(19) 대한민국특허청(KR)
(12) 공개특허공보(A)

(11) 공개번호 10-2009-0119032
(43) 공개일자 2009년11월19일

(51) Int. Cl.
G06F 9/46 (2006.01) *G06F 9/06* (2006.01)
(21) 출원번호 10-2008-0044831
(22) 출원일자 2008년05월15일
심사청구일자 2008년05월15일

(71) 출원인
재단법인서울대학교산학협력재단
서울특별시 관악구 봉천7동 산4의 2번지
(72) 발명자
채수익
서울 송파구 방이동 89번지 올림픽선수기자촌 아파트 250동 103호
박상규
경기도 화성시 반월동 신영통현대홈타운 212동 1503호
(74) 대리인
특허법인신세기

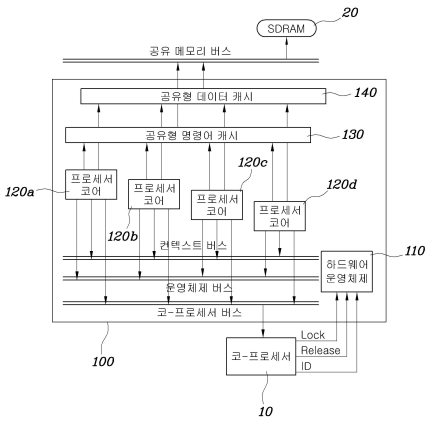
전체 청구항 수 : 총 19 항

(54) 멀티 프로세서 시스템

(57) 요 약

하드웨어 운영체제 기반 및/또는 공유형 캐시 기반의 멀티 프로세서 시스템이 개시된다. 복수의 프로세서 코어를 포함하는 멀티 프로세서 시스템에 있어서, 상기 복수의 프로세서 코어와 연결되어 상기 복수의 프로세서 코어로부터의 운영체제 서비스 요청에 대해서 인터럽트 핸들링, 작업 동기화, 작업 스케쥴링, 컨텍스트 교환 및 작업 재개 동작을 하드웨어적으로 수행하는 하드웨어 운영체제; 상기 복수의 프로세서 코어를 지원하는 명령어들이 미리 저장되는 공유형 명령어 캐시; 및 상기 복수의 프로세서 코어를 지원하는 데이터들이 미리 저장되는 공유형 데이터 캐시를 포함하는 멀티 프로세서 시스템이 제공된다. 기존 방법에 비하여 다중 프로세서 코어를 운용할 때 발생하는 성능 지연 및 오버헤드가 적고, 경량화된 프로세서를 사용하고 면적 면에서 효율적이어서 모바일용 멀티미디어 시스템에 최적으로 적용될 수 있다.

대 표 도 - 도1



특허청구의 범위

청구항 1

복수의 프로세서 코어; 및

상기 복수의 프로세서 코어와 운영체제 버스를 통해 연결되며, 상기 복수의 프로세서 코어로부터의 운영체제 서비스 요청에 대해서 인터럽트 핸들링, 작업 동기화, 작업 스케쥴링, 컨텍스트 교환 및 작업 재개 동작을 하드웨어적으로 수행하는 하드웨어 운영체제를 포함하는 멀티 프로세서 시스템.

청구항 2

제1항에 있어서,

상기 하드웨어 운영체제는,

작업 상태, 우선순위, 중단 큐(Wait Queue), 세마포어 상태 및 이들의 결합 중 하나인 작업 정보를 저장하는 쓰레드 제어 메모리;

대기 큐에 등록된 작업 중 가장 높은 우선순위의 작업을 선택하고 세마포어 접근을 제어하는 쓰레드 관리자;

생성된 작업의 컨텍스트가 저장되는 컨텍스트 메모리;

수행을 재개할 작업의 컨텍스트 또는 수행이 중단된 작업의 컨텍스트가 저장되는 컨텍스트 버퍼;

상기 쓰레드 관리자에 의해 선택된 작업에 대한 컨텍스트를 상기 컨텍스트 메모리로부터 읽어와 상기 컨텍스트 버퍼에 저장하거나 수행이 중단되어 상기 컨텍스트 버퍼에 저장된 작업의 컨텍스트를 상기 컨텍스트 메모리에 저장하는 컨텍스트 관리자; 및

상기 운영체제 버스에 연결되며, 상기 프로세서 코어로부터의 상기 운영체제 서비스 요청에 따라 상기 쓰레드 관리자 및 상기 컨텍스트 관리자를 제어하는 주 제어기를 포함하는 것을 특징으로 하는 멀티 프로세서 시스템.

청구항 3

제2항에 있어서,

상기 쓰레드 제어 메모리는, 상기 작업이 대기 상태인 경우 제1 값을 가지고 상기 작업이 중단된 경우 제2 값을 가지는 대기 큐(Ready Queue)와, 상기 작업의 우선순위를 포함하는 작업 스케쥴링을 위한 정보와, 각 세마포어별 세마포어 상태 플래그와 세마포어 값을 포함하는 세마포어 접근 제어를 위한 정보와, 상기 세마포어에 대기하고 있는 작업들이 등록되는 중단 큐(Wait Queue)를 포함하는 것을 특징으로 하는 멀티 프로세서 시스템.

청구항 4

제3항에 있어서,

상기 쓰레드 제어 메모리는 쓰레드 디스크립터(Thread Descriptor)와 세마포어 디스크립터(Semaphore Descriptor)를 포함하되,

상기 쓰레드 디스크립터의 필드값과 상기 세마포어 디스크립터의 필드값으로부터 상기 작업 스케쥴링을 위한 정보, 상기 세마포어 접근 제어를 위한 정보 및 상기 중단 큐의 획득이 가능한 것을 특징으로 하는 멀티 프로세서 시스템.

청구항 5

제4항에 있어서,

상기 작업 스케쥴링을 위한 정보는 상기 쓰레드 디스크립터의 필드 중 상기 우선순위의 상위 비트 및 하위 비트를 나타내는 2개의 필드에 저장되고,

상기 세마포어 접근 제어를 위한 정보는 상기 세마포어 디스크립터의 필드 중 상기 세마포어 디스크립터의 사용 여부를 나타내는 플래그, 상기 세마포어의 종류를 나타내는 플래그 및 상기 세마포어 값이 저장되는 3개의 필드에 각각 저장되는 것을 특징으로 하는 멀티 프로세서 시스템.

청구항 6

제5항에 있어서,

상기 중단 큐는 상기 세마포어의 디스크립터 중 나머지 1개의 필드와, 상기 쓰레드 디스크립터 중 나머지 2개의 필드를 이용하여 구현되는 것을 특징으로 하는 멀티 프로세서 시스템.

청구항 7

제6항에 있어서,

상기 세마포어 디스크립터의 나머지 1개의 필드가 음수인 경우 한 개 이상의 작업이 상기 중단 큐에 등록되어 있음을 나타내며, 상기 쓰레드 디스크립터의 나머지 2개의 필드 중 제1 필드에는 첫 번째 작업의 ID가 저장되고,

제2 필드가 제1 값인 경우 타 작업이 추가적으로 상기 중단 큐에 등록되어 있음을 나타내고, 후속 작업의 ID가 상기 제1 필드에 저장되어 있고,

상기 제2 필드가 제2 값인 경우 상기 작업이 상기 중단 큐에 등록된 마지막 작업임을 나타내는 것을 특징으로 하는 멀티 프로세서 시스템.

청구항 8

복수의 프로세서 코어를 포함하는 멀티 프로세서 시스템에 있어서,

상기 복수의 프로세서 코어와 연결되어 상기 복수의 프로세서 코어로부터의 운영체제 서비스 요청에 대해서 인터럽트 핸들링, 작업 동기화, 작업 스케쥴링, 컨텍스트 교환 및 작업 재개 동작을 수행하는 운영체제;

상기 복수의 프로세서 코어를 지원하는 명령어들이 미리 저장되는 공유형 명령어 캐시; 및

상기 복수의 프로세서 코어를 지원하는 데이터들이 미리 저장되는 공유형 데이터 캐시를 포함하는 멀티 프로세서 시스템.

