論文92-29B-7-1

유연한 창문 구조를 갖는 레지스터 파일

(Flexible Register File with a Window Structure)

鄭 己 鉉*

(Gi Hyun Jung)

要 約

본 논문에서는 창문 구조를 갖는 레지스터 파일의 개요를 설명하며, 이 기술의 장점및 한계를 기술한다. 그 장·단점을 기초로 하여 큰 크기의 레지스터 파일 설계를 위한 하나의 독창적인 접근 방식이 기술되고, 분석되며 또한 기존의 방식들과 비교된다. 이 새로운 레지스터 파일 구조의 장·단점이 논의되며, 이 방식이 기존의 방식들보다 좋은 특성을 나타내는 조건들이 요약된다. 또한 설계시 유념해야 할 사항들이 정성적, 실험적으로 고찰되고, 그 결과가 본 논문의 결론부에서 정리된다.

Abstract

This paper gives an overview of register windowing structure and presents advantages and limitations. Based on these advantages and disadvantages, an original approach for the design of large register file is presented, analyzed and compared with existing approaches. The advantages and disadvantages of this new approach to register file design are discussed, and conditions under which it works better than the existing approaches are outlined. Design tradeoffs are examined in an analytic and empirical study, and the results of which are summarized in the conclusion of this paper.

I. 서 론

RISC시스템 설계자들은 단순화된 명령어 해석 로직을 사용함으로써 많은 칩면적을 절약할 수가 있다. 이렇게 얻어진 칩 면적은 시스템의 효율및 성능을 향상시키는데 사용될 수 있는 기능을 갖는 부분에 할당 될 수 있다. 하나의 가능한 방법으로 이 잉여 칩 영역을 부분적으로 중복되는 창문 구조를 가지는 큰 레지스터 파일을 실현하는데 사용하는 것이다. 그 레지스터 파일은 많은 경우에 시스템 효율을 크게 향상시키며, 특히 서브루틴의 호출이 여러번 겹쳐지는 고급 언어프로그램에서는 큰 이득을 가져온다.

부분적으로 중복되는 창문 구조를 가지는 큰 레지스터 파일 (a large register file with partially overlapped windows)을 갖는 시스템의 설계 예로서는 미국 버클리대의 RISC Ⅰ, RISC Ⅱ[1], SOAR[2] 그리고 SPUR[3]를 들 수있다. 마이애미 대학의 RISC 시스템[4]에서는 두개의 크기를 가지는 창문 구조 레지스터를 사용하고있다. 조지 메이슨 대의 MULTIRIS[5]는 실시간 다중 작업에 효율적인 RISC 시스템 구축을 위해 보다 큰 레지스터 파일을 사용한다. 한편 위스콘신대의 PIPE[6]는 단 두개의 레지스터 창문을 같은 목적으로 사용한다.

상업적으로는 Pyramid[7] 그리고 SPARC[3] 계열 시스템에서 이 창문 구조의 큰 레지스터 파일을 적용하고 있다. 이들 RISC 컴퓨터는 적은 레지스터 파일을 기초로 설계된 컴퓨터 보다 훨신 향상된 성능을 나타낸다. 실제로 창문 레지스터 파일 구조는 RISC (창문구조를 채택한) 시스템이 CISC 시스템보다 성능상 유리한 점을 갖게하는 하나의 큰 요소이다.

* 正會員, 亞洲大學校 電子工學科
(Dept. of Elec. Eng., Ajou Univ.)
接受日字 : 1991年 9月 20日

Cowell[8] 및 Hitchcock, Sprunt[9]는 창문 구조를 가진 레지스터 기술이 복잡한 명령어 컴퓨터(CISC)에서도 적용될 수 있음을 보여 주었다. 그들의 실험적 분석 결과에서는 창문 구조를 가진 레지스터 기술을 채택한 기존의 CISC 시스템에서도 역시 프로세서와 메모리간의 데이타 이동이 놀랄 만큼 감소되고 있음을 보여준다. Halbert와 Kessler[10]도 다중레지스터 구조의 성능을 측정하였다. 그들의 실험 결과는 디지털사의 VAX시스템도 창문 구조의 레지스터 파일을 적용함으로써 성능의 향상을 크게 가져올 수 있음을 보여 주고있다. 표 1은 창문 구조를 사용하고 있는 시스템의 예를 나타낸 것이다.

기존의 대부분의 창문 구조의 레지스터 파일을 가진 시스템에서는 고정된 (혹은 조금 수정된) 크기의 창문을 적용하고 있다. 가변의 크기를 가지는 창문 구조의 레지스터 파일은 복잡한 디코딩과 제어 구조의 필요성으로 인하여 극히 보기가 힘들다. 중복되는 창문 구조의 장점들을 실현키 위해 칩의 많은 부분이 레지스터 파일을 위해 사용된다. 따라서 명백하게 대부분의 다중 레지스터 파일 구조의 큰 단점 중의 하나는 레지스터 파일이 칩 면적을 과다하게 차지한다는 것이다. 실제로 그 영역들은 캐쉬 메모리, 소수점 환산을 위한 하드웨어 (floating point hardware)등의 다른 용도로도 사용될 수도 있다. 이와 같은 근본적인 레지스터 파일 구조의 단점을 보완하기위해, Katevenis[1]는 레지스터 파일 크기를 줄이기위해 여러가지 아이디어를 제공하였다. 하지만, 그는 대부분의 아이디어들이 실제적으로 실현되기 어렵다고 결론 짓고 있다. 본 논문에서는 다중의 중복되는 창문 구조의 레지스터 파일과 관련된 구조적인 장점을 손상하지 않고 칩 면적을 줄일수 있는 효과적인 방법이 제시되고 있다. 다중 레지스터 파일 구조의 설계시 유의사항을 보여줄 수 있는 정성적 분석과, 그 정성적 분석및 제안된 방법의 효용성을 뒷받침을 할 수 있는 실험적 결과가 역시 나열된다.

Ⅱ. 기존의 창문 구조 레지스터 파일 제어 전략들

고급 언어 프로그램에 대한 연구 결과에 따르면 고급 언어 수행중 가장 시간 소비를 많이하는 부분 중의 하나는 서브루틴의 연결에 있다.[12]. 관련된 두 가지 오버헤드가 큰 동작들은 (1) 서브루틴을 호출하는 프로시듀의 레지스터 저장, (2) 두 프로시듀간의 파라미터 이동등이 있다. 통상적인 고급언어 프로그램에서 가장 빈번히 사용되고 있는 명령어들은 표2에 나열되어 있다. 서브루틴의 연결과 관련된 오버 헤드를 줄이기위해 부분적으로 중복된 창문 구조의 레지스터 파일을 이용할 수가 있다. 이 전략은 여러 부류의 RISC 경향이 있는 시스템 설계에 적용되고 있다.

