좌우하게 된다.[14]

대부분의 프로그램들은 곱셈을 직접 또는 간접적으로 여러번 사용한다. 거의 대부분의 고급 언어들은 명확한 연산자(예, "*")로 이같은 연산자들을 직접 지원하고, 내재되어 요구되는 곱셈의 구성을 거의 전부 지원한다.

예를 들면, C에서 2차원 배열 구조를 액세스하기 위해서는

m=structure M[x][y].n

다음과 같은 2개의 내재된 곱셈을 요구한다.

((x*y max)+y)*sizeof(structure M))

(여기서 y max는 2차원의 최대 범위를 선언한 것이다.)

잘 설계된 컴파일러에서는 "strength reduction"이라 불리우는 기법을 사용하여 프로그램에서 곱셈 연산의 횟수를 많이 감소시키고 있다. strength reduction은 증분(increment)에 의한 덧셈이나 덧셈에 의하여 곱셈을 대치함으로써 구체화 된다. 왜냐하면 덧셈은 곱셈보다 비용면에서 작기 때문이다. 예를 들면,

```
for (n=0 ; n<10 ; n=n+1)
        a=a+n*25
```

이와같이 간단한 예에서, 25에 의한 곱셈은 25를 n 횟수 만큼 더하는 것에 의해서 대치될 수 있다.

프로그램상에서 곱셈은 다른 명령어에 비해서 발생빈도가 비교적 적은 편이다. Gibson mix(IBM 704 아키텍춰에 기초)는 곱셈의 발생빈도를 전체 명령어의 0.6%로 측정하였다.[10] 서로 다른 예제 프로그램으로 곱셈의 발생빈도를 폭넓게 측정한 다른 연구에 의하면, 곱셈이 0.0%와 2.5%사이의 발생빈도를 보

여주고 있다.[11],[12] 이러한 결과는 머신을 설계하는데 있어, 곱셈 명령 하드웨어를 특별히 고려할 필요가 없음을 발생빈도에서 알 수 있다.

곱셈의 분류에는 상수에 의한 곱셈과 변수에 의한 곱셈 두가지가 있다. 그리고 그 두가지 중에서도 곱셈의 타입(type)이 정수와 부동소수점에 따라 처리방법이 다를 것이다.

곱셈의 피연산자의 상수를 곱셈하는 경우와 변수를 곱셈하는 경우로 나눌 수 있다. 프로세서가 곱셈 명령어를 가지고 있지 않다면, 변수에 의한 곱셈처리는 다음과 같은 간단한 연산들을 사용하여 구성할 수 있다.

- an ADD operation
- an arithmetic right shift operation, and
- a bit test(or test for odd) instruction

이 간단한 알고리듬과 함께, 승수의 비트들은 LSB로부터 그 비트가 1이면, 피승수가 결과 레지스터에 더해지고, 그 비트가 0이면 결과는 그대로 둔다.

부동 소수점 타입에서의 연산은 CISC 머신의 경우, 일반적인 곱셈 하드웨어 명령 "mul"로 처리한다. 그러나 명령어 사이즈가 일정하고 단순화된 RISC 머신에서는 이러한 타입의 곱셈 처리에 어려움이 많기 때문에 이 타입을 처리하기 위해 FPU(floating point unit)에서 부동 소수점 연산같이 여러 사이클 동안 수행되어야 하는 복잡한 연산들을 처리한다.

(1) 상수에 의한 곱셈

곱셈의 승수가 되는 피연산자의 내용이 상수이면 컴파일시 피연산자의 내용을 이용하여 곱셈을 처리할 수 있다. 그래서 다음 과정을 통하여 구현할 수 있는 알고리듬을 제안한다. 상수에 의한 곱셈은 특수 명령어 (예, SPARC머신의 "mulscc", HP Precision 아키텍춰의 "Shift and Add" 명령)들이 없는 RISC머신을 대상으로 하고, 피연산자의 발생빈도에서 볼 때 확장된 곱셈보다 표준 곱셈이 자주 발생하므로 32비트 결과치를 갖는 표준 곱셈만을 가정한다. 그러므로 상수에 의한 곱셈에서는 표준 곱셈을 하기 위하여 32비트의 절반인 16비트로 표현할 수 있는 10,000 이하 수준의 상수에 의한 곱셈만 생각하기로 한다.