청구항 9

제8항에 있어서,

상기 공유형 명령어 캐시는,

내부 통신 네트워크 및 복수의 온 칩 메모리를 포함하되, 하나의 명령어 블록을 구성하는 워드들이 서로 다른 온 칩 메모리에 분산 저장되고, 상기 분산 저장된 워드들을 동시에 읽을 수 있는 인터리브 명령어 캐시 메모리(Interleaved Instruction Cache Memory);

상기 인터리브 명령어 캐시 메모리에 저장된 명령어 블록의 일부분을 태그로 저장하고 있는 태그 메모리;

상기 운영체제 서비스 요청에 따른 명령어의 일부분과 상기 태그 메모리에 저장된 태그를 비교하여 상기 요청된 명령어가 상기 인터리브 명령어 캐시 메모리에 존재하는지를 확인하는 태그 매칭(Tag Matching)을 수행하고, 상기 태그 매칭 결과 상기 요청된 명령어가 상기 인터리브 명령어 캐시 메모리에 존재하지 않는 경우 상기 명령어 블록 요청 신호를 출력하는 캐시 제어기; 및

상기 명령어 블록 요청 신호에 따라 시스템 버스를 통해 외부 메모리로부터 명령어 블록을 읽어와 상기 인터리브 명령어 캐시 메모리에 저장하는 블록-필 제어기를 포함하는 것을 특징으로 하는 멀티 프로세서 시스템.

청구항 10

제9항에 있어서,

상기 인터리브 명령어 캐시 메모리는 서로 다른 경로(way)에 대해서 상기 복수의 온 칩 메모리에 분산 저장되는 순서가 다른 것을 특징으로 하는 멀티 프로세서 시스템.

청구항 11

제9항에 있어서,

상기 공유형 명령어 캐시는 명령어 블록에 처음 접근하려는 경우에 상기 태그 매칭을 수행하고, 상기 명령어 블록에 속한 명령어들을 순차적으로 읽어오는 경우에는 상기 태그 매칭을 건너뛰는 태그 매칭 스키핑(Tag Matching Skipping) 방식을 이용하는 것을 특징으로 하는 멀티 프로세서 시스템.

청구항 12

제11항에 있어서,

상기 공유형 명령어 캐시는 상기 복수의 프로세서 코어 중 하나가 분기 명령을 수행한 경우, 상기 프로세서 코어가 순차적인 주소를 가진 명령어를 요청했으나 상기 명령어가 타 명령어 블록에 저장된 경우 상기 태그 매칭을 수행하는 것을 특징으로 하는 멀티 프로세서 시스템.

청구항 13

제9항에 있어서,

상기 공유형 명령어 캐시는, 상기 복수의 프로세서 코어마다 할당되고, 상기 할당된 프로세서 코어가 명령어를 요청하기 전에 미리 상기 명령어를 상기 인터리브 명령어 캐시로부터 읽어와 저장하고 있는 명령어 버퍼를 더 포함하는 것을 특징으로 하는 멀티 프로세서 시스템.

청구항 14

제13항에 있어서,

상기 명령어 버퍼는 상기 할당된 프로세서 코어가 분기 명령을 수행하였는지 여부에 대한 분기 상태 신호(Branch Status Signal)와 상기 명령어 버퍼에 저장된 명령어들의 개수에 대한 프리-펫치 압력 신호(Pre-fetch Pressure Signal)를 상기 캐시 제어기에 전달하고,

상기 캐시 제어기는 상기 분기 상태 신호와 상기 프리-펫치 압력 신호들을 이용해 읽고자 하는 명령어를 결정한 후 상기 내부 통신 네트워크를 통해 상기 복수의 온 칩 메모리로 온 칩 메모리 선택 신호를 출력하여 상기 명령어가 상기 명령어 버퍼에 삽입되도록 하는 것을 특징으로 하는 멀티 프로세서 시스템.

청구항 15

제8항에 있어서,

상기 공유형 데이터 캐시는,

내부 통신 네트워크 및 복수의 온 칩 메모리를 포함하되, 하나의 데이터 블록을 구성하는 워드들이 서로 다른 온 칩 메모리에 분산 저장되고, 상기 분산 저장된 워드들을 동시에 읽을 수 있는 인터리브 데이터 캐시 메모리(Interleaved Data Cache Memory);

상기 인터리브 데이터 캐시 메모리에 저장된 데이터 블록의 일부분을 태그로 저장하고 있는 태그 메모리;

상기 운영체제 서비스 요청에 따른 데이터의 일부분과 상기 태그 메모리에 저장된 태그를 비교하여 상기 요청된 데이터가 상기 인터리브 데이터 캐시 메모리에 존재하는지를 확인하는 태그 매칭을 수행하고, 상기 태그 매칭 결과 상기 요청된 데이터가 상기 인터리브 데이터 캐시 메모리에 존재하지 않는 경우 상기 데이터 블록 요청 신호를 출력하는 캐시 제어기; 및

상기 데이터 블록 요청 신호에 따라 시스템 버스를 통해 외부 메모리로부터 데이터 블록을 읽어와 상기 인터리브 데이터 캐시 메모리에 저장하는 블록-필 제어기를 포함하는 것을 특징으로 하는 멀티 프로세서 시스템.

청구항 16

제15항에 있어서,

상기 공유형 데이터 캐시는,

소정 시간 내에 접근하였던 온 칩 메모리의 주소들의 태그를 저장하는 히스토리 버퍼; 및

상기 히스토리 버퍼에 저장된 태그와 요청된 주소를 비교하여 상기 요청된 데이터가 상기 인터리브 데이터 캐시 메모리에 존재하는지를 판별하는 히스토리 테스터를 더 포함하는 것을 특징으로 하는 멀티 프로세서 시스템.

청구항 17

제16항에 있어서,

상기 캐시 제어기는 상기 히스토리 테스터에 의해 판별이 끝나지 않는 경우 상기 태그 메모리로부터 상기 태그를 읽어와 상기 태그 매칭을 수행하는 것을 특징으로 하는 멀티 프로세서 시스템.

청구항 18

제16항에 있어서,

상기 히스토리 버퍼는 스택 메모리 접근과 힙 메모리 접근을 구별하여 상기 태그를 저장하는 것을 특징으로 하는 멀티 프로세서 시스템.

청구항 19

복수의 프로세서 코어를 포함하는 멀티 프로세서 시스템에 있어서,

상기 복수의 프로세서 코어와 연결되어 상기 복수의 프로세서 코어로부터의 운영체제 서비스 요청에 대해서 인터럽트 핸들링, 작업 동기화, 작업 스케쥴링, 컨텍스트 교환 및 작업 재개 동작을 하드웨어적으로 수행하는 하드웨어 운영체제;

상기 복수의 프로세서 코어를 지원하는 명령어들이 미리 저장되는 공유형 명령어 캐시; 및

상기 복수의 프로세서 코어를 지원하는 데이터들이 미리 저장되는 공유형 데이터 캐시를 포함하는 멀티 프로세서 시스템.

명 세 서

발명의 상세한 설명

기 술 분 야

<1> 본 발명은 멀티 프로세서 시스템(Multi-processor system)에 관한 것으로, 보다 상세하게는 하드웨어 운영체제 기반 및/또는 공유형 캐시 기반의 멀티 프로세서 시스템에 관한 것이다.

배 경 기 술

<2> 최근 휴대폰, PDA, 전자 사전 등의 디지털 정보화 기기 뿐만 아니라 TV, 냉장고 등의 백색 가전도 인공 지능화 및 정보화가 빠르게 이루어지고 있다. 이에 따라, 전자 산업 전반에서 소프트웨어의 비중이 급격히 증가하고 있으며, 이에 따라 소프트웨어를 실행하기 위한 프로세서에 대한 수요도 증가하고 있다. 디지털 정보화 기기에 내장되는 프로세서를 내장형 프로세서(Embedded Processor)라고 부르고, 내장형 프로세서에서 실행되는 소프트웨어를 내장형 소프트웨어(Embedded Software)라고 부른다.

<3> 일반적으로 디지털 정보화 기기들은 다양한 외부 자극에 빠르게 반응하여, 실시간 작업들(Real-time Task)을 동시에 실행해야 하기 때문에, 내장형 소프트웨어는 다수의 쓰레드로 구성된 멀티 쓰레드 프로그램(Multi-threaded Program)으로 구현된다.

<4> 일반적으로 내장형 프로세서가 멀티 쓰레드 프로그램을 수행하는 과정은 인터럽트 핸들링(Interrupt Handling), 작업 동기화(Job/Task Synchronization), 작업 스케쥴링(Task Scheduling), 컨텍스트 교환(Context Switching), 작업 재개(Task Resume)의 다섯 가지 세부 단계로 구성된다. 각 세부 단계를 좀 더 상세하게 설명하면 다음과 같다.