표 1. 창문 구조를 사용하고 있는 시스템의 예

Table 1. List of the systems with a register windon.

Processor	Manufacturer	Register Set	Windowing Structure
RISC Ⅱ	University of California, Berkerly	138 32-bit general purpose registers	Fixed 8 windows, 32 registers/window 6 high, 10 local, 6 low, and 10 global registers
SOAR	University of California, Berkerly	72 32-bit dual port general purpose registers	Fixed 7 windows, 32 registers/window 8 high, 8 low, 8 special, and 8 global registers
SPUR	University of California, Berkerly	138-bit general purpose registers 7 32-bit special, 15 87-bit floating point reg.	Fixed 8 windows, 32 registers/window 6 high, 10 local, and 6 low, and. 10 global registes
UM-RISC	University of Miami	64 16-bit dual port general purpose registers	Two size windows small window:6 high, 2 local, 6 low large window:6 high, 10 local, 6 low Max. window numbers:10 small windows 10 global registers
MULTIRIS	George Mason University	1024 32-bit general 16 32-bit interupt control registers	Fixed 64 windows. 16 registers/window non-overlapping windows
PIPE	University of Wisconsin	14 16-bit general purpose registers	Fixed 2 windows. 7 registers/window non-overlapping window
Pyramid 90X	Pyramid Technology	528 32-bit general purpose registers	Fixed 16 windows. 64 registers/window 16 high, 16 local, 16 low, and 16 global registers
SPARC	Sun Microsystem	120 32-bit general purpose registers	Fixed 7 windows. 24 registers/window 8 high, 8 local, 8 low, and 8 global registers
CYC 601	Cypress Semiconductor	136 32-bit general purpose registers	Fixed 8 windows, 32 registers/window 8 high, 8 local, 8 low, and 8 global registers
AM29000	Advanced Micro Devices, Inc.	192 32-bit general purpose 23 32-bit special 128 TLB registers	64 global registers 128 local registers 16 register banks. 16 registers/bank (4 unimplemented banks) overlapping and variable windows
C1260 (ACCEL)	Ceierity Computing	16 32-bit general purpose 1800 64-bit FPR's 16384 32-bit stack reg.	8 FPR banks. 15 frames /bank 16 registers/frame. 16 stack register bank. 32 frames/bank 32 registers/frame

중복된 창문 구조의 레지스터 파일은 다음과 같이 동작한다. 매번 프로시듀의 호출이 생길 때마다 큰 레지스터 파일로 부터 새로운 레지스터 창문이 호출된 프로시듀에 할당된다. 따라서 호출하는 프로시듀의 레지스터를 저장

할 필요가 없이 호출된 프로시듀를 위한 레지스터를 확보할 수 있으며, 아울러 이에 관련된 오버헤드를 줄일 수 있다. 특정 프로시듀를 위해 할당된 레지스터 창문은 해당 프로시듀에 의해서만 액세스가 가능하며, 이는 로컬 변수가 다른 프로시듀에의해 액세스되는 것을 구조적으로 보호해주는 방안이다.

호출된 프로시듀와 호출한 프로시듀의 레지스터 창문 사이에는 부분적으로 중복되는 부분이 있다. 프로시듀사이에서 파라미터 이동은 바로 이 부분을 통해서 일어난다. 여기에서 주의하고 고찰할 점은 주 메모리에 저장된 스택을 사용하는 스택 구조의 서브루틴의 연결 방법을 이용하는 종래의 시스템과 큰 차이가 있다는 점이다. 만약 이 부분이 없다면, 이동되는 파라미터들은 스택과 같은 일시적인 메모리에 저장 했다가 다시 참조해야하므로 오버 헤드를 필연적으로 동반한다. 글로벌 레지스터들은 시스템 및 전 프로시듀에서 공통적으로 사용되는 글로벌 변수를 위해 할당되며, 모든 프로시듀에 의해 액세스가 가능한 유일한 창문이다. 도해적인 설명의 예로서 그림 1에 버클리 대학의 RISC Ⅱ에서 사용하고 있는 레지스터 창문 구조를 도시했다.

표 2. 파스칼과 C프로그램에서의 통계
Tabel 2. Pascal and C program statistics.

Statement	Ocurrence (%)		Weighted Mem. Ref. (%)	
	PASCAL	C	PASCAL	C
Call /Return	15±1	12±5	44±4	45±19
Loops	5±0	3±1	33±2	26±5
Assign	45±5	38±15	14±2	15±6
If	29±8	43±17	7±2	13±5
With	5±5	-	1±0	-
Case	1±1	1±1	1±1	1±1
Goto	-	3±1	-	0±0

다른 대부분의 창문 구조 레지스터 파일과 같이 그림 1의 구조적인 단점은 창문 크기와 수가 정해져있다는 것이다. 특정 프로시듀의 로컬 변수및 파라미터의 수가 창문 크기보다 "로컬 변수 오버플로우"가 발생하게 된다. 이 경우 잉여 변수들은 지정된 스택이나 메모리의 일부에 저장되어져야 하고, 이들은 후에 필요에 의해 다시 레지스터 파일로 옮겨와야 한다. 따라서 이 잉여 변수들의 저장과 호출에 따른 오버헤드가 발생하며, 이는 창문 구조 레지스터 파일을 가지는 시스템의 성능 저하를 가져온다.

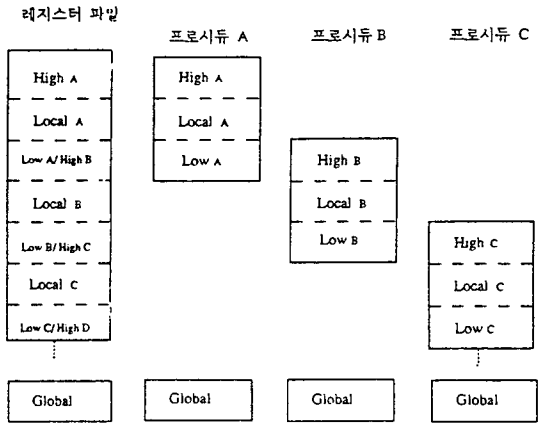

그림 1. RISC Ⅱ의 창문 구조레지스터 파일
Fig. 1. Register file of RISC Ⅱ.

