하드웨어 트레이스 생성 시스템의 개발

윤 형 민[†] · 박 기 호[†] · 이 길 환[†] · 한 탁 돈[††]
김 신 덕[††] · 양 성 봉[†††] · 이 용 석[††††]

요 약

캐쉬 메모리 시스템의 성능 측정 방법으로 이제까지 널리 사용되고 있는 방법이 트레이스 구동 시뮬레이션이다. 트레이스 구동 시뮬레이션의 정확성은 사용하는 트레이스의 크기, 포함된 정보의 종류 등에 의해서 크게 영향을 받는다. 이에 따라 보다 정확한 트레이스를 생성하기 위해 많은 방법들이 제안되었으며 그 중 하드웨어 모니터링 기법에 의해서 얻어진 트레이스는 응용 프로그램의 메모리 참조에 대한 정보뿐만 아니라, 문맥 교환이나 시스템 프로그램의 메모리 참조에 대한 정보, 메모리 참조가 발생한 시간 정보 등을 가진다는 장점을 갖는다. 그러나 하드웨어 모니터링 시스템은 트레이스를 생성하기 위한 시스템에 따라서 설계가 변화되어야 하는 단점이 있다. 본 논문에서는 이러한 하드웨어 모니터링 시스템의 단점을 완화하기 위해서 EPLD(Erasable Programmable Logic Device)를 사용하여 트레이스 생성 시스템을 구성하여, 보다 간단한 수정으로 여러 시스템에서 트레이스 생성이 가능한 하드웨어 시스템을 설계하였다. 또한 제작된 트레이스 생성 시스템은 66Mhz의 고속 버스 시스템에서 동작할 수 있는 특징을 갖는다.

Development of Hardware Trace Generating System

Hyung-Min Yoon[†] · Gi-Ho Park[†] · Kil-Whan Lee[†] · Tack-Don Han[††]
Shin-Dug Kim[††] · Sung-Bong Yang[†††] · Yong-Surk Lee[††††]

ABSTRACT

Trace-driven simulation is widely used method to evaluate the performance of the cache memory systems. The accuracy of trace-driven simulation highly depends on the length of the traces and the information contained in the traces. Various methods are used to generate the accurate traces. Among these methods, hardware monitoring technique generates the traces which have more information than others. The traces obtained by hardware monitoring technique, have much information such as about context switch, OS references, and timing of memory reference. However, hardware monitoring system has a drawback that it is difficult to be modified to get the traces from various systems. In this paper, a hardware monitoring system is designed using the EPLD(Erasable Programmable Logic Device)s to alleviate the drawback and easy to get the traces from various systems. The designed hardware monitoring system can be operated with high clock frequency up to 66Mhz.

※본 논문은 1995년도 한국학술진흥재단의 대학부설연구소 연구과제 연구비에 의하여 연구되었음.

† 준 회 원:연세대학교 컴퓨터과학과 병렬처리시스템 연구실
†† 정 회 원:연세대학교 컴퓨터과학과 병렬처리시스템 연구실
††† 정 회 원:연세대학교 컴퓨터과학과 알고리즘 연구실
†††† 정 회 원:연세대학교 전자공학과 VLSI&Cad 연구실

논문접수:1997년 11월 4일, 심사완료:1997년 12월 15일

1. 서 론

컴퓨터 시스템에서 CPU(Central Processing Unit)와 메모리간의 성능 격차가 심해짐에 따라 효율적인 메모리 시스템의 구성이 중요하게 되었으며 이에 따라 효율적인 메모리 시스템 구성을 위하여, 대부분의 컴퓨터 시스템에서는 캐쉬 메모리를 사용하고 있다.

캐쉬 메모리 시스템의 평균 메모리 접근 시간 등은 프로세스의 스케줄링 시의 스케줄링 시간이나 다중 쓰레드 시스템에서의 레지스터의 수, 시스템에서 지원하는 쓰레드의 수 등을 결정하는데 중요한 요소로 작용한다. 이러한 캐쉬 시스템의 성능을 측정하기 위해서 트레이스 구동 시뮬레이션(Trace Driven Simulation)이 많이 사용되고 있으며, 정확한 트레이스 구동 시뮬레이션은 시스템 설계 시에 중요한 역할을 하게된다[13][14][17].

트레이스 구동 시뮬레이션은 시스템에 작업부하(workload)를 주어서, 해당 작업부하가 수행되는 과정을 트레이스(trace)로 생성한 후, 평가하고자 하는 시스템의 모델의 입력으로 사용해서 성능평가를 수행하는 방법이다. 트레이스 구동 시뮬레이션의 정확도는 평가하고자하는 시스템의 모델의 정확도와 트레이스의 정확도에 의해서 결정되는데, 시스템 모델의 정확도는 필요에 따라서 보다 세밀한 모델링을 통하여 높일 수 있으나, 트레이스의 정확도는 트레이스를 얻는 방법에 의해서 트레이스 생성 시에 결정된다[12][13]. 따라서 정확한 성능평가를 위해서는 성능평가에 요구되는 많은 정보를 가지고 있는 완전한 트레이스를 생성하는 것이 필수적이다.

또한 정확한 트레이스 구동 시뮬레이션을 수행하기 위해서는 단순한 메모리 참조이외의 다양한 정보를 가진 트레이스가 요구된다. 운영체제(Operating System, OS)에 의한 메모리 참조나 문맥교환(context switch) 등의 정보를 포함하지 않는 트레이스를 이용한 성능평가는 실제보다 캐쉬 시스템의 성능을 과대평가 하게 하며[12], 선인출 기법 등에 대한 성능평가는 메모리 참조가 발생한 시간에 대한 정보가 성능평가에 중요한 영향을 미친다.

이러한 트레이스를 생성하는 방법으로 하드웨어 모니터링, 마이크로코드 수정, 트랩 비트 방법, 프로세서 시뮬레이션 등이 있으며, 이 중 하드웨어 모니터링 방법에 의해서 가장 완전하고 많은 정보를 가지는 트레이스 정보를 얻을 수 있다[4][6][12]. 본 연구에서는, 이러한 장점을 가지는 하드웨어 모니터링을 수행하는 재구성이 용이한 트레이스 생성 시스템(Reconfigurable Trace Generating System, RTGS)를 설계하였다.

RTGS는 트레이스 하고자 하는 시스템의 변경이나 생성하고자하는 트레이스의 요구의 변화에 보다 효율적으로 적용할 수 있도록, 재설정 가능하고 재프로그래밍 가능한 제어 부분을 갖추도록 설계하였다. 이러한 제어 부분을 가지도록 제어의 핵심 부분을 EPLD(Erasable Programmable Logic Device)로 설계하였으며. 제어부분을 구성하는 EPLD를 다시 프로그래밍 하는 것에 의해서 여러 시스템들에 사용될 수 있다[3][11]. 또한 설계된 트레이스 생성 시스템은 최고 66Mhz의 고속에서도 동작이 가능하다는 특징을 갖는다.

본 논문의 구성은 다음과 같다. 2장에서는 트레이스를 생성하는 기법을 소개하고 이들의 장단점을 제시한다. 본 논문에서 설계한 하드웨어 모니터링 시스템인 RTGS의 구성과 동작을 3장에서 설명한다. 4장에서는 RTGS의 특징 및 장점을 보이며, 5장에서는 본 논문의 결론을 제시한다.