<5> (1) 인터럽트 핸들링 단계에서 타이머 등의 외부 자극(인터럽트)이 프로세서에 가해지면, 프로세서는 외부 자극을 발생시킨 하드웨어 장치가 무엇인지 알아내고, 이에 대응되는 인터럽트 핸들러 루틴을 호출한다. (2) 작업 동기화는 인터럽트 핸들러 루틴이 하드웨어 장치가 요청한 내용에 따라 적절한 작업을 수행한 후, 세마포어, 배리어 등 동기화 관련 변수 (variable)들의 값을 변경함으로써 중지되었던 응용 작업을 깨워 대기 큐(Ready Queue)에 삽입하는 것을 말한다. (3) 작업 스케쥴링은 대기 큐에 저장된 작업 중에서 가장 우선 순위가 높은 작업을 선택하는 것을 말한다. (4) 컨텍스트 교환은 내장형 프로세서에서 수행중인 작업에 관한 각종 정보들을 지

정된 메모리 영역에 대피시키고, 작업 스케쥴링 단계에서 선택된 작업에 관한 정보들을 특정 메모리 영역으로부터 읽어와 내장형 프로세서의 내부 레지스터들에 설정하는 것을 말한다. (5) 작업 재개는 상기 과정을 통하여 프로세서에 설정된 작업의 수행을 개시하는 것을 말한다.

<6> 일반적인 시스템에서 상기 과정들은 소프트웨어적으로 수행되는데, 상기 과정들을 담당하는 소프트웨어를 운영체제라 한다. 상기 과정들을 소프트웨어적으로 수행하기 위해서는 프로세서의 연산 자원(Computing Power)을 소비해야 한다. 운영체제 수행을 위한 연산 자원의 양을 멀티 쓰레딩 오버헤드라 한다. 앞에서 설명한 다섯 단계를 수행하기 위해 필요한 연산 자원은 운영체제의 알고리즘과 복잡도에 따라 다르지만, 수 백 싸이클에서 수 천 싸이클 정도이다. 이 정도의 오버헤드는 쓰레드 간 전환이 수 만 싸이클 이상의 단위로 이루어지는 소프트웨어의 경우에는 크게 문제되지 않는다. 예를 들어, 개인용 PC에서 많이 사용하는 윈도즈나 리눅스의 경우 멀티 쓰레딩 오버헤드는 전체 연산 자원의 5%를 넘지 않는다.

<7> 한편, 최근들어 개발되는 모바일 멀티미디어 시스템들은 여러 개의 프로세서들이 협업하여 하나의 응용을 실행하는 멀티 프로세서 시스템 구조를 채택하고 있고, 영상. 음성, 3D 그래픽스 등의 가속을 위한 하드웨어 가속기를 내장하고 있다. 이러한 응용에서는 성능을 극대화하기 위하여 쓰레드 전환이 매우 빈번하게 이루어지고 있다. 예를 들어, 하드웨어 가속기와 소프트웨어가 협업하는 동영상 코덱의 경우, 약 1천 ~ 5천 싸이클마다 쓰레드 전환이 발생하는데, 이런 경우 멀티 쓰레딩 오버헤드의 비중이 매우 커지게 된다. 예를 들어, 앞에서 설명한 다섯 단계를 수행하기 위해 필요한 연산 자원이 약 1천 싸이클이라고 할 때 쓰레드 전환이 3천 싸이클마다 발생한다면, 멀티 쓰레딩 오버헤드는 약 25%이다. 이는 한 시스템이 4개의 내장형 프로세서를 가지고 있는 경우를 가정할 때, 내장형 프로세서 중 하나는 멀티 쓰레딩 오버헤드를 위해 허비되는 것과 같다. 이 정도의 멀티 쓰레딩 오버헤드를 감수하는 것은 비실용적이므로, 일반적으로는 쓰레드 전환이 드물게 발생하도록 내장형 소프트웨어를 작성한다. 하지만, 이 경우 외부 자극이나 하드웨어 가속기의 요청에 기민하게 대응할 수 없기 때문에 성능 면에서 한계가 있다.

<8> 멀티 쓰레딩 오버헤드의 근본적인 원인은 상술한 인터럽트 핸들링, 작업 동기화, 작업 스케쥴링, 컨텍스트 교환, 작업 재개의 과정들이 소프트웨어적으로 수행되기 때문이다. 멀티 쓰레딩을 위한 다섯 가지 동작들을 하드웨어적으로 수행한다면, 멀티 쓰레딩 오버헤드를 최소화할 수 있다. 사실, 소프트웨어 작업의 일부를 하드웨어 가속기로 구현함으로써 성능을 개선한다는 것은 보편적인 전략으로, 멀티 쓰레딩 오버헤드를 줄이기 위하여 많은 프로세서들이 발표되었다. 하지만, 지금까지 발표된 프로세서 구조들은 단 하나의 프로세서 코어만을 지원하는 솔루션이거나 상술한 다섯 가지 단계 중에서 일부만을 하드웨어로 구현한 솔루션이다. 기존의 방법들의 장단점은 다음과 같다.

<9> 한 쓰레드의 임의의 순간의 상태(State)를 규정할 수 있는 정보의 집합을 해당 쓰레드의 컨텍스트라고 한다. 일반적인 RISC(Reduced Instruction Set Computer) 구조의 프로세서는 내부적으로 단 하나의 컨텍스트만을 저장할 수 있는 레지스터 세트를 가지고 있다. 이에 반하여, MIPS(Microprocessor without Interlocked Pipeline Stages) 32K, SPARC(Scalable Processor Architecture) V9 등의 프로세서들은 복수의 컨텍스트를 저장할 수 있도록 여분의 레지스터 세트를 가지고 있기 때문에, 매 싸이클마다 서로 다른 쓰레드를 실행할 수 있다.

<10> 이러한 멀티 컨텍스트 멀티 쓰레딩 프로세서(Multiple Context Multithreading Processor)는 컨텍스트 교환과 쓰레드의 선택이 하드웨어적으로 이루어지기 때문에 작업 스케쥴링 오버헤드와 컨텍스트 교환 오버헤드를 줄일 수 있다. 하지만, 일반적으로 멀티 컨텍스트 멀티 쓰레딩 프로세서들은 4개 전후의 컨텍스트 세트를 제공하기 때문에 전체 작업의 개수가 하드웨어적으로 지원하는 컨텍스트 세트의 개수보다 큰 경우에는 소프트웨어 운영체제의 개입이 필요하므로 멀티 쓰레딩 오버헤드가 증가하게 된다. 또한, 시중에 출시된 멀티 컨텍스트 멀티 쓰레딩 프로세서들은 인터럽트 핸들링, 작업 동기화 등에 관해서는 고려하지 않기 때문에, 외부 자극 및 하드웨어 가속기로부터의 요청이 빈번한 시스템에서는 멀티 쓰레딩 오버헤드가 여전히 상당하다. 한편, 멀티 컨텍스트 멀티 쓰레딩 프로세서는 컨텍스트는 복수개를 가지고 있지만 정작 연산을 수행할 데이터패스는 단 하나만을 가지고 있으므로, 멀티 쓰레딩 오버헤드를 줄이는 것 이외에는 성능 효과를 기대할 수가 없다. 더구나, 경량 프로세서의 경우 데이터패스가 차지하는 면적이 대략 20K ~ 30K 게이트 정도임에 반하여, 하나의 컨텍스트를 위한 레지스터 세트가 차지하는 면적은 대략 10K 게이트 정도이고 4개의 컨텍스트를 위한 면적은 대략 40K 게이트 정도로 공유할 자원인 데이터 패스보다 공유할 수 없는 자원인 레지스터 세트가 훨씬 크기 때문에 면적면에서 비효율적이다. 예를 들어, 4개의 컨텍스트를 지원하는 멀티 컨텍스트 멀티 쓰레딩 프로세서의 면적은 약 60K 게이트임에 반하여, 하나의 컨텍스트를 지원하는 RISC 프로세서의 면적은 약 30K 게이트에 불과하다. 즉, 같은 면적을 사용하는 경우에는, 두 개의 프로세서를 병렬로 사용할 수 있는 RISC 프로세서가 오히려 성능 면에서 유리하다.

<11> 동시적 멀티 쓰레딩(Simultaneous Multithreading; 이하 SMT라 칭함) 프로세서는 복수의 컨텍스트를 지원함과 동시에 복수의 데이터 패스를 지원하는 구조의 프로세서이다. SMT 프로세서는 멀티 쓰레딩 오버헤드를 줄이는 것 뿐만 아니라, 하나의 쓰레드의 복수의 명령어를 동시에 수행하거나, 여러 쓰레드의 명령어들을 동시에 수행함으로써 성능을 대폭 향상시킬 수 있는 구조이다. 일반인에게 널리 알려져 있는 인텔 펜티엄4 프로세서에 적용된 하이퍼 쓰레딩(Hyper-threading) 기술이 바로 SMT이다. SMT 프로세서의 단점은 면적이 지나치게 크다는 점이다. 일반적으로 SMT 프로세서는 내부적으로 대용량의 명령어 큐를 가지고 있고, 레지스터 재명명(Register renaming)과 예약 테이블(Reservation Table) 등의 복잡한 회로를 가지고 있다. 3개의 명령어를 동시에 수행할 수 있는 SMT 프로세서의 경우, 데이터 패스의 면적은 약 70K 게이트 수준임에 반하여, 이를 운용하기 위하여 부가적으로 필요한 회로의 면적은 120K 게이트를 넘는다. 한편, 멀티 컨텍스트 멀티 쓰레딩 프로세서와 마찬가지로 SMT 프로세서도 인터럽트 핸들링과 작업 동기화에 관한 지원이 없고, 시스템 내의 작업의 개수가 하드웨어 컨텍스트의 개수를 초과하는 경우 소프트웨어 운영체제의 개입이 필요하기 때문에, 역시 멀티 쓰레딩 오버헤드를 줄이는데 있어 한계가 있다.