중간 결과들을 위해 필요로하는 템포러리 레지스터의 체인의 길이를 좌우하지만, 체인이 새로운 중간 결과를 저장하지 않도록 구성된다면, 중간 결과를 위해 템포러리 레지스터를 여러개 사용할 필요가 없다.

RISC 아키텍춰 상에서 상수에 의한 곱셈은 덧셈 체인으로 생각할 수 있다. Knuth는 'n'이란 숫자에 대한 덧셈 체인(chain)을 다음과 같은 정수의 시퀀스로 정의했다.[13]

$1 = C_0, C_2, C_2, \cdots\cdots, C_r = n$

이러한 정수의 시퀀스는 다음과 같은 룰에 의해서 생성된다.

$c_a = C_b + C_d$

그중에 b와 d는 [1, r] 사이의 모든 a에 대해 $d \leq b < a$이다.

Knuth의 정의를 기초로 하여 RISC 아키텍춰에서도 다음과 같은 유사한 체인을 구성할 수 있다.

$C_{-1} = 0, C_0 = 1, C_1, C_2, \cdots, C_r = n$

다음과 같은 룰들은 몇개의 명령들로 구현될 수 있다. 이 룰들을 이용하여 간단한 짝수나 홀수를 만들 수 있다.

$C_a = C_b + C_d$
$C_a = C_b - C_d$
$C_a = C_b \ll m$ ($\ll$:left shift, $\gg$:right shift)
$C_a = 2C_b + C_d$
$C_a = 4C_b + C_d$
$C_a = 8C_b + C_d$

이중에 d, b, a는 $d \leq b < a$이고, $m < 31$이다. C_{-1}은 대부분의 RISC 아키텍춰에서 하드웨어적으로 항상 0 값을 포함하는 글로발(global) 0 레지스터의 활용을 전제로 한다. 예를 들면, 숫자 12는 다음과 같은 체인으로 구성할 수 있다.

$C_0 = 1$
$C_1 = C_0 \ll 1$
$C_2 = C_1 \ll 1$
$C_3 = C_2 + C_1$
$C_4 = 2C_3 + C_{-1}$

그러나, 12에 의한 곱셈은 다음과 같이 3개의 스텝으로 끝날 수 있으므로 가능하면 최적의 시퀀스로 구성하려고 시도해야 한다.

r1 = s≪2 (s : 피승수)
r2 : r1≪1
r3 = r2 + r1

상수에 의한 모든 곱셈은 이 방법으로 구성할 수 있다.

(2) 알고리듬의 구성

위 방법으로 체인들을 탐색하기 위해 rule- based 프로그램으로 표현하였다. 이 프로그램을 사용하는 heuristic들을 간단히 표현하기 위하여, 우선적으로

일부 집합이 정의되어야 한다. 하나 혹은 두개의 명령스텝으로 숫자들을 구성할 수 있는 집합을 C_1= {2, 3, 4, 5, 8, 9, 16, …}이라 놓자. 그리고 셋 이상 다섯개의 명령스텝으로 숫자들을 구성할 수 있는 집합을 C_2= {6, 7, 10, 11, 12, 13, …}이라 놓는다. 정수의 시퀀스를 조사하기 위한 숫자들의 집합을 S라 하자(초기에 S={ }).

체인의 구성을 위해 명령어 스택(instruction stack)을 두고, 각 체인에서 사용되는 명령어들이 축적되도록 한다.