또한 만약 호출되는 프로시듀의 수가 준비된 레지스터 창문 수 보다 많을 경우에는 문제가 발생된다. 그와 같은 경우는 "레지스터 창문 오버플로우"가 일어나 가장 오래된 레지스터 창문이 메모리에 저장되어져야만 한다. 이는 레지스터 창문 구조의 기본 이념에서 벗어나는 것이며 또한 원치않는 오버 헤드를 동반한다. 뿐만 아니라 메모리에 저장된 레지스터는 후에 복귀 동작시에 다시 레지스터 파일로 이동되어야만 하며, 이때 다시 오버 헤드를 일으킨다. 레지스터 창문 오버플로우와 언더플로우의 제어 구조는 참고 문헌[1]에 잘 기술되어있다.

레지스터 창문 오버플로우 및 언더플로우시에 몇 개의 레지스터 창문을 동시에 이동할 것인가 하는 것 또한 시스템 성능에 영향을 미친다. Tamir와 Sequin는 그들의 연구결과[15] 를토대로, 각 오버플로우 및 언더플로우 당 하나의 레지스터 창문을 이동함이 이상적이라 주장한다.

여러 연구결과[10,13,14]에서 보여지는 바와같이 거의 95% 이상의 프로시듀에서는 로컬 변수및 파라미터 수가 13개 이하이며, 70%이상의 경우 그 수는 8개 이하이다. 따라서 SPARC이나 RISC Ⅰ, Ⅱ와 같이 16개의 레지스터수의 반은 사용되지 않고 있다. 이는 각 창문당 16개의 레지스터로 8개의 레지스터 창문을 가질 경우 (10개의 글로벌 레지스터 포함) 54%의 레지스터만이 통상적으로 사용된다는 것이다. 이는 비싼 칩 면적을 소비할 뿐만 아니라, 오버플로우및 언더플로우시 프로세서와 메모리간의 불필요한 데이타 전송을 초래, 결과적으로 전체시스템의 성능저하를 가져온다. 왜냐하면, 레지스터 창문 이동시에 로컬 변수를 위해 쓰여지지 않은 레지스터들도 쓰여진 레지스터와 함께 이동 되어야 하기 때문이다.

또한 위의 연구결과에 따르면 99%이상의 경우 연속해

서 호출되는 프로시듀의 수는 8개 이하이다. 실제로 대부분의 창문 구조의 레지스터 파일을 가지는 RISC시스템에서는 8개의 레지스터 창문을 가진다. 예를들면, Pyramid 컴퓨터는 16개의 레지스터 창문을 가진다. 레지스터 창문 오버플로우및 언더플로우시에 동반되는 심한 오버헤드 때문에 레지스터 창문의 수를 8개 이하로 줄이는 것은 가능한 설계의 변수로는 고려하지 않는게 보통이다.

하지만 큰 레지스터 파일을 사용하는 데에는 몇가지 문제점있다. 첫번재로 가장 큰문제는 소비되는 칩 면적이다. 예를 들면 버클리대의RISC Ⅰ, Ⅱ에서 레지스터 파일과 그에 필요한 제어 회로를 구성하는 데 필요한 면적은 전체 칩 면적의 약 66%를 차지한다. 이 증가된 칩 면적에 따라, 적은 레지스터 파일을 쓸 때보다 상당량의 전력 소비의 증가는 필연적으로 동반된다. 또 하나의 간과할 수 없는 문제는 작업 전환(context switching)과 관련된다. 작업 전환(예를 들면, 페이지 오류나 인터럽트)의 필요성이 발생하였을 경우, 후에 작업복귀를 대비하여 현재의 시스템상태를 저장하여야 한다. 이때 큰레지스터 파일을 가지고 있는 시스템은 적은 파일을 가지고있는 시스템보다 훨씬 많은 양의 메모리와 레지스터파일간 데이타 전송이 요구된다. 예를 들어 인텔 80386에서 자동적으로 이루어지는 작업 전환과 버클리대의 RISC Ⅱ의 인터럽트 서비스 루틴의 실행을 통하여 이루어지는 레지스터 파일의 저장과 복원의 시간차이를 비교해보라. 큰 레지스터 파일을 가지고 있는 시스템이 효과적이고 가격면에서 유리해지기 위해서는 이상의 문제점들이 선결되어져야 한다.

창문 구조를 가지는 레지스터 파일은 크게 두가지로 분류 된다. 현재 실용화되어 있는 대부분의 RISC 시스템에서는 창문의 크기가 변할수 없는 구조를 채택하고 있다. 구체적인 예는 그림 1에서 보여진다. 여기에서는 창문의 크기가 하드웨어로 먼저 정해진다. 프로시듀의 호출(복귀) 명령이 수행될때, 먼저 현재의 창문 위치를 가르키는 지시자(currnrent window pointer)가 다음(복귀시는 이전) 창문 레지스터의 시작하는 번지를 가르키게 된다. 이 경우 실제로 사용 되는 로컬 변수의 수와는 관계없이 하드웨어에서 정해진 수만큼 항상 일정한 수의 레지스터가 각 창문에 주어진다. 이 구조는 명백한 하드웨어 자원의 손실에도 불구하고 보편적으로 사용되고 있다. 그 이유는 실현하기가 쉽고 디코딩과 제어 구조를 간단히 할 수가 있기 때문이다. 하지만 하드웨어 자원의 손실과 함께 이 구조의 가장 큰 단점은 창문의 오버플로우와 언더플로우시 사용되지 않는 레지스터들이 저장/복귀 된다는 사실이다. 이는 시스템의 가격 효율및 창문 레지스터의 유용성을 크게 떨어드리며 결과적으로는 그 시스템의 경쟁력을 잃게 한다.

창문의 크기가 변할 수 없는 구조의 한 대안으로서 변하는 크기의 창문을 가지는 구조가 있다. 여기에서는 각 창문의 크기는 호출된 프로시듀에서 실제로 사용되는 변수의 수에 따른다. 따라서 사용되지 않는 레지스터를 없애며, 궁극적으로 잉여 변수(창문 크기보다 변수가 많은 경우)에 의한 로컬 변수 오버플로우를 없앨 수 있다. 이 구조를 사용하면, 시스템 전체적인 효율을 크게 떨어뜨리지 않고 각 창문의 레지스터 수를 효과적으로 줄일 수 있다. 또한 잉여 변수에의한 오버플로우로 발생되는 불필요한 파라미터를 없앨수 있다. 하지만 이 구조를 실현키 위해서는 복잡한 제어 구조가 필요하고 실현된 복잡한 제어 구조에서 발생되는 오버헤드 때문에 변하는 크기의 창문을 가지는 레지스터는 찾기 힘들다.