<12> 매우 많은 작업을 효과적으로 지원하기 위한 방법으로는 컨텍스트 교환을 하드웨어적으로 수행하는 것과 작업 스케쥴링을 하드웨어적으로 수행하는 것이 있다.

<13> 먼저, 컨텍스트 교환에 관하여 살펴보겠다. 컨텍스트 교환을 하드웨어적으로 수행하는 프로세서로는 인텔 사의 펜티엄 프로세서가 가장 대표적이다. 펜티엄 프로세서에서 각 쓰레드는 TSS(Task State Segment)라는 메모리 영역을 할당받아 가지고 있는데, 이 쓰레드의 컨텍스트 정보들이 이 TSS에 저장되며, 특정 TSS를 가리키는 ID를 인수로 받는 특별한 호출(call) 명령어를 이용해 컨텍스트를 교환할 수 있다. 인터럽트가 발생한 경우에는, 프로세서 레지스터 값들이 현재 실행중인 쓰레드의 TSS에 자동으로 대피하고, 인터럽트를 발생한 하드웨어에 대응되는 핸들러 루틴의 컨텍스트를 자동적으로 로드하고 실행한다. 펜티엄 프로세서는 컨텍스트 교환과 인터럽트 핸들링을 하드웨어적으로 처리함으로써 멀티 쓰레딩 오버헤드를 감소시켰으나, 작업 스케쥴링과 작업 동기화는 여전히 소프트웨어적으로 수행해야 한다는 한계가 있다. 더욱이, 작업 스케쥴링을 통해 재개할 쓰레드가 결정되었으나, 이 쓰레드의 컨텍스트가 외부 메모리에 대피되어 있는 경우, 외부 메모리로부터 컨텍스트를 읽어오는 동안 프로세서는 수행이 정지된다는 문제점이 있다. 예를 들어, 컨텍스트가 32개의 워드로 구성되어 있고, 프로세서는 800 MHz로, 외부 메모리는 200 MHz로 동작하는 경우, 최소한 128 (프로세서) 싸이클 동안 프로세서는 수행이 중단된다.

<14> 작업 스케쥴링 방식은 운영체제의 구현에 따라 혹은 작업의 동적 특성에 따라 적응적으로 선택해야 하기 때문에, 개인용 컴퓨터와 같이 다양한 응용 소프트웨어를 수행하는 시스템에서는 소프트웨어적으로 처리하는 것이 필수일 수 있으나, 모바일 멀티미디어 핸드폰이나 PDA와 같은 내장형 시스템에서는 특정 스케쥴링 방식을 하드웨어적으로 구현하더라도 큰 문제가 없다. 하지만, 일부 발표된 작업 스케쥴링 방식을 하드웨어적으로 수행하는 하드웨어 가속기들은 운영체제의 핵심 기능들은 여전히 소프트웨어적으로 구현되고, 극히 일부 기능만 하드웨어 가속기를 통해 수행하기 때문에, 멀티 쓰레딩 오버헤드 감소 효과가 크지 않고, 프로세서와의 통신 및 컨텍스트 교환을 위해 많은 시간이 소모되어 스케쥴링을 하드웨어적으로 수행하는 효과를 반감시킨다. 더욱이, 일부는 너무 많은 면적을 차지하기 때문에 실용성이 없다.

<15> 최근의 시장 요구에 따라 소프트웨어의 복잡도는 지속적으로 증가하고 있으므로, 더 이상 하나의 내장형 프로세서로는 주어진 성능을 만족할 수 없다. 이 때문에, 하나의 시스템에 점점 더 많은 프로세서들이 내장되는 추세이다. 멀티 프로세서 상에서 소프트웨어를 수행할 때에는 각 프로세서마다 독립적인 소프트웨어를 수행하는 경우도 있지만, 이보다는 소프트웨어를 복수개의 작업으로 나누어 구현하고, 각 쓰레드들을 다수의 프로세서 상에 분산시켜 병렬로 수행시킬 수 있도록 멀티 쓰레드 프로그램으로 구현하는 것이 일반적이다.

<16> 멀티 쓰레드 프로그램을 멀티 프로세서 시스템 상에서 수행하는 경우, 앞에서 설명한 멀티 쓰레딩 오버헤드 이외에 작업 이전 오버헤드(Task Migration Overhead)가 추가로 발생한다. 작업 이전 오버헤드란, 임의의 한 프로세서 상에서 수행되던 쓰레드를 중단한 후, 이 쓰레드를 다른 프로세서에서 재개할 때 추가로 발생하는 시간 지연을 말한다. 이를 구체적으로 설명하면 다음과 같다.

<17> 고성능 프로세서들은 외부 메모리의 통신 지연(Latency)으로 인한 성능 감소를 방지하기 위하여 명령어 캐시와 데이터 캐시를 가지고 있다. 명령어 혹은 데이터를 처음 읽는 경우에는, 어쩔 수 없이 외부 메모리로부터 정보를 가져오기 위하여 긴 시간을 기다려야 하지만, 그 후에 이 명령어 혹은 데이터가 다시 읽히는 경우에는 캐시가 직접 공급함으로써 프로세서의 성능을 개선할 수 있다. 데이터를 저장할 때에는 일단 데이터 캐시 상에만 저장하고, 외부 메모리에는 추후에 반영함으로써 프로세서의 성능을 개선할 수 있다. 한 프로세서에서 수행되다

중단되었던 작업을 다른 프로세서에서 수행하려는 경우, 이 작업을 수행하며 변경된 데이터들은 이전 프로세서의 캐시에는 저장되어 있으나 아직 외부 메모리에는 반영되지 않았을 수 있고, 이 경우, 새 프로세서에서는 올바른 데이터를 참조할 수가 없다. 따라서, 작업을 이전할 때에는 데이터의 일관성(Coherency)를 위하여 기존 프로세서의 데이터 캐시 상에만 변경된 데이터들을 외부 메모리에 강제로 반영시키는 데이터 캐시 플러시(Data Cache Flush)를 수행해야 하는데, 이 기간에는 프로세서가 다른 작업을 수행할 수 없기 때문에 장시간의 시간 지연이 불가피하다. 더욱이, 데이터 캐시는 데이터를 변경한 쓰레드가 무엇인지 알 수가 없으므로 부득이하게 모든 변경된 데이터를 외부 메모리에 반영해야 하므로, 시간 지연은 매우 심각하다고 할 수 있다. 한편, 명령어 캐시에 대해서도 이전 프로세서의 캐시에 해당 명령어가 이미 있음에도 불구하고 새 프로세서를 위해 외부 메모리로부터 명령어들을 다시 읽어 와야 하고, 시간 지연이 발생한다. 작업 이전 오버헤드는 위와 같이 작업을 이전할 때 발생하는 부가적인 시간 지연들을 말한다.

<18> 일부 시스템에서는 메모리의 용량은 작지만 고속으로 동작할 수 있는 캐시인 L1 캐시들은 각 프로세서마다 배치된다. L1 캐시와 외부 메모리 사이에서 프로세서와는 다소 거리를 두고 떨어져 배치되어 있으나 대용량의 캐시 메모리를 사용하는 캐시인 L2 캐시는 다수의 프로세서가 공유하는 계층적 구조를 채택함으로써 작업 이전 오버헤드를 줄이고 캐시 메모리의 면적 효율을 개선하고 있다. 하지만, L1 캐시들은 여전히 각 프로세서마다 분산 배치되어 있으므로 작업 이전 오버헤드를 완벽히 제거할 수가 없는 문제점이 있다.