덧셈 체인의 생성 알고리듬은 다음과 같다.

```
while(S가 비어 있지 않으면){
  S에서 한 수를 선택하여 그 수를 K로 놓자;
  if(K∈C1)
    명령어를 스택에 집어 넣고 순환(recursion);
      〈accept rule을 시도〉
  else{
       for all c∈C1 {
            accept_distribution(c, k)
       }
       for all c∈C2 {
            accept_distribution(c, k)
       }
  }
}
명령어 스택의 인용을 인쇄;
```

여기서 accept_distribution(c, k)는 다음과 같이 확장할 수 있다.

```
f((k-c)%8==0){
      S=S U {(k-c)/8, c};
      k=8*((k-c)/8)+c를 명령어 스택에 집어
        넣고 순환;
      〈distribution rule을 시도〉
      〈odd_divisor rule을 시도〉
      〈complex rule을 시도〉
}
else if((k-c)%4==0){
      S=S U {(k-c)/4, c};
      k=4*((k-c)/4)+c를 집어 넣고 순환;
      〈distribution rule을 시도〉
      〈odd_divisor rule을 시도〉
      〈complex rule을 시도〉
}
else if((k-c)%2==0){
      S=S U {(k-c)/2, c};
      k=2*((k-c)/2)+c를 집어 넣고 순환;
      〈distribution rule을 시도〉
      〈odd_divisor rule을 시도〉
      〈complex rule을 시도〉
}
else{
      S=S U {k-c, c};
      k=(k-c)+c를 집어 넣고 순환;
      〈distribution rule을 시도〉
      〈odd_divisor rule을 시도〉
      〈complex rule을 시도〉
}
```

여기서 대부분의 많은 룰들을 다 보일 수는 없다. 그러나 distribution rule은 하나의 큰 숫자를 작은 두 숫자로 분할하려고 시도함으로써 명령어의 구성을 간략화 시킨다. 예를 들면, 다음과 같다.

```
if (k<1024){
     k1=k≫5;
     k2=k % 32;
     S=S U {k1, k2},
     c=k1≪5;
     k=c+k2를 집어 넣음;
     c=k1≪5를 집어 넣고 순환;
}
```

odd_divisor rule들은 제수 특성에 따라 하나의 숫자를 간략화 하여야 한다. 예를 들면, 다음과 같다.

```
switch(k % 9){
      case 0;/* try 9x=8(x)+(x) */
       k1=k/9;
       s=s U {k1};
       k=8*(k1)+k1를 집어 넣고 순환;
       break;
      case 1;/* try 9x+1=8(x)+(x+1) */
       k1=(k-1)/9;
       k2=k1+1;
       s=s U {k1, k2};
       k=8*(k1)+k2를 집어 넣고 순환;
       break;
      case 5;/* try 9x+32=8(x+4)+(x) */
       k1=(k-32)/9;
       k2=k1+4;
       s=s U {k1, k2};
       k=8*(k1)+k2를 집어 넣고 순환;
       break;
}
```

한편, 보다 처리가 까다로운 경우에 사용되는 다양한 종류의 룰들이 있다. 그러나, 그 예는 생략한다.

V. 실험 및 검토

RISC 머신 코드 생성기를 구현하기 위해 대상머신으로 SUN SPARC Station을 선택하였다. 정수 곱셈의 구현을 설명하기 전에 대상 머신의 정수 곱셈 방법을 살펴본다.

상용 머신인 SUN SPARC 머신에서는 정수 곱셈 처리를 위해 표현이 간결한 2의 멱수와 2의 배수 등의 작은 정수로 분할 구성하며, 분할 구성이 어려운 정수에 대하여는 SPARC 머신이 가지고 있는 mulscc(booth 알고리듬으로 구현된 하드웨어)명령어가 반복적으로 구성된 매크로 루틴 '.mul'을 호출하여 처리한다. 이러한 처리방법은 간결하게 구성된 정수의 경우는 빠르지만, 그렇지 못한 정수의 경우는 많은 명령 스텝을 수행하게 되기 때문에 전체적인 프로그램의 수행속도를 저하시키게 된다.