Ⅲ. 유연한 창문 구조를 가진 레지스터 파일 제어 전략

이상적으로 적은 크기의 창문 구조 레지스터 파일이 큰 레지스터 파일이 가지는 이점을 잃지 않는 것이 좋은 레지스터파일 설계의 기본 요건이다. 따라서 레지스터 파일 설계의 중요한 요소 중의 하나는 적절한 창문 크기를 정하는 것이다. 만약 너무 크게되면 하나의 창문이 모든 로컬 변수를 포함할 수 있어 로컬 변수에의한 오버플로우를 없앨 수 있다는 잇점이 있으나, 사용되지 않는 레지스터가 발생될 가능성이 높아지게되고 따라서 레지스터 메모리간의 데이타 전송을 많이 일어키며 또한 칩면적의 손실을 가져온다. 만약 너무 작게되면 창문의 부족에 따른 오버플로우는 줄어드나, 로컬 변수에 의한 오버플로우의 확율이 매우 높게된다. 두 경우의 장점을 살리는 하나의 방법은 비교적 적은 수의 레지스터를 쓰면서 필요에 따라 하나 이상의 레지스터 창문을 특정의 프로시듀에 할애하는 방법이다. (기존의 시스템에서는 하나의 창문만이 각 프로시듀에 주어진다.) 고급 언어 프로그램의 분석에서 얻어진 통계(고급 언어 프로시듀의 약 70%는 8개 이하의 로컬 변수를 갖는다.)에 따르면 가장 적당한 크기의 창문 크기는 8개의 레지스터를 갖는 것이다. 이상의 요소를 감안하여 여기에 새로운 창문 구조인 "유연한 창문 구조레지스터 파일"을 제안한다.

유연한 창문 구조 레지스터 파일의 한 예를 그림 2에서 볼수 있다. 이 구조에서는 만약 어떤 프로시듀의 로컬 변수가 창문의 크기(여기서는 8)를 초과하면 프로세서는 하나 이상의 레지스터 창문을 그 프로시듀에 할당한다. 그리고 주어진 창문들은 서로 연결되어 하나의 큰 창문을 이룬다. 적은 수의 로컬 변수를 갖는 프로그램에서 이 전략은 기존의 구조인 불변의 창문 구조보다 훨씬 효과적 제어 방식이 될 수 있다.

이 유연한 창문 구조를 갖는 레지스터 파일은 다음과 같이 동작한다. 프로시듀의 호출이 시작되면, 프로세서는

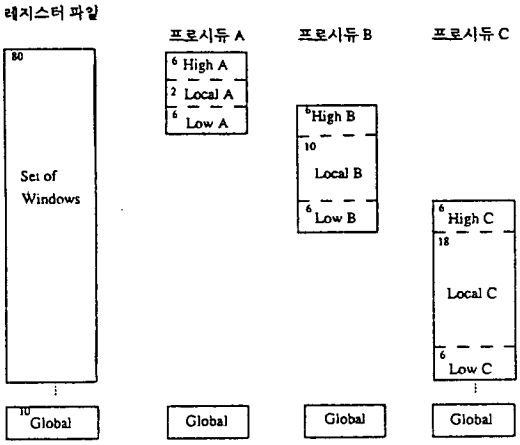

그림 2. 유연한 창문 구조를 갖는 레지스터 파일 구조
Fig. 2. Flexible register window organization.

먼저 현 창문 위치자의 내용을 현재 창문의 아래 (Low) 창문의 맨 상위 레지스터에 위치시킨다. 이는 후에 호출된 프로시듀로 부터 복귀시 현 창문 위치자를 컴파일러의 도움없이 복원하기 위한 것이다. 또한 이 정보는 창문 오버플로우나 언더 플로우시에도 유용하게 사용된다. (후에 설명)

다음으로, 호출되는 프로시듀에 필요로하는 인수를 호출하는 프로시듀가 점유하고 있는 현재 창문의 아래 창문에 두번째 레지스터부터 위치시킨다(첫번째 레지스터는 창문 위치가 저장). 만약 인수의 수가 아래 창문의 크기보다 크면, 잉여 인수는 캐쉬 메모리나 주 메모리에 저장되어져야하며 이들은 호출된 프로시듀가 수행을 시작할 때 그 프로시듀에 할당된 창문에 옮겨진다.

인수의 저장이 끝나면 프로세서는 현 창문 위치자를 정수{(A+L+1)/S)}의 함수로서 이동시킨다. 여기서 A는 인수의 수, L은 로컬 변수의 수, S(정해진 크기의 수 ; 여기서는 8)는 창문의 크기이다. 정수{ } 함수속의 1은 현 창문 위치자의 내용을 저장하기 위한 하나의 레지스터이다. 예를 들어, 7개의 로컬 변수와 5개의 인수를 필요로하는 프로시듀가 호출되었을시는 정수{(5+7+1/8)}+1의 함수의 식으로부터 얻어지는 값 즉, 2개의 창문의 크기(여기서는 16개의 레지스터)만큼 현 창문 위치자를 이동시킨다. 즉, 하나의 논리 창문의 크기는 어떤 프로시듀에 필요한 모든 로컬 변수들을 수용할 수 있는 크기만큼의 창문 수에다 잉여의 변수가 들어갈 수 있는 창문이 더 필요하게 되는 것이다.

프로시듀가 복귀를 시작하면 먼저 프로세서는 복귀시켜야할 정보, 즉 호출된 프로시듀에서 계산된 결과값을 현 창문의 위(High) 창문의 두번째 레지스터부터 저장한다. 이 위 창문은 복귀될 프로시듀의 창문의 아래 창문에 해당된다. 결과 값의 저장이 끝나면 프로세서는 위 창문의 맨 상위 레지스터에 저장되어있는 창문 위치지시자의 값으로 현 창문 위치자의 값을 대치하여 복귀된 프로시듀에 필요한 창문을 복원하며, 아래창문에 저장된 하위 프로시듀에서 계산된 값을 참조하여 프로시듀 수행을 계속한다.