2. 관련 연구

트레이스 구동 시뮬레이션에 요구되는 트레이스를 생성하기 위하여 다양한 트레이스 생성 기법이 제안되었다. 본 장에서는 이러한 방법들을 소개하고, 이들의 장단점에 대해서 살펴보도록 하겠다.

2.1 트레이스 생성 기법들

이제까지 많은 트레이스 생성 방법들이 제안되었으며, 이들은 명령어 수정 방법, 마이크로 코드 수정방법, 트랩비트방법, 프로세서 시뮬레이션, 하드웨어 모니터링 등이다[1][4][13][14][16][17].

2.1.1 명령어 수정 방법

명령어 수정 방법은 트레이스 되는 프로그램이 참조하는 메모리의 가상 주소를 생성하도록 하는 명령어를 삽입함으로써 트레이스를 생성하는 기법이다. 이러한 가상 어드레스 트레이스는 적당한 TLB 시뮬

레이터를 써서 캐쉬 메모리 연구에 적당한 물리 어드레스로 변환될 수 있다. 이러한 명령어 수정 방법은 프로그램의 소스코드에 트레이스 생성을 위한 명령어를 삽입하는 소스 코드 인라이닝 (source code inlining), 오브젝트 코드에 트레이스 명령어를 삽입하는 오브젝트 코드 인라이닝 (object code inlining) 등의 방법이 있다.

2.1.2 마이크로 코드 수정방법

마이크로코드 수정방법은 메모리를 참조하는 명령어에 해당하는 마이크로 코드부분에 트레이스를 생성하는 명령어를 삽입하여 트레이스를 생성하는 기법이다. 마이크로코드 수정 방법의 사용은 마이크로코드 방식의 수행을 하는 컴퓨터에서만 수행이 가능하며, 이러한 수행방식을 가지는 VAX 계통의 프로세서에서 사용되었다.

2.1.3 트랩 비트 방법

트랩 비트 방법은 각 명령어의 수행마다 CPU에 인터럽트를 발생시켜서 해당 인터럽트 루틴이 트레이스를 저장하여 트레이스를 생성하는 방법을 말한다.

2.1.4 프로세서 시뮬레이션

프로세서 시뮬레이션 방법은 트레이싱하려는 프로세서와 시스템 환경을 모델링하여 이러한 환경 하에서 응용프로그램을 수행시키면서 트레이스를 생성하는 방법이다.

2.1.5 하드웨어 모니터링 방법

하드웨어 모니터링 방법은 작동중인 시스템에 수동적인 시스템(논리 분석기, 데이터 획득 시스템등)을 연결시켜 동작중인 CPU에서 발생되는 신호를 모니터링 함으로써 트레이스를 생성하는 방법이다[6][16].

위에서 제시한 트레이스 생성기법들은 각각의 장·단점을 가지고 있으며, 다음절에서는 이들 방법들의 장단점을 분석하고, 본 연구에서 개발한 하드웨어 모니터링 시스템의 필요성에 대해서 이야기하겠다.

2.2 트레이스 생성 기법의 비교

위에서 제시한 트레이스 생성 기법은 각 기법 나름대로의 장·단점을 가지고 있다. 본 절에서는 트레이스 생성 기법을 비교하는 기준을 제시하며, 각 기법을 제시한 기준에 의해서 평가하여, 이들의 장·단점을 살펴본다.

트레이스 생성 기법들은 다음의 평가 기준들에 의해서 분류될 수 있으며, 이들 기준을 만족하는가 그렇지 못한가에 의해서 정확하고 완전한 트레이스인지가 결정된다.

(1) 트레이스가 그 데이터를 생성하기 위해서 사용되는 시스템에 의해 영향을 받는가?
(2) 트레이스가 어드레스 참조에 대한 것뿐만 아니라 작업의 동작과 동기화에 대한 정보들을 가지고 있는가?
(3) 임의의 수의 프로세스들에 대한 큰 트레이스들도 생성할 수 있는가?
(4) 트레이스를 생성하는데 요구되는 시간이 어느 정도인가?

위에서 제시한 평가 기준들을 만족하지 못하는 트레이스 생성 기법들에 의해서 생성된 트레이스들은 디스토션 (distortion)을 가지게 되며[1], 이러한 디스토션의 정도에 따라서 생성된 트레이스의 정확도가 결정된다. 이러한 트레이스에 존재하는 디스토션은 〈표 1〉에서 보이는 바와 같이 누락(omission), 스타트업(startup), 구현(implementation), 시간(time) 디스토션의 4가지로 분류된다[4][12][17].

〈표 1〉 디스토션의 종류
〈Table 1〉 Various types of distortion

디스토션	설 명
누락(omission) 디스토션	입출력, 인터럽트, 문맥교환, 운영체제동작 등의 정보를 얻어내지 못하는 경우
스타트업(startup) 디스토션	트레이스의 크기가 매우 작아서 이로 인해 시뮬레이션 결과가 신뢰성이 없는 경우
구현(implementation) 디스토션	트레이스의 생성이 특정한 하드웨어에 의존하는 경우
시간(time) 디스토션	트레이스 하는 프로그램의 수행속도가 느려지는 경우

누락 디스토션이란 운영체제에 의한 메모리 참조나, 문맥교환으로 인한 메모리 참조 등에 대한 정보

를 가지지 않는 트레이스를 발생시키는 경우에 발생하는 디스토션이다. 트레이스 생성 기법에 의해서 얻어진 트레이스의 길이가 짧은 경우에는 응용 프로그램의 전체 수행시간동안의 메모리 참조에 의한 성능 분석이 아닌 순간적인(transient) 메모리 참조에 의한 성능 분석에 의해서 캐쉬 메모리의 경우에는 초기 접근 실패(compulsory miss)만을 강조하게 되는 등의 디스토션을 발생시킬 수 있으며, 이러한 디스토션은 스타트업(startup) 디스토션이라고 한다. 트레이스 발생 기법이 특정한 하드웨어에서만 트레이스를 발생시킬 수 있는 경우에 가지게 되는 디스토션이 구현 디스토션이며, 트레이스를 발생시키기 위한 방법에 의해서 트레이스를 생성하는데 사용된 프로그램의 수행속도가 느려진 경우에 프로세서의 스케줄링 순서 등에 영향을 주어서 부정확한 트레이스를 얻게 되는데 이에 의한 디스토션을 시간 디스토션이라 한다.

2.1절에서 소개한 트레이스 생성 방법들은 위의 디스토션들을 지니게 되며, 이들을 〈표 2〉에서 제시하고 있다.

〈표 2〉에서 보면 명령어 수정 기법은 수행되는 응용 프로그램 내에 트레이스 생성을 위한 명령어를 삽입하므로, 운영체제 참조나 문맥 교환 등의 정보를 가지고 있지 못하기 때문에 누락 디스토션을 가지며, 상대적으로 긴 트레이스를 생성할 수 있고, 코드 상에서의 변환에 의한 것이므로 스타트업, 구현, 시간 디스토션은 작다[14][17]. 트랩 비트와 프로세서 시뮬레이션 방법은 명령어 수정 방법과 마찬가지로 누락 디스토션이 크나, 시스템 모델을 변화시키기 쉽기 때문에 구현 디스토션이 없다. 프로세서 시뮬레이션의 경우에 모델의 정확도를 운영체제나 인터럽트 등까지 모델링 할 수 있을 정도로 구현하기는 어렵기 때문에 누락 디스토션을 갖는다. 마이크로코드 수정 방법은 디스토션이 매우 낮은 정도지만 VAX계통의 마이크로프로세서에서만 사용 가능한 단점이 있다[1][17].