정수 곱셈 처리에서 상수에 의한 곱셈은 제안된 알고리듬을 통해 SPARC 머신에 적용하고, 변수에 의한 곱셈은 SPARC 머신의 "mulscc" 명령어로 구성된 매크로 루틴 '.mul'을 사용하여 처리한다.

그밖의 부동 소수점 연산은 일반적인 RISC 머신이 갖고 있는 FPU에 있는 명령어를 사용하여 처리한다.

1. 정수형 상수 곱셈 처리의 성능 평가

제안된 정수형 상수 곱셈처리 알고리듬과 SPARC 머신의 정수형 상수 곱셈처리의 성능을 비교 평가했다. 표 1에서는 10,000이하의 정수형 상수 곱셈처리 결과를 나타낸다. 10,000까지 비교한 이유는 위에서도 언급하였듯이 표준 곱셈을 대상으로 할 때 overflow를 고려해서 32비트의 절반이 채 안되는 곱셈만을 처리하기 위해서 이다. 표 1처럼 템포러리 레지스터를 1개 사용하였을 때, 본 알고리듬은 SPARC 머신의 정수형 상수 곱셈 처리 결과보다 약 20%의 성능이 향상되었다.

표 1. 10000까지의 정수형 상수 곱셈처리 수행 후의 평균 명령스텝

Table 1. Averge instruction step after processing integer constant multiplication to 10,000.

본 알고리듬	SPARC의 곱셈 처리부
13,8669스텝	17,2809스텝

2. 코드 생성 후 결과 코드

아래에 두 가지 예제 프로그램에 대한 제안된 코드 생성기의 출력 코드를 PCC의 코드와 SUN SPARC 머신의 코드 생성기의 출력 코드와 비교한다. 이 프로그램의 출력 코드는 대상 머신인 SPARC에서 에러없이 수행되었다.

다음은 bubble sort 프로그램의 코드 출력의 코드 출력을 나타낸다.

```
main()
{
        register  int top, i, temp, list[70];

        top = 70;
        while (top > 1) {
                top = top - 1;
                i = 0;
                while (i < top) {
                        if (list[i] > list[i+1]) {
                                temp = list[i];
                                list[i] = list[i+1];
                                list[i+1] = temp;
                        }
                        i = i + 1;
                }
        }
}
```

```
        .data
        .text
        .align  1
        .global _main
_main:
        .word   L12
        jbr     L14
L15:
        cvtbl   $70,r11
L16:
        cmpl    r11,$1
        jlseq   L17
        subl3   $1,r11,r0
        movl    r0,r11
        clrl    r10
L18:
        cmpl    r10,r11
        jgeq    L19
        addl3   $1,r10,r0
        cmpl    -280(fp)[r10],-280(fp)[r0]
        jleq    L20
        movl    -280(fp)[r10],r9
        addl3   $1,r10,r0
        movl    -280(fp)[r0],-280(fp)[r10]
        addl3   $1,r10,r0
        movl    r9,-280(fp)[r0]
L20:
        addl3   $1,r10,r0
        movl    r0,r10
        jbr     L18
L19:
        jbr     L16
L17:
        ret
        .set    L12,0x800
L14:
        movab   -280(sp),sp
        jbr     L15
        .data
```