프로시듀의 호출이 계속되어 어느 시점에서는 호출된 프로시듀에 할당 할 수 있는 레지스터가 더 이상 존재하지 않을 수 있다. 이 경우를 창문 오버플로우라 한다. 이 창문 오버플로우는 통상 창문 언더플로우를 동반한다. 창문 오버플로우 발생후에 호출된 프로시듀가 복귀를 거듭하면 어느시점에서는 복귀할 프로시듀에 필요한 변수들이 해당 창문에 존재하지 않게되며 이를 창문 언더플로우라 한다. 오버플로우나 언더플로우의 제어는 창문의 크기가 정해져 있는 레지스터 파일 구조에서와 매우 흡사하다[1]. 하나 틀린 점은 만약 증가된 현 창문 위치자가 저장된 창문의 위치자(SWP ; saved window pointer, 오버플로우 때문에 발생된 메모리에 저장된 창문들 중 가장 나중에 저장된 창문을 가르킴)보다 크거나 같으면 창문 오버플로우 루틴이 작동한다. 반면 감소된 현 창문 위치자가 저장된 창문의 위치자보다 같거나 적으면 창문 언더플로우 루틴이 작동한다는 것이다.

하지만 보통의 창문구조 레지스터 파일에서와 같이 여기서도 컴파일러의 역할이 매우 중요함을 알 수가 있다. 이 창문 구조를 가진 레지스터 파일을 가지는 시스템에서 컴파일러는 필요로 하는 로컬 변수의 수를 예측하는 능력이 있어야 한다. 왜냐하면 이 예측을 기초로 해서 프로시듀 연결 명령에 의해 적당한 수의 레지스터가 새로이 발생된 프로시듀에 할당되기 때문이다. 좋은 예측이 되지않으면 잉여 변수에 의한 오버플로우가 발생할 가능성이 높게된다.

고급 언어 프로그램의 99%가 프로시듀의 연속 호출이 8개보다 적으면, 글로벌 변수의 수가 10개 이하인 점에 착안하여, 여기에 10개의 창문(2개는 로컬 변수의 수가 큰 프로시듀를 위한 여분)및 10개의 글로벌 레지스터를 가진 창문 구조의 유연한 레지스터 파일을 제안한다. 버클리 대의 RISC Ⅱ및 SPARC(138개의 레지스터)에 비교하여 제안된 구조(90개의 레지스터)는 시스템 효율을 감소시키지 않고도 약 35%의 레지스터를 줄일 수가 있다. 이와 같은 칩 면적의 절약은 공간의 제약을 받는 감룸알사나이드와 같은 프로세서에서 매우 유용하게 이용될수 있다.

이 구조의 제어 회로는 기존의 구조들에서 보다 복잡하다. 하지만 완전히 가변적인 구조에서 보다는 훨씬 간단히 구현될 수가 있으며, 이 다소 복잡한 제어 구조의 불리함도 칩 면적의 절약 및 시스템 효율 향상으로 보상받을

수 있음을 다음의 분석에서 알수가 있다.

Ⅳ. 정성적 분석

이 분석을 위해 몇 가지의 가정을 해둔다. 첫째, 모든 명령어는 하나의 주기 내에서 수행될 수 있다. 단, 32 비트 프로세서에서 하나의 데이타어를 쓰고 읽는 데에는 두 주기가 필요하다. 이는 일반적인 RISC 프로세서에서와 같다. 둘째, 평균적인 오버플로우와 언더플로우의 수는 같다. 전체적인 레지스터 어드레싱 시간은 레지스터 수의 제곱근에 비례한다[16]는 것도 역시 고려하였다.

창문 구조의 레지스터 파일에 의해서 발생하는 수행시간중의 오버헤드는 아래와 같은 요소가 있다[14]

(i) T1 : 레지스터 파일과 외부 메모리 사이의 실제적인 데이타 전송에 소요되는 시간.

(ii) T2 : 잉여 변수에 의한 오버플로우 동안 메모리에 잉여 변수를 옮기는 데에 필요한 시간.

(iii) T3 : 오버플로우와 언더플로우를 감지하고 현 창문의 지시자를 정정하는 데에 필요한 시간.

(iv) T4 : 인터럽트를 수행하는 데에 필요한 시간.

(v) T5 : 창문의 오버플로우와 언더플로우에 의해 발생되는 데이타의 전송에 필요한 시간.

여기에서는 사소한 요소에 의한 오버헤드는 고려하지 않았다. 그러므로 제시하는 분석은 개략적인 것이다.

각 오버헤드에 소요되는 시간은 소요되는 주기의 수(n_i)와 해당되는 주기시간(t_c)의 곱이다. 실행 시간에 소요되는 총 시간은 T1에서 T5까지의 합이다. 해당 주기수는 다음의 식에서 구해진다[4].

(1) n_1 : 레지스터 파일과 외부 메모리 사이의 실제적인 데이타 전송에 소요되는 주기.

$$n_1=2\sum_{i=0}^{c} min(s,x_i) : \text{불변의 창문 구조}$$

$$n_1=2\sum_{i=0}^{c} min(w_s, x_i) : \text{유연한 창문 구조}$$

여기에서 x_i는 로컬변수의 수, c는 *call*명령의 수, s는 불변의 창문 크기, 그리고 w_s는 가변의 창문 크기이다.

(2) n_2 : 잉여 변수에 의한 오버플로우 동안 메모리에 잉여 변수를 옮기는 데에 소요되는 주기.

$$n_2=2\sum_{i=0}^{c} f_i\, pos(x_i-s) \quad : \text{불변의 창문 구조}$$

$n_2=0$: 유연한 창문 구조

여기에서 f_i는 로컬 변수와 파라미터들의 사용 빈도이며, pos는 변수의 값이 양일때에만 정의하는 함수이다. (변수의 값이 음일 때는 0)

(3) n_3 : 오버플로우와 언더플로우를 감지하고, 현 창문의 지시자를 정정하는데에 소요되는 주기.

$n_3=2c$: 불변의 창문 구조

$n_3=4c$: 유연한 창문 구조

(4) n_4 : 인터럽트를 수행하는 데에 소요되는 주기.

$n_4=30\ O_v$: 불변의 창문 구조

$n_4=75\ O_v$: 유연한 창문 구조

여기에서 O_v는 오버플로우의 수.

(5) n_5 : 창문의 오버프로우와 언더 플로우에 의해 발생되는 데이타의 전송에 소요되는 주기.