〈표 2〉 트레이스 생성 기법에 따른 디스토션들
〈Table 2〉 Distortion of each trace technique

트레이스 생성 기법들		누락 디스토션	스타트업 디스토션	구현 디스토션	시간 디스토션
명령어 수정		HIGH	LOW	LOW	LOW
마이크로코드 수정		NONE	HIGH	HIGH	LOW
트랩 비트		HIGH	LOW	NONE	HIGH
프로세서 시뮬레이션		HIGH	LOW	NONE	HIGH
하드웨어 모니터링	논리 분석기 데이터 획득 시스템	NONE	HIGH	HIGH	NONE
	BACH	NONE	LOW	HIGH	NONE
	RTGS	NONE	LOW	LOW	NONE

이에 반해 하드웨어 모니터링 방법은 수행 중인 시스템의 마이크로프로세서를 직접적으로, 동시에 모니터링을 하기 때문에, 문맥 교환(context switch), 인터럽트, 운영체제의 정보를 얻지 못하는 누락 디스토션이 없으며 트레이싱으로 인한 지연을 가지는 시간 디스토션이 없다. 또한 CPU에서 나오는 시간 정보, 전체 어드레스, 데이터들이 논리 분석기(logic analyzer), 또는 데이터 획득 시스템(data acquisition system)등을 통해서 얻어질 수 있다[16][17]. 그러나 이 방법은 버퍼의 크기가 작고, 수행중인 프로그램을 잠시 중지시키고 재 시작하는 홀팅(halting)기법이 없기 때문에 긴 트레이스를 생성할 수 없으며, 이로 인해 높은 스타트업 디스토션을 가진다[4][12].

위에서 제시한 여러 디스토션 중 누락 디스토션에 의한 시뮬레이션 결과의 오차는 매우 큰데, 이는 특히 문맥교환이나 운영체제에 의한 메모리 참조가 트레이스에 반영되지 않은 경우에 매우 크게 나타난다. 일반적으로 캐쉬 실패는 매우 다양한 양상을 보이며 발생하고, 많은 캐쉬 접근 실패가 인터럽트와 시스템 콜과 같은 시스템 코드에 의해서 발생하며 상대적으로 사용자 코드에 의한 캐쉬 실패는 적다. 이에 따라서 과거에 많이 사용되었던 사용자 코드만 가진 트레이스에서 예측된 것에 비해 시스템 콜, 인터럽트 등을 포함하는 완전한 트레이스를 이용한 시뮬레이션에 의해 예측된 캐쉬 실패율이 50배 이상이나 큰 경우도 발생한다[12]. 이는 [12]에서 보이는 바와 같이 운영체제에 의한 참조의 비율은 12%에서 28%밖에 되지 않으나, 운영체제에 의한 메모리 참조는 캐쉬 내에서 많은 간섭(interference)에 의한 캐쉬 접근 실패를 발생시키기 때문에 운영체제 참조에 의해 발생되는 캐쉬 간섭이 캐쉬 실패율을 크게 증가시키기 때문이다. 이와 같이 운영체제 등의 참조를 포함하지 않은 트레이스를 이용한 캐쉬 시스템의 성능평가는

실제보다 너무나 낙관적인 성능평가 결과를 제공하며, 이에 따른 메모리 시스템의 설계 및 전체 컴퓨터 시스템 설계 변수의 결정은 매우 심각한 시스템의 성능 저하를 야기시킨다.

이러한 이유로 누락 디스토션을 가지지 않는 트레이스 생성 기법인 하드웨어 모니터링 기법을 수행하는 하드웨어 모니터링 시스템의 중요성이 대두되게 되었으며, 이러한 하드웨어 모니터링 시스템이 개발되었다[4][6][12].

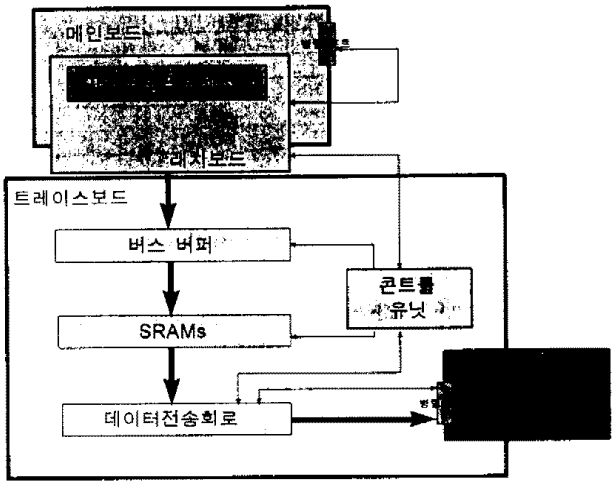
그러나, 이러한 하드웨어 모니터링 시스템은 하드웨어 모니터링 시스템 내에 존재하는 메모리의 양의 제한에 의해서 짧은 트레이스만을 생성하게 됨으로 스타트업 디스토션을 가지며, 하드웨어에 의존적으로 하드웨어 모니터링 시스템을 설계하게 됨으로 구현 디스토션을 갖는 단점을 가지고 있다. 이러한 단점 중에서 스타트업 디스토션을 해소하기 위해서 BYU(Brigham Young University)에서 제작한 BACH (BYU Address Collection Hardware) 시스템은 높은 우선 순위의 인터럽트 기법과 내부 버퍼를 사용해서 긴 트레이스를 생성하므로 써 낮은 스타트업 디스토션을 가지도록 설계되었다[4][6][12]. 그러나, BACH도 다른 하드웨어 모니터링 방법들처럼 구현 디스토션을 가지는 단점을 가진다.

이에 따라 본 연구에서는 BACH가 가지는 하드웨어 모니터링 시스템의 장점을 가지며 EPLD(Erasable Programmable Logic Device)를 사용해서 설계하여 재 프로그래밍 하는 것에 의해 쉽게 여러 가지 시스템들에 적용시킬 수 있도록 하여 낮은 구현 디스토션을 가지는 하드웨어 모니터링 시스템을 설계하였다.

3. 재구성이 용이한 하드웨어 트레이스 생성 시스템(Reconfigurable Trace Generating System)의 설계

본 절에서는 하드웨어 모니터링 기법의 장점과 함께, 낮은 구현 디스토션을 가지도록 제작된 RTGS의 구조와 동작 방식에 대해서 설명하도록 한다.

RTGS는 동작중인 마이크로프로세서에서 참조하는 메모리 어드레스를 캡쳐하여 정확하고 완전한 트레이스 데이터를 생성하고, 이를 이용하여 트레이스 구동 시뮬레이션을 하기 위하여 설계되었다. 이를 위하여 마이크로프로세서의 핀에 직접 트레이싱 시스템을 연결하여 마이크로프로세서에서 참조하는 메모리 어드레스와 데이터 및 제어를 시스템의 버퍼에 저장한다. 그리고, 버퍼가 다 채워지면 이 버퍼에 저장된 내용을 저장 분석 서버로 전송하기 위해 트레이스되는 시스템에 안정적인 인터럽트를 걸고, 이 컴퓨터에서 수행중인 프로그램을 중지시키고, 저장 분석 서버로 버퍼에 저장된 내용을 전송하여 저장한다.