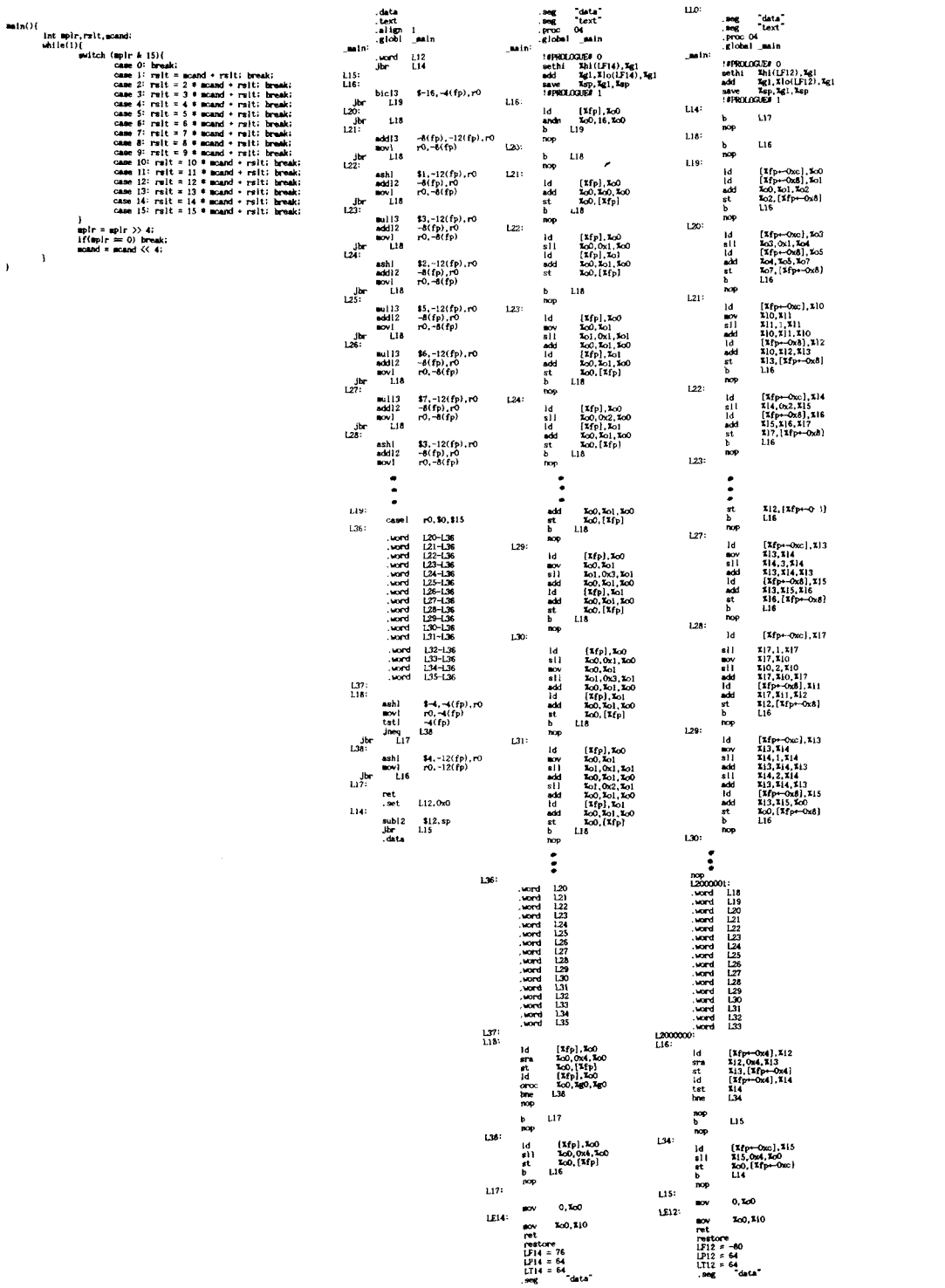
```
        .seg    "data"
        .seg    "text"
        .proc   04
        .global _main
_main:
!#PROLOGUE# 0
        sethi   %hi(LF14),%g1
        add     %g1,%lo(LF14),%g1
        save    %sp,%g1,%sp
!#PROLOGUE# 1
        mov     0x46,%i5
L16:
        cmp     %i5,0x1
        ble     L17
        nop
        sub     %i5,0x1,%o0
        mov     %o0,%i5
        or      %g0,%g0,%i4
L18:
        cmp     %i4,%i5
        bge     L19
        nop
        add     %i4,0x1,%o1
        sub     %fp,0x118,%o0
        sll     %o1,0x2,%o2
        ld      [%o0+%o2],%o3
        sub     %fp,0x118,%o4
        sll     %o4,0x2,%o5
        ld      [%o4+%o5],%o7
        cmp     %o7,%o3
        ble     L20
        nop
        sub     %fp,0x118,%o3
        sll     %i4,0x2,%o1
        ld      [%o0+%o1],%o2
        mov     %o2,%i3
        sub     %fp,0x118,%o3
        add     %i4,0x1,%o4
        sll     %o4,0x2,%o5
        ld      [%o3+%o5],%o7
        sub     %fp,0x118,%10
        sll     %i4,0x2,%11
        st      %o7,[%10+%11]
        sub     %fp,0x118,%12
        add     0x1,%i4,%o0
        sll     %i3,0x2,%i4
        st      %i5,[%12+%i4]
        add     0x1,%i4,%o0
L20:
        add     0x1,%i4,%o0
        mov     %o0,%i4
        b       L18
        nop
L19:
        b       L16
        nop
L17:
        mov     0,%o0
LE14:
        mov     %o0,%10
        ret
        restore
LF14 = -344
LP14 = 64
LT14 = 64
        .seg    "data"
```

```
L10:
        .seg    "data"
        .seg    "text"
        .proc   04
        .global _main
_main:
!#PROLOGUE# 0
        sethi   %hi(LF12),%g1
        add     %g1,%lo(LF12),%g1
        save    %sp,%g1,%sp
!#PROLOGUE# 1
        mov     0x46,%i5
L14:
        cmp     %i5,0x1
        ble     L15
        nop
        sub     %i5,0x1,%o0
        mov     %o0,%i5
        mov     0,%i4
L16:
        cmp     %i4,%i5
        bge     L17
        nop
        sub     %fp,0x118,%o0
        add     %i4,0x1,%o1
        sll     %o1,0x2,%o2
        ld      [%o0+%o2],%o3
        sub     %fp,0x118,%o4
        sll     %i4,0x2,%o5
        ld      [%o4+%o5],%o7
        cmp     %o7,%o3
        ble     L18
        nop
        sub     %fp,0x118,%o0
        sll     %i4,0x2,%o1
        ld      [%o0+%o1],%o2
        mov     %o2,%i3
        sub     %fp,0x118,%o3
        add     %i4,0x1,%o4
        sll     %o4,0x2,%o5
        ld      [%o3+%o5],%o7
        sub     %fp,0x118,%10
        sll     %i4,0x2,%11
        st      %o7,[%10+%11]
        sub     %fp,0x118,%12
        add     %i4,0x1,%i3
        sll     %i3,0x2,%i4
        st      %i3,[%12+%i4]
L18:
        add     %i4,0x1,%i5
        mov     %i5,%i4
        b       L16
        nop
L17:
        b       L14
        nop
L15:
        mov     0,%o0
LE12:
        mov     %o0,%10
        ret
        restore
LF12 = -344
LP12 = 64
LT12 = 64
        .seg    "data"
```