$n_5=4\ O_{vs}$: 불변의 창문 구조

$n_5=4\ O_v w_s(x_i)$: 유연한 창문 구조

여기에서 $w_s(x_i)$는 로컬 변수에따라 변하는 유연한 창문의 크기. t_c는 레지스터 파일크기의 제곱근에 비례한다.[16] 여기에서 하나 유의할점은 제안된 창문구조의 오버헤드와 불변의 창문을 가진 구조의 오버헤드를 비교하려 하는 것이다. 따라서 불변 창문 구조의 t_c는 '1'이다.

위의 식들에 기초를 해서 두 구조의 실행시간에 소요되는 오버헤드를 측정하였다. 그림 3에 그 비교치를 도시하였다.(wn은 창문의 수이다.) 실행 시간비 1 이하는 제안 구조의 오버헤드가 적음을 나타낸다.

이 그림에서 볼 수 있듯이, 창문의 수가 같을 경우, 제안된 구조의 레지스터 파일은 창문의 크기(불변의 구조)가 10 이하일 때에, 불변구조의 창문 보다 적은 오버헤드를 가진다는 것을 보여준다. 이는 고급 언어의 경우 75% 이상의 프로그램이 8개 이하의 로컬 변수를 가진다는 것을 감안할 때, 약 80%이상의 경우는 제안된 구조가 우수함을 나타낸다. 또한, 창문의 수가 증가할 수록 적은 오버헤드 비를 가진다는 것을 보여준다. 이것은 큰 레지스터 파일에서 가변 창문 구조가 더욱 유용하게 사용되어 짐을 나타내는 것이다. 창문의 크기가 8개를 초과하는 경우 거의 모든 실제의 범주에 제안된 구조는 불변의 구조보다 적은 오버헤드를 나타낸다. 로컬 변수의 사용 빈도가 증가할경우 오버헤드 비는 급격히 감소한다. 이것은 잉여 변수에 의한 오버플로우의 경우 불변의 구조에서는 데이타의 전송이 발생하지만 제안된 구조에서는 데이타 이동이 없기 때문이다.

얻어진 결과는 제안된 구조가 불변의 창문 구조보다 뛰어난 두가지점을 가리킨다. 첫째는 레지스터 파일크기의 감소이며, 둘째는 실행 시간상의 오버헤드 감소이다. 약간 복잡한 제어 구조를 가지나 이는 레지스터 파일의 크기 감소로 충분히 보상된다.

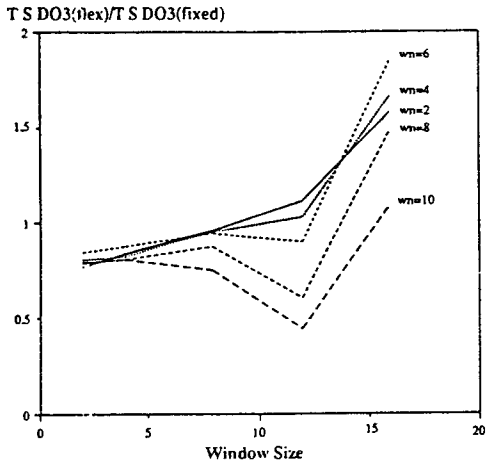

그림 3. 유연한 창문 구조에서의 실행 시간비
Fig. 3. Execution time of the flexible window scheme.

V. 실험적 분석

앞에서 설명된 두 창문 구조의 상대적인 효율 비교를 위해, 그림 1과 그림2의 두 구조에 대해 실험이 행해졌다. 실험은 미국의 PURDUE 대학의 컴퓨터 시스템에 조성된 환경에서 이루어졌으며, 사용된 소프트웨어는 ZYCAD사의 ENDOT 패키지이다.

수행시간의 측정을 위해서 Fibonacci 프로그램(N=19)이 벤치마크로 사용되었다. 사용된 Fibonacci 프로그램은 여러가지의 로컬 파라미터를 입력으로 받아들일 수 있게 수정되어 각 경우에서 원하는 부류의 수행시간을 측정할 수 있게 하였다. 유연한 창문 구조의 창문 연결 효과와 불변창문 구조의 오버헤드를 고려하여, 로컬 변수의 수는 8의 배수로 정하였다. 8은 제안된 창문 구조 레지스터의 연결되기 전의 창문크기이며, 16은 선택된 창문 구조의 불변 레지스터(예, SPARC, RISC Ⅱ)창문의 크기이다. 로컬 변수의 사용 빈도 및 호출/복귀 명령이 발생되는 간격도 또한 고려되었다. 프로시듀간의 연결 파라미터의 수는 하나에서 다섯개 이상까지 변한다. 이는 두 창문 구조의 부분적으로 겹쳐지는 레지스터의 수와 관련되어, 시스템 효율에 크게 영향을 미친다. 따라서 파라미터의 효과 또한 실행시간비에 포함되어 진다.

변형된 Fibonacci 프로그램이 짧긴하지만, 시스템 전체의 효율에 미치는 영향을 측정하는데에 충분할 만큼의 호출/복귀 명령이 포함되어 있으며, 프로그램에서 초기화에 필요한 시간의 효과를 배제할 수 있을 만큼의 충분한 수행시간을 가진다.

시뮬레이션에 사용하는 변수로서는 인수의 수, 로컬 변수의 수및 창문의 수를 사용하였다. 인수의 수및 로컬 변수의 수는 하나의 논리 창문의 수를 결정하는데에 중요한 변수이다. 즉, 두수의 합의 크기에 따라 기본적인 창문(8개의 레지스터)을 몇개나 연결 시키느냐가 결정되며, 이는 특정 프로시듀에 할당되는 창문의 크기를 결정하여 전체적인 성능에 영향을 준다. 또한 창문의 수는 우선 전체 레지스터 파일의 크기를 좌우하여 칩 면적에 영향을 끼치고, 또한 전체 레지스터의 어드레싱 시간에도 영향을 주어 결국 시스템 성능에 영향을 주게된다.

두 창문의 실행시간의 비를 구하기위해 두 구조가 모두 실행되어졌다. 또한 그 비를 균일화 하기위해 유연한 창문에서의 실행시간이 불변 창문의 실행 시간으로 나뉘어졌다. 따라서 1 이하의 비에서는 유연한 창문이 불변 창문보다 우수한 효율을 가짐을 의미한다. 표 3에 여러가지 창문의 수, 파라미터의 수 그리고 로컬 변수의 수에서의 두 창문 간의 실행시간의 비를 나타내었다.