(그림 1) RTGS 전체 구성도
(Fig. 1) RTGS structure

설계된 트레이스 시스템은 (그림 1)에서 보는 바와 같이 다음의 세 요소로 구성된다.

◆래치 보드(Latch Board): 트레이스 되는 마이크로프로세서의 신호를 캡쳐하여 트레이스 보드로 전송
◆트레이스 보드(Trace Board): 데이터를 저장하고 저장 분석 서버로 전송
◆저장 분석 서버(Storage and Analysis Server): 전송받은 데이터를 저장하고 분석

설계된 트레이스 시스템은 클락 스피드가 약 66Mhz까지 동작한다. 캡쳐되는 데이터의 크기는 최대 100bit이며 이와 동시에 8bit을 클락 정보를 저장하는데 사용한다. Intel 80486의 경우 〈표 3〉에 나타난 바와 같이 30비트의 주소, 32비트의 데이터, 메모리 콘트롤, 버스 콘트롤 등을 저장하는데 사용된다[5].

〈표 3〉 캡춰되는 주요 intel 80486 신호들
〈Table 3〉 Main signals of intel 80486 captured

크기(비트)	신호 이름	신호 기능
30	A31-A2	주소
32	D31-D0	데이터
4	BE3-BE0	바이트 인에이블
4	DP3-DP0	데이터 패리티
1	M/IO	메모리/입출력
1	D/C	데이터/콘트롤
1	W/R	쓰기/읽기
1	ADS	어드레스 상태
1	RDY	현재 버스 사이클의 종료
1	BS16	버스 사이즈 16
1	BS8	버스 사이즈 8
1	BREQ	내부적 버스 사이클 펜딩(pending)
1	BLAST	버스트 버스 사이클의 종료
1	LOCK	버스 사이클 잠김(locked)

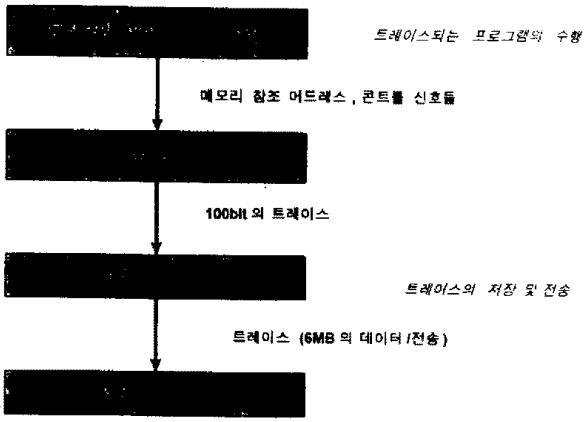
제작된 트레이스 시스템은 66Mhz에서 약 100비트의 데이터를 샘플링 하므로, 초당 약 6.6Gb를 처리할 수 있다. 이를 위하여 66Mhz EPLD와 고속 F 시리즈 TTL과 66Mhz 동기식 SRAM를 사용하였다. 그리고, 버퍼의 데이터를 다른 컴퓨터로 전송하기 위하여 동작중인 컴퓨터에 대한 적절한 홀팅(halting) 기법이 필요하며 이는 높은 우선 순위를 가지는 하드웨어 인터럽트 방식을 사용하였다[4][8].

RTGS는 클락 스피드가 약 66Mhz까지 동작하며 이는 현재 생산되는 컴퓨터의 버스 스피드가 기존의 33Mhz에서 66Mhz로 고속화되고 있고 이런 고속 버스를 가지는 시스템에서 트레이스를 받기 위해서이다.

3.1 RTGS의 동작

RTGS는 동작 중인 마이크로 프로세서를 모니터링하여 메모리 참조 어드레스, 여러 콘트롤 신호를 트레이스로 생성한다. (그림 2)에서 보이는 바와 같이 동작 중인 마이크로 프로세서의 신호들을 래치 보드에서 캡춰하여 트레이스 보드로 전달하고, 트레이스 보드는 이 데이터들을 버퍼에 저장하고 저장 분석 서버로 전송하여 트레이스 구동 시뮬레이션을 위한 트레이스를 생성한다.

제작한 트레이스 시스템의 동작은 (그림 3)에서 보이고 있는 바와 같이 동작한다. ①~③의 동작은 트레이스의 생성을 시작하며, ④~⑧의 동작은 트레이스를 생성, 저장 및 데이터 전송의 역할을 수행하며 트레이스 되는 프로그램의 종료 시까지 반복한다. ⑨~⑩은 트레이스의 종료시의 동작이다.

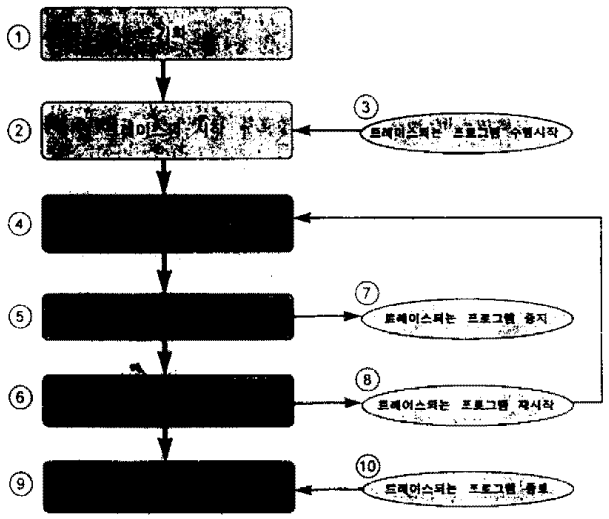


(그림 2) RTGS 데이터 흐름도
(Fig. 2) RTGS data flow

우선 동작중인 컴퓨터와 동기화를 하고 전체 보드에 대한 테스트와 초기화를 수행한다(①). 그리고, 트레이스 시작 신호를 트레이스 시스템으로 전달할 루틴이 설치된 프로그램을 트레이스 되는 시스템에서 수행하면(③), 이와 동시에 트레이스 시스템은 트레이스를 시작하고 마이크로 프로세서에서 출력되는 콘트롤 신호를 모니터링 한다(②). 래치 보드에서 마이크로 프로세서의 콘트롤 신호를 디코딩하여 캡춰된 데이터가 유효하면 트레이스 보드로 전송한다(④).

이렇게 전송 받은 데이터는 트레이스 보드의 SRAM에 저장되며 이렇게 저장을 하는 동안 트레이스 보드는 SRAM의 주소와 콘트롤을 생성하면서 저장된 데이터의 용량이 SRAM의 용량을 넘지 않도록 감시한다(⑤). 트레이스 보드는 SRAM 용량만큼 데이터가 저장되었을 때 트레이스 되는 시스템에 높은 우선 순위의 인터럽트를 걸어 트레이스 되는 프로그램의 동작을 중지시키고(⑦), 트레이스 보드의 데이터 전송 회로가 SRAM의 데이터를 저장 분석 서버의 병렬 포트를 통해 저장 분석 서버로 전송하도록 한다(⑥).

SRAM의 데이터를 모두 전송하고 난 후, 트레이스 보드는 데이터를 저장할 준비를 하고 트레이스 되는 컴퓨터에 재시작 신호를 보내 트레이스 되는 프로그램을 정상적으로 재실행하고 트레이스 생성을 계속한다(⑧). 트레이스 되는 프로그램의 실행이 종료되면 트레이스 되는 시스템에서 트레이스 시스템으로 트레이싱의 종료를 지시하는 신호가 발생되며(⑩), 트레이스 시스템은 현재 SRAM에 저장된 데이터를 모두 저장 분석 서버로 전송한 후 초기상태로 전환한다(⑨). 한편 데이터를 전송 받은 저장 분석 서버는 전송 받은 데이터를 분류하고 콘트롤 정보 및 클락 정보를 디코딩하여 저장한다.