코드 나열 순서는 왼쪽이 PCC의 출력이고, 오른쪽아 SPARC의 출력이다. 그리고, 가운데 있는 출력이 본 논문에서 설계된 코드 생성기의 출력이다.

다음은 switch 문을 갖는 프로그램의 예를 나타낸다.

출력 코드 나열 순서는 앞의 예와 같다.



V. 결 론

본 논문에서는 RISC 머신을 위한 코드 생성기를 설계하고, 이러한 코드 생성기의 설계시 깊이 고려할 사항인 정수형 상수 곱셈처리 알고리듬을 제안하였다.

제안된 코드 생성기에서는 RISC 머신이 갖고 있지 않은 명령어들과 동일한 결과를 얻기 위한 처리 수행과 RISC 머신만이 갖는 실행시 메모리 관리방법을 고려하여 설계되었다. 또한 RISC 머신에 없는 명령어들 중 프로그램 수행시 내재되어 여러번 수행되기 때문에 프로그램 수행에 많은 영향을 미칠 수 있는 정수 곱셈 연산 명령의 효율적인 처리가 고려되어야 한다. 따라서 본 논문에서는 프로그램 수행시 내재된 연산들 중 특히 발생빈도가 큰 정수형 상수 곱셈을 보다 빠르게 처리할 수 있도록 덧셈 체인으로 명령어 시퀀스를 간결하게 구성하였다.