그림 4와 5는 시뮬레이션 결과를 예시한 것이다. 로컬 변수의 영향을 그림 4에 나타내었다(L은 로컬 변수의 수)같은 수의 창문에서 로컬 변수가 9개 이하인 프로그램(Fibonacci 프로그램)을 수행하는 대부분의 경우, 유연한 구조의 창문이 불변 구조의 창문 구조보다 빠르다는 것을 알 수가 있다. 앞에서도 얘기했듯이 약 70%의 고급 언어 프로그램은 로컬 변수는 8개 이하이다. 따라서, 그림 4는 유연한 구조의 창문이 불변구조의 창문 구조보다 약 70%이상의 경우 좋은 효율을 가짐을 나타낸다. 그러나 창문의 수가 12개 이상인 경우는 유연한 구조의 창문의

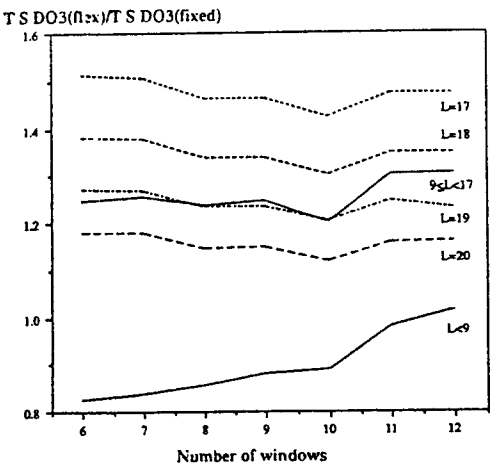

그림 4. 여러가지 수의 로컬 변수에서 유연한 구조와 불변 구조의 수행 시간비
Fig. 4. Execution time ratio for the flexible window scheme vs. the fixed window scheme.

표 3. 유연한 창문대 불변 창문 구조에서의 오버 헤드비

Table 3. Overhead ratio for the flexible vs. the fixed window schemes.

variabls	pass. windows parameters	Execution Time Ratio 6	7	8	9	10	11	12
V≤8	5	0.824	0.838	0.855	0.879	0.911	0.981	1.015
	6	0.809	0.828	0.848	0.874	0.907	0.978	1.011
	7	0.816	0.838	0.862	0.890	0.925	0.997	1.031
	8	0.922	0.847	0.874	0.904	0.939	1.013	1.047
8<V≤16	5	1.245	1.255	1.235	1.246	1.199	1.300	1.304
	6	1.181	1.195	1.183	1.196	1.175	1.255	1.262
	7	1.143	1.159	1.151	1.166	1.133	1.228	1.236
	8	1.110	1.128	1.123	1.140	1.111	1.205	1.214
	9	1.081	1.101	1.099	1.117	1.092	1.184	1.195
	10	1.056	1.077	1.077	1.096	1.076	1.166	1.178
V=17	5	1.512	1.507	1.461	1.463	1.421	1.471	1.471
	6	1.436	1.436	1.399	1.405	1.371	1.423	1.425
	7	1.385	1.388	1.357	1.364	1.355	1.387	1.392
	8	1.341	1.346	1.319	1.328	1.303	1.357	1.392
	9	1.302	1.309	1.286	1.297	1.275	1.329	1.336
	10	1.267	1.276	1.257	1.269	1.250	1.305	1.313
V=18	5	1.380	1.376	1.336	1.338	1.301	1.347	1.348
	6	1.322	1.323	1.290	1.296	1.265	1.314	1.317
	7	1.283	1.286	1.258	1.265	1.238	1.288	1.292
	8	1.247	1.253	1.229	1.237	1.215	1.265	1.271
	9	1.216	1.223	1.203	1.213	1.193	1.244	1.251
	10	1.187	1.197	1.179	1.191	1.174	1.226	1.234
V=19	5	1.271	1.269	1.232	1.235	1.202	1.245	1.232
	6	1.227	1.228	1.198	1.204	1.177	1.222	1.213
	7	1.195	1.199	1.173	1.180	1.157	1.203	1.196
	8	1.167	1.173	1.151	1.159	1.139	1.186	1.181
	9	1.142	1.129	1.130	1.140	1.122	1.171	1.167
	10	1.118	1.128	1.112	1.123	1.108	1.157	1.155
V=20	5	1.145	1.178	1.145	1.148	1.118	1.157	1.167
	6	1.120	1.147	1.120	1.126	1.098	1.144	1.147
	7	1.097	1.124	1.101	1.108	1.083	1.130	1.135
	8	1.077	1.103	1.083	1.092	1.070	1.118	1.124
	9	1.077	1.084	1.067	1.077	1.058	1.107	1.114
	10	1.058	1.067	1.053	1.064	1.047	1.097	1.104

효율이 떨어짐을 알 수가 있다. 이것은 왜냐하면 유연한 구조의 창문에서 증가된 레지스터에 의한 주기 시간이 증가하는 효과가 불변 구조에서보다 훨씬 크기 때문이다.

만약 로컬 변수의 수가 8개를 넘으면(이 경우 유연한 창문은 2개 이상의 창문을 연결할 필요가 있다), 창문을 연결시키는 오버헤드가 훨씬 심각해진다. 하지만 불변 창문은 여전히 하나의 창문만이 필요하며 유연한 창문에서와 같은 오버헤드는 없다. 따라서 수행시간 비는 증가하기 시작한다. 로컬 변수의 수가 16개를 넘어서면 문제는 더욱 심각해진다. 왜냐하면 이 경우 유연한 창문은 3개 이상의 창문을 연결해야 하나의 창문역할을 할 수가 있기 때문이다. 따라서 수행 시간 비는 1보다 크다.

로컬 변수의 수가 18개를 넘으면 수행 시간 비가 다시 감소하기 시작한다. 이것은 불변 구조의 창문이 가지는 잉여 변수때문에 일어나는 오버플로우의 오버헤드가 유연한 구조에서 일어나는 창문 연결의 오버헤드보다 크기 때문이다.

그림 4에서 볼 수 있듯이 고급 언어의 경우, 유연한 구조의 창문은 창문의 수가 10개일 때에 그 최고의 효과를 나타낸다. 불변 구조의 창문(e.g., U.C. Berkeley RISC Ⅱ)과 비교할 때에 10개의 창문을 갖는 제안된 유연한 구조의 창문은 전체 시스템의 효율을 감소시키지 않는 채, 레지스터 파일 크기를 약 35% 줄일 수가 있다. 따라서 다음의 실험에서는 10개의 창문을 갖는 유연한 구조의 창문이 활용 된다.