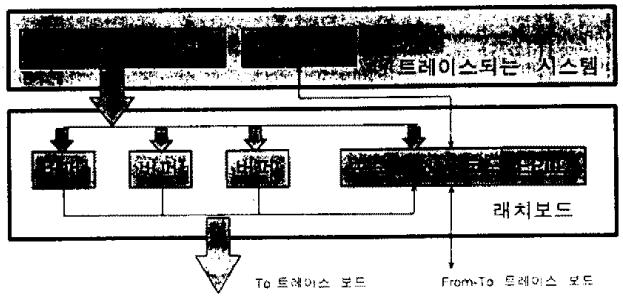
(그림 3) RTGS 동작 흐름도
(Fig. 3) RTGS operation flow

3.2 래치 보드 (Latch Board)

래치 보드는 (그림 4)에서 보이는 바와 같이 EPLD 4개를 이용하여 설계되었으며 이중 하나의 EPLD는 트레이스 되는 시스템의 마이크로프로세서에서 발생되는 〈표 3〉의 버스 콘트롤 신호들을 디코딩 하는 역할을 하며 데이터의 저장 여부를 트레이스 보드로 전달한다. 나머지 3개의 PLD는 FIFO 방식으로 동작하는 버퍼로 트레이스 되는 시스템의 데이터를 저장, 출력한다.

FIFO 방식으로 동작하는 버퍼인 3개의 PLD는 2 clock의 200비트의 데이터를 저장하고 있으며 이는 트레이스 되는 시스템의 콘트롤을 디코딩하여 버스 버퍼와 SRAM을 콘트롤하는 신호가 생성되기까지의 클락 사이클과 타이밍을 맞추어 주기 위해서이다.

래치 보드는 트레이스 되는 시스템의 마이크로프로세서의 핀에 직접 연결되어 있으며 인터럽트 신호를 주고 받기 위하여 트레이스 되는 시스템의 병렬 I/O 포트에 연결되어 있다.

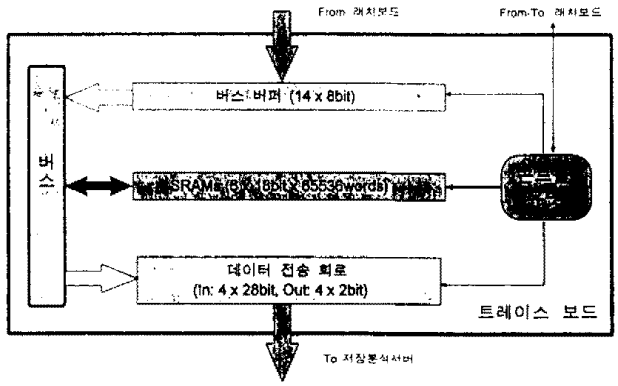


(그림 4) 래치 보드 구성도
(Fig. 4) Latch board structure

3.3 트레이스 보드 (Trace Board)

(그림 5)에서 보이는 바와 같이 트레이스 보드는 콘트롤을 위한 EPLD, 버스버퍼, SRAM, 그리고 PC로의 전송을 위한 데이터 전송 회로로 구성되어 있다. EPLD는 Altera MAX7000 시리즈를 사용하였으며 버스버퍼는 TTL F 시리즈를 사용하여 고속 클락에 정확히 작동하도록 하였다. 그리고 SRAM은 클락에 의해 작동하는 고속 동기식 SRAM을 6MB 사용하였다.

버스는 버스버퍼에서 들어오는 데이터를 SRAM에 전달하여 주거나 SRAM의 데이터를 저장 분석 서버



(그림 5) 트레이스 보드 구성도
(Fig. 5) Trace board structure

로의 전송을 위한 데이터 전송 회로로 전달하는 목적으로 구성된다.

다음의 각 소절에서는 트레이스 보드의 각 기능 블록들에 대해서 설명한다.

3.3.1 콘트롤 유닛 (Control Unit)

콘트롤 유닛은 트레이스의 생성 및 저장을 위하여 래치 보드 및 트레이스 보드를 제어하는 역할을 한다. 콘트롤 유닛의 기능은 래치 보드에 대한 제어와 트레이스 보드 내부의 버스 버퍼, SRAM에 대한 제어 신호를 발생시키는 기능, 데이터 전송 회로를 동작시키고 SRAM의 데이터를 읽어 전송하는 기능을 수행한다.

트레이스의 시작 신호가 트레이스 되는 시스템으로부터 발생되어 전달되면, 래치 보드로부터 데이터가 유효한지를 지시하는 신호에 의해서 SRAM을 콘트롤하는 신호들을 발생시킨다. 래치 보드로부터 데이터가 유효하다는 신호가 전달되면 래치 보드로부터 전송되는 데이터를 SRAM에 저장하기 위한 신호들을 발생시킨다. 그리고, SRAM에 데이터가 꽉 차면 트레이스 되는 프로그램을 중지시키기 위한 높은 우선 순위의 인터럽트를 발생시키기 위하여 신호를 발생시키고, SRAM의 데이터를 저장 분석 서버로 전송하기 위하여 데이터 전송 회로를 동작시킨다. 또한, 데이터 전송 회로를 제어하여 SRAM의 데이터를 저장 분석 서버로 전송하도록 하고, SRAM의 데이터 전송이 끝났을 때 래치 보드에 신호를 보내서 다시 트레이스를 생성하도록 한다. 데이터 전송 회로가 동작하면 콘트롤 유닛은 데이터를 SRAM에서 읽어오는 신호들을 발생시키며, 타이밍을 맞추기 위해서 데이터 전송 회로와 동기화 되어 동작한다.

콘트롤 유닛에서 나오는 모든 신호는 레지스터된 출력으로 클락에 의해 동기화 되어 나온다[11]. 콘트롤 신호의 팬 아웃으로 인한 전달 지연이나 전압 하강을 방지하기 위하여 같은 콘트롤 신호를 몇 개의 핀으로 분산하여 동시에 내보내고 이 신호들이 제어되는 IC(Integrate Circuit)에 분산되어 전달되도록 설계하였다[7][10].

3.3.2 버스버퍼 및 SRAM

버스버퍼로 보통 회로와 버스사이의 3 스테이트의 게이트를 가지는 74F244를 사용하였다. 74F244는 다른 TTL 게이트에 비해 높은 전류를 드라이브할 수 있다[7]. 74F244는 라인 리시버(line receiver)로 사용되면 라인의 작은 전압 불안정(voltage perturbation)을 무시할 수 있기 때문에 래치 보드로부터 케이블을 통해 데이터를 받는데 사용된다[7].

트레이스를 저장하는 버퍼로 사용할 SRAM은 66Mhz에서 동작하는 동기식 SRAM을 사용하였으며 18비트의 65536 워드로 구성되어 있다. 쓰기 사이클은 내부적으로 자동 제어되며 동기화 되어 동작하며 3 스테이트 입력과 출력을 가진다[15].