<19> 한편, 각 프로세서마다 고유하게 명령어 캐시와 데이터 캐시를 가지고 있는 경우, 캐시 메모리들이 잘게 나누어져 분산된다. 특정 프로세서에 대한 작업의 할당이 동적으로 이루어지는 시스템의 경우, 모든 프로세서 별로 균등한 크기의 캐시 메모리를 할당하는 것이 합리적으로 보이지만, 작업별로 필요로 하는 캐시 메모리의 크기가 다르다는 점을 고려하면 비효율을 야기한다. 예를 들어, 4KB의 데이터 캐시를 가진 프로세서 P_A와 프로세서 P_B에 각각 2KB와 6KB의 메모리를 필요로 하는 작업 T_A와 T_B가 수행중이라고 가정해보자. 두 작업을 위해 필요한 메모리는 총 8KB이고, 데이터 캐시 메모리도 총 8KB으로 같지만, P_A의 경우에는 할당된 캐시 메모리의 반만을 활용하고, P_B의 경우에는 빈번하게 캐시 미스(Miss)가 발생하여, 시간 지연이 발생한다. 이는 캐시 메모리가 분할되어 분산되어 있기 때문에 발생하는 것으로, 이러한 효과를 캐시 메모리 단편화 효과(Cache Memory Fragmentation Effect)라 한다.

<20> 각 프로세서마다 고유한 명령어 캐시와 데이터 캐시를 부여하는 기존 방식은 작업 이전 오버헤드와 캐시 메모리 단편화 현상으로 인하여 높은 성능을 기대하기가 어렵다. 이에 대한 대안은 하나의 명령어 캐시와 하나의 데이터 캐시를 여러 개의 프로세서가 공유하는 것이나, 이 경우 복수개의 프로세서를 동시에 지원할 수 있는 독창적인 캐시 구조가 필요하다.

발명의 내용

해결 하고자하는 과제

<21> 따라서, 본 발명은 인터럽트 핸들링, 작업 동기화, 작업 스케쥴링, 컨텍스트 교환 등의 과정을 하드웨어적으로 구현한 하드웨어 운영체제를 통하여 통합적으로 수행함으로써, 상기 과정에 수반되는 오버헤드를 최소화할 수 있는 멀티 프로세서 시스템을 제공한다.

<22> 또한, 본 발명은 운영체제가 하드웨어적으로 구현되므로, 프로세서가 소프트웨어 운영체제를 지원하기 위한 기능을 제공할 필요가 없어 보다 경량화된 프로세서를 사용할 수 있는 멀티 프로세서 시스템을 제공한다.

<23> 또한, 본 발명은 복수의 프로세서 코어가 하나의 명령어 캐시와 데이터 캐시를 L1 캐시로서 공유할 수 있는 공유형 명령어 캐시 및 공유형 데이터 캐시를 도입함으로써 작업 이전 오버헤드와 캐시 메모리 단편화 현상을 제거함으로써 멀티 프로세서 시스템의 성능을 대폭 개선한 효과가 있는 멀티 프로세서 시스템을 제공한다.

<24> 또한, 본 발명은 공유형 캐시를 통해 좀 더 면적 효율이 좋은 대용량 온 칩 SRAM을 복수의 프로세서 코어가 공유할 수 있고, 각 프로세서 코어마다 독점적 캐시들이 분산된 경우와 비교할 때 동일한 성능을 내기 위해 보다 적은 캐시 메모리를 사용할 수 있으므로 면적 면에서도 효율적인 멀티 프로세서 시스템을 제공한다.

<25> 또한, 본 발명은 경량화된 프로세서를 사용하고 면적 면에서 효율적이어서 모바일용 멀티미디어 시스템에 최적으로 적용될 수 있는 멀티 프로세서 시스템을 제공한다.

과제 해결수단

<26> 본 발명의 일 측면에 따르면, 복수의 프로세서 코어; 및 상기 복수의 프로세서 코어와 운영체제 버스를 통해 연결되며, 상기 복수의 프로세서 코어로부터의 운영체제 서비스 요청에 대해서 인터럽트 핸들링, 작업 동기화, 작업 스케쥴링, 컨텍스트 교환 및 작업 재개 동작을 하드웨어적으로 수행하는 하드웨어 운영체제를 포함하는 멀티 프로세서 시스템이 제공된다.

<27> 상기 하드웨어 운영체제는, 작업 정보를 저장하는 쓰레드 제어 메모리; 대기 큐에 등록된 작업 중 가장 높은 우선순위의 작업을 선택하고 세마포어 접근을 제어하는 쓰레드 관리자; 생성된 쓰레드들의 컨텍스트가 저장되는 컨텍스트 메모리; 수행을 재개할 쓰레드의 컨텍스트 또는 수행이 중단된 쓰레드의 컨텍스트가 저장되는 컨텍스트 버퍼; 상기 쓰레드 관리자에 의해 선택된 쓰레드에 대한 컨텍스트를 상기 컨텍스트 메모리로부터 읽어와 상기 컨텍스트 버퍼에 저장하거나 수행이 중단되어 상기 컨텍스트 버퍼에 저장된 쓰레드의 컨텍스트를 상기 컨텍스트 메모리에 저장하는 컨텍스트 관리자; 및 상기 운영체제 버스에 연결되며, 상기 프로세서 코어로부터의 상기 운영체제 서비스 요청에 따라 상기 쓰레드 관리자 및 상기 컨텍스트 관리자를 제어하는 주 제어기를 포함할 수 있다.

<28> 여기서 상기 쓰레드 제어 메모리는, 상기 작업이 대기 상태인 경우 제1 값을 가지고 상기 작업이 중단된 경우 제2 값을 가지는 대기 큐(Ready Queue)와, 상기 작업의 우선순위를 포함하는 작업 스케쥴링을 위한 정보와, 각 세마포어 별 세마포어 상태 플래그와 세마포어 값을 포함하는 세마포어 접근 제어를 위한 정보와, 상기 세마포어에 대기하고 있는 작업들이 등록되는 중단 큐(Wait Queue)를 포함할 수 있다.

<29> 상기 대기 큐는 하나의 비트를 매핑하는 비트 벡터일 수 있다.

<30> 상기 쓰레드 제어 메모리는, 4개의 필드로 구성된 쓰레드 디스크립터(Thread Descriptor)와 세마포어 디스크립터(Semaphore Descriptor)를 포함할 수 있다.

<31> 상기 작업 스케쥴링을 위한 정보는 상기 쓰레드 디스크립터의 필드 중 상기 우선순위의 상위 비트 및 하위 비트를 나타내는 2개의 필드에 저장되고, 상기 세마포어 접근 제어를 위한 정보는 상기 세마포어 디스크립터의 필드 중 상기 세마포어 디스크립터의 사용 여부를 나타내는 플래그, 상기 세마포어의 종류를 나타내는 플래그 및 상기 세마포어 값이 저장되는 3개의 필드에 각각 저장될 수 있다.

<32> 상기 중단 큐는 상기 세마포어의 디스크립터 중 나머지 1개의 필드와, 상기 쓰레드 디스크립터 중 나머지 2개의 필드를 이용하여 구현될 수 있다.

<33> 상기 세마포어 디스크립터의 나머지 1개의 필드가 음수인 경우 한 개 이상의 작업이 상기 중단 큐에 등록되어 있음을 나타내며, 상기 쓰레드 디스크립터의 나머지 2개의 필드 중 제1 필드에는 첫 번째 작업의 ID가 저장되고, 제2 필드가 제1 값인 경우 타 작업이 추가적으로 상기 중단 큐에 등록되어 있음을 나타내고, 후속 작업의 ID가 상기 제1 필드에 저장되어 있고, 상기 제2 필드가 제2 값인 경우 상기 작업이 상기 중단 큐에 등록된 마지막 작업임을 나타낼 수 있다.

<34> 상기 쓰레드 제어 메모리는 하나의 싱글 포트(Single-Port) SRAM일 수 있다.

<35> 본 발명의 다른 측면에 따르면, 복수의 프로세서 코어를 포함하는 멀티 프로세서 시스템에 있어서, 상기 복수의 프로세서 코어와 연결되어 상기 복수의 프로세서 코어로부터의 운영체제 서비스 요청에 대해서 인터럽트 핸들링, 작업 동기화, 작업 스케쥴링, 컨텍스트 교환 및 작업 재개 동작을 수행하는 운영체제; 상기 복수의 프로세서 코어를 지원하는 명령어들이 미리 저장되는 공유형 명령어 캐시; 및 상기 복수의 프로세서 코어를 지원하는 데이터들이 미리 저장되는 공유형 데이터 캐시를 포함하는 멀티 프로세서 시스템이 제공된다.