그림 5는 프로시듀간의 통신을 위해 사용되는 파라미터의 수의 변화에 따른 효과를 나타낸다(P는 파라미터의 수). 로컬 변수가 10개 이하인 경우에서는 파라미터의 수가 전체 수행 시간에 크게 영향을 미치지 않음을 볼 수가 있다. 이 경우 파라미터의 수와 관계없이 수행 시간비가 일 이하이다. 즉, 유연한 창문이 불변 창문보다 좋은효과를 가진다.

로컬 변수의 수가 증가할 수록 파라미터의 수의 증가의 효과는 줄어든다. 이것은 불변 창문에서 잉여 파라미터에 의한 오버플로우의 증가가 유연한 창문에서의 창문 연결에 의한 오버헤드의 증가 보다 적기 때문이다.

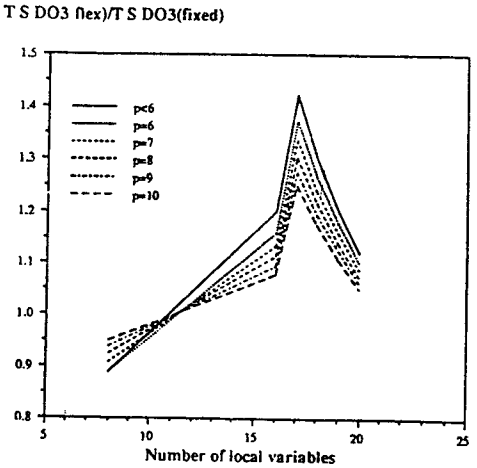

그림 5. 여러가지 수의 파라미터에서 유연한 구조와 불변 구조의 수행 시간비

Fig. 5. Execution time ratio for the flexible vs. the fixed window scheme for various number of parameters.

Ⅵ. 결 론

본 논문에서는 종래의 불변 구조의 창문을 가진 레지스터 파일의 한 가지 대안으로 유연한 구조의 창문 형태가 제안되었고 정성적, 실험적으로 고찰되었다. 분석된 자료들은 제안된 구조가 수행 시간 단축 및 레지스터가 차지하는 면적을 줄일 수 있음을 보여준다. 특히 많은 고급언어 프로그램이 8개이하의 로컬 변수를 가진다는 점을 고려할 때 제안된 구조가 보여주는 적은 수의 변수를 가지는 프로그램에서의 성능 향상은 많은 경우의 고급언어 프로그램 실행시간을 단축할 수 있음을 보여주었다. 또한 프로시듀의 호출과 복귀가 빈번한 프로그램에서는 기존의 구조보다 훨씬 효과적인 구조이다. 레지스터의 크기를 줄이는 것은 복수의 작업을 하는 시스템에서 작업간의 이동에 소요되는 오베헤드를 줄일 수 있다는 또 하나의 큰효과를 가져온다.[14] 제안된 기술은 보다 적은 칩면적을 사용하여 소기의 목적을 달성할 수가 있으며, 칩면적의 압박을 받는 창문 구조를 사용하려는 갈륨 알사나이드와 같은 RISC 형 프로세서들에서는 그 유용도가 크다.

參 考 文 獻

[1] M. Katevenis, *Reduced Instruction Set Computer Architectures for VLSI,* Ph. D. Thesis U. C. Berkeley(Oct. 1983)

[2] D. Ungar, P. Foley, D. samples, and D. Patterson, "Architecture of SOAR: smalltalk on a RISC," *Proc. of 11th Symposium on Computer Architecture,* ACM Press, pp. 188-197(June 1984).

[3] C.E. Gimarc and V. M. Milutinovic, "A survey of RISC processors and computer of the Mid-1980s," *IEEE Computer,* vol. 20, pp. 59-68(Sept. 1987).

[4] B. Furht, "A RISC architecture with two-size overlapping register windows," *IEEE Micro.,* vol. 8, pp. 67-80(Apr. 1988).

[5] D. Quamnen, D. Miller, and D. Tabak, "Register window management for a real-time multitasking, RISC," *IEEE Computer* pp. 135-142 (Jan. 1989).

[6] J. Smith, A. Pleszkun, R. Katz, and J. Goodman, "PIPE: A high performance VLSI architecture," *IEEE Computer,* pp. 131-138(July 1986).

[7] R. Kelly and R. Clark, "Applying RISC theory to a large computer," *Computer Design,* vol. 22, pp. 397-301(Sept. 1985).

[8] R. Colwell et. al., "Computers complexity, and controversy," *IEEE Computer,* vol. 18, pp. 8-19(Sept. 1985).

[9] C. Hitchcock and H. Sprunt, "Analyzing multiple register sets, *Proc. 12th Annual Int. Symposium on Computer Architecture, IEEE Press,* pp. 55-63(June 1985).

[10] D. Halbert and P. Kessler, "Windows of overlapping register frames," *CS 292R Final Class Report,* Computer Science, Div., U.C. Berkeley (June 1980).

[11] V. Milutinovic, "Microprocessor architecture and designe for GaAs technology," *Microelectronics Journal,* vol. 19, pp. 51-56 (July/Aug. 1988).

[12] D. Patterson and C. Sequin, "A VLSI RISC," *IEEE Computer,* vol. 15, pp. 8-21(Sept. 1982).

[13] G. Jung, D. Meyer, and V. Milutinovic, "Conmparison and evaluation of two catalytic migration approaches for the Design of widowing-oriented register file structures," *Proc. of the HICCS-23,* pp. 58-64,(Jan. 1990).

[14] G. Jung, D. Meyer, and V. Milutinovic, "Flexible register widowing strategy for multi-task, " *Proc. of the HICCS-24,*(Jan. 1991).

[15] Y. Tamir and C.H. Sequin, "Stregies for Memory Managing the Register File in RISC," *IEEE Trans. on Compters,* vol. 32, pp. 977-988,(Nov. 1983).

著 者 紹 介

鄭 己 鉉(正會員)

1958年 10月 21日生, 1984年 서강대학교 전자공학과 졸업. 1988年 Univ. of Illinois(공학석사). 1990年 Purdue Univ. (공학박사). 1984年～1986年 Fujitsu Korea system Engineer. 1991年～1992年 2月 현대전자반도체연구소 책임연구원. 1992年 3月～현재 아주대학교 전자공학과 조교수. 주관심분야는 컴퓨터구조, VLSI설계 및 Multi-media 분야 등임.