(그림 6) Intel 486DX system에서 동작하도록 제작된 RTGS
(Fig. 6) RTGS manufactured for operating with intel 486DX system

3.3.3 데이터 전송 회로 (Extraction Circuit)

데이터 전송 회로는 EPLD 4개로 구성되어 있으며 이중 하나가 마스터로 나머지는 슬레이브로 동작한다.

마스터는 콘트롤 유닛의 콘트롤 신호를 디코딩하여 트레이스가 멈춘 것을 확인한 후 SRAM의 데이터를 저장 분석 서버로 전송할 준비를 한다. 저장 분석 서버의 프로그램과 동기화가 이루어지면, 마스터는 콘트롤 유닛에게 SRAM의 데이터를 읽어 버스로 전달되도록 하는 신호를 전달한다. 마스터는 각 슬레이브에 데이터 읽기 신호와 데이터 출력 동기화 신호를 보내 데이터 전송을 제어한다. 총 108비트의 데이터를 모두 저장 분석 서버로 전송하고 나면 마스터는

콘트롤 유닛에게 다시 SRAM의 데이터를 읽는 신호를 내보낸다.

저장 분석 서버의 병렬 포트를 통한 전송 속도가 느리기 때문에 데이터 전송 회로는 래치된 출력을 하여 저장 분석 서버에서 데이터를 읽는 동안 전송라인에 데이터를 유지시킨다[2][8].

3.4 저장 분석 서버 (Storage and Analysis Server)

저장 분석 서버에서의 프로그래밍은 트레이스 보드로부터 데이터를 받아 디코딩을 하여 디스크에 저장하고 저장된 데이터를 분석하는 역할을 수행한다.

트레이스 보드와 저장 분석 서버와의 데이터 전송선은 일반적으로 많이 이용되는 8255A를 콘트롤러로 사용하는 병렬 포트를 사용하였다. DIO(Digital Input Output) 보드와 같은 데이터 전송을 위한 특별한 하드웨어에 비교하면 전송속도가 느리다는 단점이 있지만 일반적인 컴퓨터에 많이 사용되는 병렬 포트를 이용함으로써 프로그램의 호환성을 높였다[2][7][8].

트레이스 보드의 데이터 전송 회로에서 데이터를 내보내는 신호가 저장 분석 서버에서 인식되면 병렬 포트를 통하여 8비트의 데이터를 14번 전송 받아 108비트로 재구성하여 디스크에 저장한다. 트레이스 보드의 데이터가 모두 디스크에 저장되고 나면 저장 분석 서버는 디스크에서 저장된 데이터를 분석하고 출력하여 준다.

4. RTGS에 의한 트레이스의 생성 및 분석

여기에서는 본 연구에서 설계된 하드웨어 모니터링 시스템인 RTGS의 트레이스의 생성 결과와 대하여 설명한다.

RTGS에 의한 트레이스 생성은 Intel 486DX 33Mhz에서 수행하였으며 트레이스 되는 프로그램은 〈표 4〉에 제시된 예제 프로그램을 사용하였다. 트레이스 되는 예제 프로그램은 운영체제의 메모리 참조를 구별하기 위하여 getdate( ) 함수를 사용하였으며 이 함수의 결과 값은 C 라이브러리에서 지원하는 date형식의 구조체에 저장된다. 〈표 5〉의 트레이스에 나타난 바와 같이 〈표 4〉의 (ㄴ)(ㄷ)은 ①②⑨⑩의 메모리 참조를 발생시키며 〈표 4〉의 (ㄹ)은 ③④⑤⑥의 운영체제에 의한 메모리 참조를 발생시킨다. 〈표 4〉의 (ㄱ)은 레지스터 내부에 변수를 선언하였기 때문에 메모리 참조가 일어나지 않는다.

〈표 4〉 트레이스 되는 예제 프로그램
〈Table 4〉 Traced sample program

```
void main(void) {
    register int i;
    unsigned int pseg, qseg;
    unsigned int poff, qoff;
    int *p, *q;
    struct data *datep;

    outportb(0×378, 0×00); /* Port Initialization */
    outportb(0×378, 0×01); /* Tracing Start */

(ㄱ) for (i=0; i<35000; i++) {
(ㄴ)        *p=i;
(ㄷ)        *q=i;
(ㄹ)        getdate(datep);
    }

    outportb(0×378, 0×04); /* Tracing Complete */
}

struct date {
        int da_year;
        char da_day;
        char da_mon;
}

-- getdate()는 OS의 메모리 참조를 확인하기 위하여 사용되었음
```

〈표 5〉의 제작된 트레이스 생성 시스템에 의해 생성된 트레이스는 마이크로프로세서의 클락 정보 뿐만 아니라, 명령어 페치(fetch)를 포함하는 운영체제의 메모리 참조를 가지고 있음을 보여주고 있다. 또한 RTGS는 누락 디스토션뿐만 아니라 스타트업 디스토션이 없으며, 재설정 가능한 기능에 의하여 구현 디스토션이 낮은 장점을 가진다.

기존의 하드웨어 모니터링 방법인 논리 분석기(logic analyzer)나 데이터 획득 시스템(data acquisition system)을 이용한 방법은 얻어진 트레이스의 크기가 매우 작은 문제가 생기는 스타트업 디스토션과 시스템의 종류에 따라 트레이스 방법을 수정하여야 하는 구현 디스토션이 높은 단점이 있었다〈표 2〉.

Bringhum Young University에서는 스타트업 디스토션을 개선하기 위하여 기존의 논리 분석기나 데이터 획득 시스템 대신 BACH(BYU Address Collection Hardware)을 제작하였다[4][6][12]. BACH는 기존의 방법에 비해 정확하고 완전한 트레이스와 동작중인 마이크로프로세서의 콘트롤을 얻을 수 있으며 긴 프로그램의 트레이스도 인터럽트를 이용한 적절한 홀팅(halting) 기법을 가지고 트레이스를 생성할 수 있으며 25Mhz까지 동작한다. 그러나, BACH는 내부 버퍼에 저장할 데이터를 판별하기 위하여, Intel 80486의 경우에는 RDY, BRDY, EADS, BOFF, ADS,