<36> 상기 공유형 명령어 캐시는, 내부 통신 네트워크 및 복수의 온 칩 메모리를 포함하되, 하나의 명령어 블록을 구성하는 워드들이 서로 다른 온 칩 메모리에 분산 저장되고, 상기 분산 저장된 워드들을 동시에 읽을 수 있는 인터리브 명령어 캐시 메모리(Interleaved Instruction Cache Memory); 상기 인터리브 명령어 캐시 메모리에 저장된 명령어 블록의 일부분을 태그로 저장하고 있는 태그 메모리; 상기 운영체제 서비스 요청에 따른 명령어의 일부분과 상기 태그 메모리에 저장된 태그를 비교하여 상기 요청된 명령어가 상기 인터리브 명령어 캐시 메모리에 존재하는지를 확인하는 태그 매칭(Tag Matching)을 수행하고, 상기 태그 매칭 결과 상기 요청된 명령어가 상기 인터리브 명령어 캐시 메모리에 존재하지 않는 경우 상기 명령어 블록 요청 신호를 출력하는 캐시 제어기; 및 상기 명령어 블록 요청 신호에 따라 시스템 버스를 통해 외부 메모리로부터 명령어 블록을 읽어와 상기 인터리브 명령어 캐시 메모리에 저장하는 블록-필 제어기를 포함할 수 있다.

<37> 상기 인터리브 명령어 캐시 메모리는 서로 다른 경로(way)에 대해서 상기 복수의 온 칩 메모리에 분산 저장되는

순서가 다를 수 있다.

<38> 상기 공유형 명령어 캐시는 명령어 블록에 처음 접근하려는 경우에 상기 태그 매칭을 수행하고, 상기 명령어 블록에 속한 명령어들을 순차적으로 읽어오는 경우에는 상기 태그 매칭을 건너뛰는 태그 매칭 스키핑(Tag Matching Skipping) 방식을 이용할 수 있다.

<39> 상기 공유형 명령어 캐시는 상기 복수의 프로세서 코어 중 하나가 분기 명령을 수행한 경우 상기 프로세서 코어가 순차적인 주소를 가진 명령어를 요청했으나 상기 명령어가 타 명령어 블록에 저장된 경우 상기 태그 매칭을 수행할 수 있다.

<40> 상기 공유형 명령어 캐시는, 상기 복수의 프로세서 코어마다 할당되고, 상기 할당된 프로세서 코어가 명령어를 요청하기 전에 미리 상기 명령어를 상기 인터리브 명령어 캐시로부터 읽어와 저장하고 있는 명령어 버퍼를 더 포함할 수 있다.

<41> 상기 명령어 버퍼는 상기 할당된 프로세서 코어가 분기 명령을 수행하였는지 여부에 대한 분기 상태 신호(Branch Status Signal)와 상기 명령어 버퍼에 저장된 명령어들의 개수에 대한 프리-펫치 압력 신호(Pre-fetch Pressure Signal)를 상기 캐시 제어기에 전달하고, 상기 캐시 제어기는 상기 분기 상태 신호와 상기 프리-펫치 압력 신호들을 이용해 읽고자 하는 명령어를 결정한 후 상기 내부 통신 네트워크를 통해 상기 복수의 온 칩 메모리로 온 칩 메모리 선택 신호를 출력하여 상기 명령어가 상기 명령어 버퍼에 삽입되도록 할 수 있다.

<42> 상기 공유형 데이터 캐시는 히스토리 기반 계층적 태그 매칭 방식이 적용될 수 있다.

<43> 상기 공유형 데이터 캐시는, 내부 통신 네트워크 및 복수의 온 칩 메모리를 포함하되, 하나의 데이터 블록을 구성하는 워드들이 서로 다른 온 칩 메모리에 분산 저장되고, 상기 분산 저장된 워드들을 동시에 읽을 수 있는 인터리브 데이터 캐시 메모리(Interleaved Data Cache Memory); 상기 인터리브 데이터 캐시 메모리에 저장된 데이터 블록의 일부분을 태그로 저장하고 있는 태그 메모리; 상기 운영체제 서비스 요청에 따른 데이터의 일부분과 상기 태그 메모리에 저장된 태그를 비교하여 상기 요청된 데이터가 상기 인터리브 데이터 캐시 메모리에 존재하는지를 확인하는 태그 매칭을 수행하고, 상기 태그 매칭 결과 상기 요청된 데이터가 상기 인터리브 데이터 캐시 메모리에 존재하지 않는 경우 상기 데이터 블록 요청 신호를 출력하는 캐시 제어기; 및 상기 데이터 블록 요청 신호에 따라 시스템 버스를 통해 외부 메모리로부터 데이터 블록을 읽어와 상기 인터리브 데이터 캐시 메모리에 저장하는 블록-필 제어기를 포함할 수 있다.

<44> 상기 공유형 데이터 캐시는, 소정 시간 내에 접근하였던 온 칩 메모리의 주소들의 태그를 저장하는 히스토리 버퍼; 및 상기 히스토리 버퍼에 저장된 태그와 요청된 주소를 비교하여 상기 요청된 데이터가 상기 인터리브 데이터 캐시 메모리에 존재하는지를 판별하는 히스토리 테스터를 더 포함할 수 있다.

<45> 상기 캐시 제어기는 상기 히스토리 테스터에 의해 판별이 끝나지 않는 경우 상기 태그 메모리로부터 상기 태그를 읽어와 상기 태그 매칭을 수행할 수 있다.

<46> 상기 히스토리 버퍼는 스택 메모리 접근과 힙 메모리 접근을 구별하여 상기 태그를 저장할 수 있다.

<47> 본 발명의 또 다른 측면에 따르면, 복수의 프로세서 코어를 포함하는 멀티 프로세서 시스템에 있어서, 상기 복수의 프로세서 코어와 연결되어 상기 복수의 프로세서 코어로부터의 운영체제 서비스 요청에 대해서 인터럽트 핸들링, 작업 동기화, 작업 스케쥴링, 컨텍스트 교환 및 작업 재개 동작을 하드웨어적으로 수행하는 하드웨어 운영체제; 상기 복수의 프로세서 코어를 지원하는 명령어들이 미리 저장되는 공유형 명령어 캐시; 및 상기 복수의 프로세서 코어를 지원하는 데이터들이 미리 저장되는 공유형 데이터 캐시를 포함하는 멀티 프로세서 시스템이 제공된다.

<48> 전술한 것 외의 다른 측면, 특징, 이점이 이하의 도면, 특허청구범위 및 발명의 상세한 설명으로부터 명확해질 것이다.

효 과

<49> 본 발명에 따른 멀티 프로세서 시스템은 인터럽트 핸들링, 작업 동기화, 작업 스케쥴링, 컨텍스트 교환 등의 과정을 하드웨어적으로 구현한 하드웨어 운영체제를 통하여 통합적으로 수행함으로써, 상기 과정에 수반되는 오버헤드를 최소화한 효과가 있다.

<50> 또한, 운영체제가 하드웨어적으로 구현되므로, 프로세서가 소프트웨어 운영체제를 지원하기 위한 기능을 제공할

필요가 없어 보다 경량화된 프로세서를 사용할 수 있다.

<51> 또한, 복수의 프로세서 코어가 하나의 명령어 캐시와 데이터 캐시를 공유할 수 있는 공유형 명령어 캐시 및 공유형 데이터 캐시를 도입함으로써 작업 이전 오버헤드와 캐시 메모리 단편화 현상을 제거함으로써 멀티 프로세서 시스템의 성능을 대폭 개선한 효과가 있다.

<52> 또한, 공유형 캐시를 통해 좀 더 면적 효율이 좋은 대용량 온 칩 SRAM을 복수의 프로세서 코어가 공유할 수 있고, 각 프로세서 코어마다 독점적 캐시들이 분산된 경우와 비교할 때 동일한 성능을 내기 위해 보다 적은 캐시 메모리를 사용할 수 있으므로 면적 면에서도 효율적이다.

<53> 또한, 경량화된 프로세서를 사용하고 면적 면에서 효율적이어서 모바일용 멀티미디어 시스템에 최적으로 적용될 수 있다.

발명의 실시를 위한 구체적인 내용

<54> 본 발명은 다양한 변환을 가할 수 있고 여러 가지 실시예를 가질 수 있는 바, 특정 실시예들을 도면에 예시하고 상세한 설명에 상세하게 설명하고자 한다. 그러나, 이는 본 발명을 특정한 실시 형태에 대해 한정하려는 것이 아니며, 본 발명의 사상 및 기술 범위에 포함되는 모든 변환, 균등물 내지 대체물을 포함하는 것으로 이해되어야 한다. 본 발명을 설명함에 있어서 관련된 공지 기술에 대한 구체적인 설명이 본 발명의 요지를 흐릴 수 있다고 판단되는 경우 그 상세한 설명을 생략한다.

<55> 제1, 제2 등의 용어는 다양한 구성요소들을 설명하는데 사용될 수 있지만, 상기 구성요소들은 상기 용어들에 의해 한정되어서는 안 된다. 상기 용어들은 하나의 구성요소를 다른 구성요소로부터 구별하는 목적으로만 사용된다.