설계된 코드 생성기와 제안된 알고리듬은 상용 RISC 머신인 SUN SPARC station(UNIX BSD4.3) 상에서 C 언어로 구현하였다. 그리고, 코드 생성기는 사용자 프로그램 수행이 가능하였고, 본 알고리듬과 SPARC 머신의 정수형 처리부와 성능을 비교하여 약20%의 성능이 향상될 수 있음을 입증하였다.

앞으로의 연구 과제로는 TMDL(target machine discription language)을 사용하여 코드 생성기를 설계하는 것과 정수 곱셈과 나눗셈 처리중 나눗셈 처리에 대한 알고리듬 개발이 필요하다.

參 考 文 獻

[1] D.A. Patterson, "Reduced Instruction Set Computers," *Communications of the ACM*, vol. 15, no. 9, pp. 8-21, Jan. 1985.

[2] D.A. Patterson and C.H. Sequin, "A VLSI RISC," *IEEE Computer*, vol. 15, no. 9, pp. 8-21, Jan. 1985.

[3] M.G. Katevenis, "Reduced Instruction Set Computer Architectures for VLSI," *Ph.D. Thesis*, Univ. of California at Berkeley, Oct. 1983.

[4] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers Principles, Techniques, and Tools*, Addison-Wesley 1986.

[5] Cattell, R.G.G., "Automatic derivation of code generators from machine descriptions," *ACM Trans. Program. Lang. Syst.* 2, 2(Apr. 1980), 1973-190.

[6] Ganapathi, M., Fischer, C.N., and Hennessy, J.L. "Retargetable compiler code generation," *ACM Comput.* Survey, Dec. 1982.

[7] Glanville, R.S. "A machine-independent algorithm for code generation and its use in retargetable compilers," *Ph. D. dissertation*, Univ. of California, Berkeley, Dec. 1977.

[8] S.L. Graham, "Table-Driven Code Generation," *IEEE Comput, Aug. 1980.*

[9] S.C. Johnson, "A Portable compiler: Theory and Practice," Conf. Record 5th Ann. *ACM Symp. Principles of Programming Langnages*, Jan 1978.

[10] J.C. Gibson, "The Gibson mix," *Rep. TR 00.2043, IBM Syst.* Develop. Div., Poughkeepsie, NY, 1970.

[11] J.C. Huck, *"Comparative analysis of computer architecturs,'* Ph. D. dissertation, Stanford Univ., May 1983

[12] C.J. Neuhauser, *"Instruction stream monitoring of the PDP-11,"* Stanford Univ. Dep. Elec. Eng., Comput., Syst. Lab., Tech. Note 156, May 1979.

[13] D. Knuth, *The Art of Computer Programming*, vol. 2, Seminumerical Algorithms. Reading, MA: Addison-Wesley, pp. 444-446, 1981.

[14] D.J. Magerheimer, Liz Peters, K.W. Pettis, Dan Zuras, "Integer Multiplication and Division on the HP Precision Architecture," *IEEE Transactions on Computers*, vol. 37, no. 8, pp. 980-990, Aug. 1988.

[15] 김주형, 이형우, 조정삼, 홍인식, 박종득, 김은성, 임인칠, "RISC 머신의 정수형 상수 곱셈처리 알고리듬" 대한정보과학회 춘계 학술 발표논문집, 1990. 4.

[16] 박종득, 임인칠 "RISC 컴파일러의 기계 독립적 Global Optimizer 설계", 대한전자공학회 논문지, 1990. 3.

著 者 紹 介

朴 鍾 得 (正會員) 第27卷 第8號 參照
현재 한양대학교 전자공학과 박사과정

林 寅 七 (正會員) 第25卷 第8號 參照
현재 한양대학교 전자공학과 교수