〈표 5〉 생성된 예제 트레이스

〈Table 5〉 Sample traces generated

Clock	Address	Data	M/IO	D/C	W/R	LOCK	PLOCK	RDY	BRDY	BLAST	BREQ	BOFF	BS16	BS8	BE03	
38	176c8	3e8fe81	1	0	0	1	0	1	1	0	1	1	1	1	0000	Memory Code Read Trace Start
84	176cc	4b0e07c	1	0	0	1	0	1	1	0	1	1	1	1	0000	Memory Code Read
...																
...																
ba	176c4	46595918	1	0	0	1	0	1	1	0	1	1	1	1	0000	Memory Code Read
46	176c8	3e8fe81	1	0	0	1	0	1	1	0	1	1	1	1	0000	Memory Code Read
e6	176cc	4b0e07c	1	0	0	1	1	0	1	0	1	1	1	1	0000	Memory Code Read
e	1a2ac	171e	1	1	0	1	1	0	1	0	0	1	1	1	0011	Memory Data Read
1e	1a2ae	7246	1	1	0	1	1	0	1	0	0	1	1	1	1100	Memory Data Read
21	73b7e	0	1	1	1	1	1	0	1	0	1	1	1	1	1100	Memory Data Write ① *p=0
b1	1a2b0	fea	1	1	0	1	1	0	1	0	0	1	1	1	0011	Memory Data Read
a9	1a2b2	16	1	1	0	1	1	0	1	0	0	1	1	1	1100	Memory Data Read
85	114a	0	1	1	1	1	1	0	1	0	1	1	1	1	1100	Memory Data Write ② *q=0
55	1a2b6	30	1	1	0	1	1	0	1	0	0	1	1	1	1100	Memory Data Read
75	1a2a0	30	1	1	1	1	1	0	1	0	1	1	1	1	0011	Memory Data Write
cd	1a2b4	177c	1	1	0	1	1	0	1	0	1	1	1	1	0011	Memory Data Read
3d	176d0	ee0378ba	1	0	0	1	0	1	1	0	1	1	1	1	0000	Memory Code Read
b3	176d4	26f85ec4	1	0	0	1	0	1	1	0	1	1	1	1	0000	Memory Code Read
...																
...																
5a	b9d8	83f77207	1	0	0	1	0	1	1	0	1	1	1	1	0000	Memory Code Read
fa	b9dc	127777f9	1	0	0	1	1	0	1	0	1	1	1	1	0000	Memory Code Read
66	*1a286*	*105*	*1*	*1*	*1*	*1*	*1*	*0*	*1*	*0*	*1*	*1*	*1*	*1*	*1100*	*Memory Data Write* ③ *da_day=5 da_mon=01*
96	*1a284*	*7bc*	*1*	*1*	*1*	*1*	*1*	*0*	*1*	*0*	*1*	*1*	*1*	*1*	*0011*	*Memory Data Write* ④ *da_year=1980*
4e	b9e0	ee74f60a	1	0	0	1	0	1	1	0	1	1	1	1	0000	Memory Code Read
ee	b9e4	ea74d20a	1	0	0	1	0	1	1	0	1	1	1	1	0000	Memory Code Read
...																
...																
fa	1a280	2a06	1	1	0	1	1	0	1	0	0	1	1	1	0011	Memory Data Read
e6	1a282	fea	1	1	0	1	1	0	1	0	0	1	1	1	1100	Memory Data Read
f6	*1a284*	*7bc*	*1*	*1*	*0*	*1*	*1*	*0*	*1*	*0*	*0*	*1*	*1*	*1*	*0011*	*Memory Data Read* ⑤ *da_year=1980*
ee	*1a286*	*105*	*1*	*1*	*0*	*1*	*1*	*0*	*1*	*0*	*0*	*1*	*1*	*1*	*1100*	*Memory Data Read* ⑥ *da_day=5 da_mon=01*
7e	1a008	531042bb	1	0	0	1	1	0	1	0	1	1	1	1	0000	Memory Code Read
fe	1a288	1	1	1	0	1	1	0	1	0	0	1	1	1	0011	Memory Data Read
e1	1a28a	40	1	1	0	1	1	0	1	0	0	1	1	1	1100	Memory Data Read
f1	1a28c	fc8	1	1	0	1	1	0	1	0	0	1	1	1	0011	Memory Data Read
...																
...																
fc	18e6c	c421cd2c	1	0	0	1	1	0	1	0	1	1	1	1	0000	Memory Code Read
12	1a29e	177c	1	1	0	1	1	0	1	0	0	1	1	1	1100	Memory Data Read
72	1a2a0	30	1	1	0	1	1	0	1	0	0	1	1	1	0011	Memory Data Read
5a	1a7c	7bc	1	1	1	1	1	0	1	0	1	1	1	1	0011	Memory Data Write ⑦ datep.da_year=1980
c6	1a29e	177c	1	1	0	1	1	0	1	0	1	1	1	1	1100	Memory Data Read
36	18e70	8926065e	1	0	0	1	0	1	1	0	1	1	1	1	0000	Memory Code Read
8e	18e74	65ec40f	1	0	0	1	0	1	1	0	1	1	1	1	0000	Memory Code Read
6e	18e78	2578926	1	0	0	1	0	1	1	0	1	1	1	1	0000	Memory Code Read
de	18e7c	cb5d	1	0	0	1	1	0	1	0	1	1	1	1	0000	Memory Code Read
c1	1a288	fe8	1	1	0	1	1	0	1	0	0	1	1	1	0011	Memory Data Read
21	1a2a0	30	1	1	0	1	1	0	1	0	0	1	1	1	0011	Memory Data Read
9	1a7e	105	1	1	1	1	1	0	1	0	1	1	1	1	1100	Memory Data Write ⑧ datep.da_day=5 datep.da_mon=01
99	1a298	fe8	1	1	0	1	1	0	1	0	0	1	1	1	0011	Memory Data Read
85	1a29a	35	1	1	0	1	1	0	1	0	0	1	1	1	1100	Memory Data Read
...																
...																
5c	176cc	4b0e07c	1	0	0	1	1	0	1	0	1	1	1	1	0000	Memory Code Read
c2	1a2ac	171e	1	1	0	1	1	0	1	0	0	1	1	1	0011	Memory Data Read
d2	1a2ae	7246	1	1	0	1	1	0	1	0	0	1	1	1	1100	Memory Data Read
ea	73b7e	1	1	1	1	1	1	0	1	0	1	1	1	1	1100	Memory Data Write ⑨ *p=1
6	1a2b0	fea	1	1	0	1	1	0	1	0	0	1	1	1	0011	Memory Data Read
16	1a2b2	16	1	1	0	1	1	0	1	0	0	1	1	1	1100	Memory Data Read
2e	114a	1	1	1	1	1	1	0	1	0	1	1	1	1	1100	Memory Data Write ⑩ *q=1
be	1a2b6	30	1	1	0	1	1	0	1	0	0	1	1	1	1100	Memory Data Read
81	1a2a0	30	1	1	1	1	1	0	1	0	1	1	1	1	0011	Memory Data Write
61	1a2b4	177c	1	1	0	1	1	0	1	0	1	1	1	1	0011	Memory Data Read
...																
...																

BLAST의 6 신호만이 디코딩되어 저장 신호를 발생시키도록 설계되었고, 내부 래치가 한 클락 사이클의 데이터만을 캡춰하기 때문에 고정된 제어와 데이터 경로를 가지기 때문에 버스 구조가 다른 시스템에서 동작하기 위해서는 재디자인을 필요로 하는 구현 디스토션이 높다. 또한 현재 생산되고 있는 펜티움, 울트라스팍등의 고속 버스 시스템에서 동작하기 위해서는 전체 시스템의 재디자인이 필요하다[4][6][12].

본 논문에서는 BACH가 가지는 구현 디스토션을 최소화하고 25Mhz이상의 버스 속도를 지원하는 고속 하드웨어 모니터링 시스템을 설계하였다. 트레이스 보드는 재설정가능하도록 EPLD(Erasable Programmable Logic device)를 이용하여 구성하였으며 트레이스 되는 시스템의 구성이 바뀌어 디코딩할 콘트롤과 트레이스 받는 정보의 종류를 HDL(Hardware Description Language)의 재프로그래밍을 통해 쉽게 반영할 수 있다[3][9][11].

최근 시스템의 버스 속도가 33Mhz이하에서 66Mhz까지 고속화하고 있는 점을 반영하여, 트레이스 생성 시스템은 최대 66Mhz의 고속 버스 시스템에서 동작 가능하도록 설계하였으며 초당 6.6Gb의 트레이스를 생성할 수 있다. 현재는 66Mhz를 지원하는 EPLD와 66Mhz 동기식 SRAM을 이용하여 트레이스 시스템을 설계하였기 때문에 66Mhz이상의 버스 속도를 가지는 시스템을 트레이스할 수 없으나 EPLD와 동기식 SRAM을 66Mhz이상의 속도를 지원하는 것으로 교체하면 66Mhz이상의 버스 속도를 가지는 시스템에서도 트레이스가 가능하다.