<56> 본 출원에서 사용한 용어는 단지 특정한 실시예를 설명하기 위해 사용된 것으로, 본 발명을 한정하려는 의도가 아니다. 단수의 표현은 문맥상 명백하게 다르게 뜻하지 않는 한, 복수의 표현을 포함한다. 본 출원에서, "포함하다" 또는 "가지다" 등의 용어는 명세서상에 기재된 특징, 숫자, 단계, 동작, 구성요소, 부품 또는 이들을 조합한 것이 존재함을 지정하려는 것이지, 하나 또는 그 이상의 다른 특징들이나 숫자, 단계, 동작, 구성요소, 부품 또는 이들을 조합한 것들의 존재 또는 부가 가능성을 미리 배제하지 않는 것으로 이해되어야 한다.

<57> 이하, 본 발명의 실시예를 첨부한 도면들을 참조하여 상세히 설명하기로 한다.

<58> 본 발명에서는 프로세서 코어에서 수행되는 작업(task)은 쓰레드(thread) 또는 프로세스(process)일 수 있으며, 이하에서는 발명의 이해와 설명의 편의를 위해 프로세서 코어에서 수행되는 작업이 쓰레드인 것을 가정하여 설명하기로 한다.

<59> 도 1은 본 발명의 일 실시예에 따른 멀티 프로세서 시스템의 구조를 나타낸 개략도이다. 멀티 프로세서 시스템(100), 복수의 프로세서 코어(120a, 120b, 120c, 120d, 이하 120으로 통칭함), 하드웨어 운영체제(Hardware Operating System)(110), 공유형 명령어 캐시(130), 공유형 데이터 캐시(140), 코프로세서(10), SDRAM(20)이 도시되어 있다.

<60> 일반적인 프로세서 코어는 외부 디바이스로부터의 서비스 요청이나 이벤트 발생을 통지받기 위해 인터럽트 방식을 이용한다. 여기서, 인터럽트 방식은 외부의 이벤트가 있는 경우 프로세서 코어가 현재 수행중인 작업을 일시 중단하고, 인터럽트 핸들러 루틴으로 강제 분기하는 것을 말한다. 외부의 이벤트 이외에도 내부적인 오류, 소프트웨어 인터럽트, 디버거 트랩, 페이지 폴트 등의 서비스들도 동일한 방식으로 처리될 수 있으며, 이를 예외 처리(Exception Handling)라 한다. 인터럽트 방식에 따른 인터럽트 핸들링을 위해서는 프로세서 코어가 내부적으로 이를 지원하기 위한 제어 로직은 물론 인터럽트 모드를 위한 전용 레지스터 세트를 가지고 있어야 한다.

<61> 본 발명의 일 실시예에서는, 이러한 인터럽트 핸들링을 프로세서 코어(120)가 아닌 하드웨어 운영체제(110)가 직접 담당한다. 따라서, 프로세서 코어(120)는 인터럽트를 지원하기 위한 로직 및 레지스터 세트를 가지고 있을 필요가 없으므로, 프로세서 코어 자체의 면적을 줄이는 것이 가능하다. 프로세서 코어(120)는 인터럽트 핸들링을 위한 레지스터가 제거되어 있어 일반적인 프로세서 코어에 비하여 좁은 면적을 차지한다.

<62> 예를 들어, 영국 ARM사의 ARM9 프로세서의 경우 데이터 패스의 면적이 약 15K 게이트 정도이고, 기본 컨텍스트를 위한 면적이 약 10K 게이트 정도이며, 시스템 서비스를 위해 부가적으로 제공하는 레지스터들의 면적이 약 15K 게이트 정도로, 도합 40K 게이트 정도의 면적을 차지한다. 하지만, 본 발명의 일 실시예에 따르면, 시스템 서비스는 하드웨어 운영체제(110)에 의하여 수행되기 때문에, 프로세서 코어(120)마다 별도의 지원이 필요하지

않아, 30K 게이트 수준의 경량화된 프로세서 코어(120)를 사용할 수 있다. 도 1에 도시된 것과 같이 4개의 프로세서 코어(120)를 통합한 경우 약 40K 게이트 정도를 절감할 수 있다.

<63> 여기서, 멀티 프로세서 시스템(100)에는 하드웨어 운영체제(110)를 위한 별도의 면적이 요구된다. 하지만, 이는 추후 설명할 쓰레드 제어 메모리를 이용한 작업 스케쥴링 방법과 작업 동기화 방법을 적용함으로써, 멀티 프로세서 시스템(100)에서 필요로 하는 하드웨어 운영체제(110)의 면적은 40K 게이트를 넘지 않는다. 즉, 하드웨어 운영체제(110)를 위한 면적 증가분와 경량 운영체제를 활용함으로써 얻어지는 프로세서 코어(120)의 면적 감소분이 상쇄되므로, 큰 면적 증가 없이 멀티 쓰레딩 오버헤드를 최소화함으로써 성능을 높일 수 있다. 멀티 쓰레딩 오버헤드에는 작업 스케쥴링 오버헤드, 컨텍스트 교환 오버헤드, 인터럽트 핸들링 오버헤드, 작업 동기화 오버헤드가 포함된다.

<64> 프로세서 코어(120)는 운영체제 버스(Operating System Bus)를 통하여 하드웨어 운영체제(110)와 연결된다. 프로세서 코어(120)는 운영체제 버스를 통하여 운영체제 서비스 요청, 즉 쓰레드의 생성 및 파괴, 세마포어의 생성, 값 변경 및 파괴에 관한 요청을 하드웨어 운영체제(110)에 전달한다. 프로세서 코어(120)는 코프로세서 버스를 통해 코프로세서(10)에 연결되어 데이터를 주고 받을 수 있다. 또한, 프로세서 코어(120)는 하드웨어 운영체제(110)와 컨텍스트 버스(Context Bus)로도 연결되어 있다. 하드웨어 운영체제(110)는 컨텍스트 버스를 통하여 중단된 쓰레드와 새로 재개할 쓰레드들의 컨텍스트를 교환한다.

<65> 본 발명의 다른 실시예에서 멀티 프로세서 시스템(100) 내에는 공유형 명령어 캐시(130) 및 공유형 데이터 캐시(140)가 구비되어 있고, 공유형 명령어 캐시(130) 및 공유형 데이터 캐시(140)는 공유 메모리 버스를 통해 SDRAM(20)에 접근할 수 있다.

<66> 캐시는 캐시 컨트롤러와 캐시 메모리로 구성된다. 캐시 메모리는 온 칩 SRAM 셀(cell)을 이용하여 구현될 수 있다. SRAM 셀은 용량이 클수록 밀도(density)가 높아진다. 예를 들어, 0.18um 공정에서 2KB의 SRAM 셀은 1 비트(bit)당 약 10 um^2 정도의 면적을 차지함에 반하여, 8KB의 SRAM 셀은 1 비트당 6 um^2 정도의 면적을 차지한다. 만일 4개의 프로세서 코어가 각각 2KB 캐시를 가지고 있는 경우, 캐시 메모리가 차지하는 면적은 약 80,000 um^2 이지만, 하나의 8KB 공유형 캐시를 사용하는 경우에는 48,000 um^2의 면적만을 차지한다. 즉, 캐시 메모리가 차지하는 면적이 대략 반이 된다. 일반적으로 캐시 메모리는 전체 시스템의 면적의 50 ~ 75%를 차지하는 만큼 공유형 캐시를 활용함에 따라 얻어지는 면적 감소의 효과가 매우 크다.

<67> 도 1에서는 컨텍스트 버스, 운영체제 버스, 코프로세서 버스, 공유 메모리 버스를 구분하여 도시하였으나, 이는 어디까지나 기능상의 구분으로 실제로는 다양한 구조의 물리적 버스로 구현할 수 있다. 일 실시예로, 공유 메모리 버스와 코프로세서 버스를 하나의 물리적 버스로 통합하고, 컨텍스트 버스와 운영체제 버스를 하나의 물리적 버스로 통합하는 구현도 가능하다. 또 다른 실시예로, 운영체제 버스, 코프로세서 버스를 하나의 물리적 버스로 통합하고 공유 메모리 버스와 컨텍스트 버스는 별개의 물리적 버스로 구현하는 것도 가능하다.

<68> 본 발명의 일 실시예에 따른 멀티 프로세서 시스템은 내부에 구비된 하드웨어 운영체제(110)를 통해 멀티 쓰레딩 오버헤드를 줄일 수 있다. 이에 대해서 도 2 내지 도 5를 참조하여 상세히 후술하기로 한다.

<69> 또한, 본 발명의 다른 실시예에 따른 멀티 프로세서 시스템은 내부에 구비된 공유형 명령어 캐시(130) 및 공유형 데이터 캐시(140)를 통해 작업 이전 오버헤드를 줄이고 전체 면적을 줄일 수 있다. 이에 대해서 도 6 내지 도 10을 참조하여 상세히 후술하기로 한다.