5. 결 론

본 연구에서는 66Mhz 고속에서 동작하며 초당 6.6Gb의 트레이스를 처리할 수 있으며, 기존의 하드웨어 모니터링 기법이 가지는 장점과 함께, 하드웨어 모니터링 기법의 단점인 구현 디스토션과 스타트업 디스토션을 완화시킨 재구성이 용이한 트레이스 생성 시스템(RTGS)을 개발하였다. 기존의 트레이스 생성 시스템에 대한 연구를 통하여 각 트레이스 생성 시스템의 장, 단점을 제시하였으며, 이러한 기존의 기법들의 단점을 보완한 하드웨어 모니터링 시스템을 설계하였다.

설계된 RTGS는 기존의 하드웨어 트레이싱 시스템에서는 불가능한 고속의 시스템에서도 트레이스 생성이 가능하며, 제어회로 등을 재설정가능하고 재프로그래밍 가능한 EPLD(Erasable Programmable Logic device)로 구성하였으므로 다양한 시스템에서 쉽게 적용이 가능하다는 장점을 갖는다.

참 고 문 헌

[1] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz, "ATUM : A New Technique for Capturing Address Traces Using Microcode," Proc. of the 13th Annual International Symposium on Computer Architecture, pp. 119-127, June 1986.

[2] Nikitas Alexandridis, "Design of Microprocessor Based Systems," Prentice-Hall, 1993.

[3] Stephen Brown, Jonathan Rose, "FPGA and CPLD Architectures : A Tutorial," IEEE Design & Test of Computers, 1996.

[4] J. K. Flanagan, B. Nelson, J. Archibald, and K. Grimsrud, "BACH:BYU Address Collection Hardware ; The Collection of Complete Traces," Proc. of the 6th Int. Conf. on Modeling Techniques and Tools for Computer Performance Evaluation, pp. 51-65, Sep. 1992.

[5] Intel Corporation, "Microprocessors Vol. I," Intel Corporation, 1993.

[6] K. Grimsrud, J. Archibald, M. Ripley, K. Flanagan, and B. Nelson, "BACH : A Hardware Monitor for Tracing Microprocessor-Based Systems," Microprocessors and Microsystems, Vol. 17, No. 8, pp. 443-455, Oct. 1993.

[7] Joseph D. Greenfield, "Practice Digital Design using ICs," Prentice-Hall, 1994.

[8] Walter A. Triebel, "The 80386DX Microprocessor Hardware, Software, & Interfacing," Prentice-Hall, 1992.

[9] Altera Corporation, "MAX+PLUS II AHDL Version 6.0," Altera Corporation, Nov. 1995.

[10] John F. Wakerly, "Digital Design Principles and

Practices," 2nd Ed., Prentice-Hall, 1994.

[11] Altera Corporation, "Data Book," Altera Corporation, Jun. 1996.

[12] J. K. Flanagan, B. Nelson, J. Archibald, and K. Grimsrud, "Incomplete Trace Data and Trace Driven Simulation," Proc. of the International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, pp. 203-209, 1993.

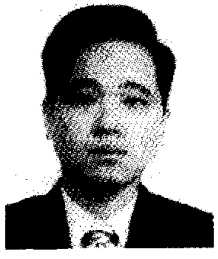
[13] Wen-Hann Wang, Jean-Loup Bear, "Efficient Trace-Driven Simulation Methods for Cache Performance Analysis," ACM trans. on Computer Systems, pp. 222-241, Vol. 9, No. 3, Aug. 1991.

[14] J. Bradley Chen, "Software Methods for System Address Tracing," Technical Report CMU-CS-93-188, Carnegie Mellon Univ., Aug. 1993.

[15] Samsung Corporation., "SRAM Data Book," Samsung Corp., 1996.

[16] David Nagle, Richard Uhlig, Trevor Mudge, "Monster: A Tool for Analyzing the Interaction Between Operating Systems and Computer Architectures," Technical Report, Univ. of Michigan, May 1992.

[17] Anant Agarwal, "Analysis of Cache Performance for Operating Systems and Multiprogramming," Kluwer Academic Publichers, pp. 1-30, 1989.

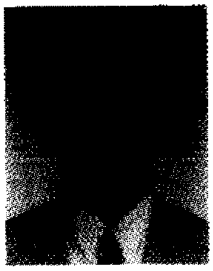


윤 형 민

1995년 연세대학교 지질학과(학사)

1998년 연세대학교 컴퓨터과학과(석사)

관심분야: HCI, Digital 회로 설계 및 FPGA 설계, 컴퓨터 구조 설계



박 기 호

1993년 연세대학교 전산과학과(학사)

1995년 연세대학교 대학원 전산과학과(석사)

1995년~현재 연세대학교 대학원 컴퓨터과학과 박사과정

관심분야: 컴퓨터 구조, 병렬처리 시스템, 고성능 메모리 시스템 구조

이 길 환

1996년 연세대학교 컴퓨터과학과(학사)

1996년~현재 연세대학교 대학원 컴퓨터과학과 석사과정

관심분야: 컴퓨터 구조, 고성능 메모리 시스템 구조

한 탁 돈

1978년 연세대학교 공과대학 전자공학과(학사)

1983년 Wayne State University 컴퓨터공학(석사)

1987년 University of Massachusetts 컴퓨터공학(박사)

1987년~1989년 Cleveland 주립대학 조교수

1989년~현재 연세대학교 공과대학 컴퓨터과학과 부교수

관심분야: 병렬처리 컴퓨터 구조 및 알고리즘, 오류보정 시스템, VLSI 설계, HCI

김 신 덕

1982년 연세대학교 공과대학 전자공학과(학사)

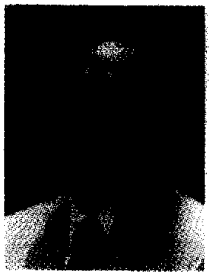
1987년 University of Oklahoma 전기공학(석사)

1991년 Purdue University 전기공학(박사)

1993년 2월~1995년 2월 광운대학교 컴퓨터공학과 조교수

1995년 3월~현재 연세대학교 공과대학 컴퓨터과학과 부교수

관심분야: 병렬처리 시스템, 컴퓨터 구조, Heterogeneous Computing



양 성 봉

1981년 연세대학교 공과대학 요업공학과 졸업(학사)
1984년 University of Oklahoma 컴퓨터과학(학부과정)
1986년 University of Oklahoma 컴퓨터과학(석사)
1992년 University of Oklahoma 컴퓨터과학(박사)
1992년 8월~1992년 12월 University of Oklahoma Adjunct Assistant Professor
1993년 3월~1993년 8월 전주대학교 전자계산학과 전임강사
1994년 9월~현재 연세대학교 공과대학 컴퓨터과학과 조교수
관심분야: 병렬 알고리즘, Genetic 알고리즘

이 용 석

1973년 연세대학교 전기공학과 졸업(학사)
1977년 University of Michigan, An Arbor 전자공학과 (석사)
1981년 University of Michigan, An Arbor 전자공학과 (박사)
현재 연세대학교 전자공학과 교수
관심분야: 마이크로프로세서, SRAM, 캐쉬 및 DSP 설계