<70> 또한, 본 발명의 또 다른 실시예에 따른 멀티 프로세서 시스템은 하드웨어 운영체제(110)와, 공유형 명령어 캐시(130) 및 공유형 데이터 캐시(140)를 포함하여 멀티 쓰레딩 오버헤드 및 작업 이전 오버헤드를 줄이고 전체 면적을 줄일 수 있다.

<71> 이하에서는 우선 일 실시예에 따라 하드웨어 운영체제(110)를 통해 멀티 쓰레딩 오버헤드를 줄이는 멀티 프로세서 시스템에 대해서 설명하기로 한다.

<72> 도 2는 하드웨어 운영체제의 구조를 설명하는 상세도이며, 도 3은 쓰레드 제어 메모리의 구조를 설명하는 상세도이다.

<73> 하드웨어 운영체제(110)는 주 제어기(210), 쓰레드 관리자(230), 컨텍스트 관리자(250), 컨텍스트 버퍼(260), 쓰레드 제어 메모리(220), 컨텍스트 메모리(240)를 포함한다.

<74> 프로세서 코어(120)는 데이터패스(126), 컨텍스트 제어기(124), 레지스터 파일(122)을 포함한다. 데이터패스

(126)는 운영체제 버스를 통해 하드웨어 운영체제(110)의 주 제어기(210)와 직접 연결된다. 컨텍스트 제어기(124)는 컨텍스트 버스를 통해 하드웨어 운영체제(110)의 컨텍스트 관리자(250)와 직접 연결되며, 컨텍스트 제어기(124)와 컨텍스트 관리자(250)는 프로세서 코어(120)에서 수행 중이거나 수행이 중단된 작업의 컨텍스트를 주고 받는다. 레지스터 파일(122)은 프로세서 코어(120)에서 수행되는 작업의 컨텍스트를 임시로 저장한다.

<75> 하드웨어 운영체제(110)는 인터럽트 핸들링, 작업 동기화, 작업 스케쥴링, 컨텍스트 교환, 작업 재개의 다섯 가지 세부 단계로 구성되는 멀티 쓰레드 프로그램 수행 과정을 담당한다. 각 세부 단계에 대해서는 앞서 상세히 설명하였는 바 상세한 설명은 생략하기로 한다.

<76> 주 제어기(210)는 운영체제 버스에 연결되어 프로세서 코어(120)가 제기한 운영체제 서비스 요청을 받고, 이에 따라 내부의 쓰레드 관리자(230), 컨텍스트 관리자(250) 등의 다른 구성요소들을 제어한다.

<77> 쓰레드 제어 메모리(220)는 작업 정보인 쓰레드의 작업 상태, 우선순위, 중단 큐(Wait Queue), 세마포어 상태 및 이들의 결합 중 하나를 저장한다.

<78> 쓰레드 관리자(230)는 대기 큐(Ready Queue)에 등록된 작업들 중에서 가장 높은 우선순위의 작업을 선택하고, 세마포어 접근을 제어한다.

<79> 컨텍스트 메모리(240)는 생성된 작업들의 컨텍스트가 저장되어 있다.

<80> 컨텍스트 버퍼(260)는 수행을 재개할 작업의 컨텍스트를 임시로 저장하거나, 수행이 중단된 작업의 컨텍스트가 임시로 저장된다. 컨텍스트 버퍼(260)는 온 칩 메모리로 이루어질 수 있다.

<81> 컨텍스트 관리자(250)는 쓰레드 관리자(230)에 의해 선택된 작업에 대한 컨텍스트를 컨텍스트 메모리(240)로부터 읽어와 컨텍스트 버퍼(260)에 저장하거나, 역으로 수행이 중단된 작업에 상응하여 컨텍스트 버퍼(260)에 저장된 컨텍스트를 컨텍스트 메모리(240)에 저장한다.

<82> 컨텍스트 관리자(250)는 재개할 작업의 컨텍스트를 컨텍스트 메모리(240)로부터 읽어와서 컨텍스트 버퍼(260)에 저장한다. 그리고 임의의 한 프로세서 코어(120)에서 수행되는 작업이 중단되거나 이 작업의 우선순위보다 컨텍스트 버퍼(260)에 준비된 작업의 우선순위가 높은 경우 컨텍스트 버스를 통해 컨텍스트를 교환한다. 이 과정은 소프트웨어의 개입 없이 하드웨어적으로 수행되므로 매우 빠르게 작업을 전환할 수 있다.

<83> 하드웨어적 수행을 위해서 하드웨어 운영체제(110)는 쓰레드들의 컨텍스트를 저장하는 메모리, 작업을 스케쥴링할 때 필요한 정보를 저장하는 메모리, 세마포어 접근을 제어하기 위한 정보를 저장하는 메모리, 중단 큐를 구현하기 위한 메모리, 대기 큐를 구현하기 위한 메모리를 필요로 한다.

<84> 우선, 컨텍스트를 저장하는 메모리(컨텍스트 메모리(240)에 해당함)는 용량이 매우 크기 때문에 이들 모두를 온 칩 메모리로 구현하는 것이 불가능한 바 외부의 메모리를 이용하여 구현한다. 반면에, 나머지 메모리들은 용량이 그리 크지 않기 때문에 온 칩 메모리로 구현하는 것이 가능하다. 본 발명의 실시예에서는 컨텍스트 메모리(240)를 제외한 나머지 메모리들을 다음과 같이 구현한다.

<85> 우선 대기 큐는 각 쓰레드마다 하나의 비트를 매핑하는 비트 벡터(bit vector)로 구현한다. 예를 들어, 한 쓰레드가 대기 상태인 경우 이 쓰레드에 대응되는 비트는 제1 값(예를 들어, 1)을 가지고 이 쓰레드가 중단된 경우에는 제2 값(예를 들어, 0)을 가질 수 있다. 대기 큐를 비트 벡터로 구현하는 경우, 각 쓰레드 별로 1개의 레지스터(6개의 NAND 게이트의 면적에 해당함)만으로 표현할 수 있다. 일반적으로 내장형 시스템에서 쓰레드의 개수는 128개를 넘지 않으므로, 대기 큐의 면적은 1K 게이트를 넘지 않는다.

<86> 작업 스케쥴링을 위해서는 각 쓰레드 별로 우선순위를 알아야 한다. 여기서, 작업 스케쥴링을 위한 정보는 각 쓰레드의 우선순위이다. 대기 큐에 등록된(즉, 해당 비트가 1로 설정되어 있는) 쓰레드 별로 우선순위 값을 읽고, 순차적으로 우선순위를 비교해나가며 우선순위 값이 가장 큰 쓰레드를 선택한다. 이러한 스케쥴링 방식에 따를 때, 대기 큐에 등록된 쓰레드의 개수가 k인 경우 스케쥴링은 k 클럭 싸이클이 필요하다. 만일 대기 큐에 등재된 쓰레드가 적은 경우, 작업 스케쥴링은 그리 오래 걸리지 않는다. 특히 하드웨어 운영체제 기반의 멀티 프로세서 시스템에서 작업 스케쥴링은 프로세서 코어(120)와는 병렬로 수행되기 때문에 시스템의 성능을 저하시키지는 않는다. 반면, 대기 큐에 등재된 쓰레드가 많은 경우, 작업 스케쥴링 자체는 장시간이 소요될 수 있다. 하지만, 대기 큐에 쓰레드가 많다는 것은 이미 대기중인 다른 쓰레드가 컨텍스트 버퍼(260)에 프리펫치되어 있을 가능성이 매우 높기 때문에 스케쥴링 시간은 성능 면에서 그리 문제되지 않는다. 이러한 스케쥴링 방식의 장점은 하드웨어로 구현할 시 우선순위를 비교하는 비교기가 단 하나만 필요하고, 쓰레드의 우선순위 정보를 순차

적 접근이 가능한 온 칩 메모리(SRAM)에 저장할 수 있어 면적 면에서 유리하다는 점이다.

<87> 세마포어 접근을 제어하기 위해서는 각 세마포어 별로 세마포어의 상태를 나타내는 세마포어 상태 플래그와 세마포어 값이 필요하고, 세마포어에 대기하고 있는 쓰레드들이 등록되는 중단 큐가 필요하다. 여기서, 세마포어 접근 제어를 위한 정보는 세마포어 상태 플래그 및 세마포어 값이다.

<88> 세마포어에 관한 대표적인 서비스는 세마포어 대기(Semaphore wait)와 세마포어 포스트(Semaphore post)이다.

<89> 임의의 한 쓰레드가 어떤 세마포어에 세마포어 대기를 요청한 경우에는 다음과 같이 동작한다. 하드웨어 운영체제(110)는 세마포어 값을 읽고, 이 값이 양수이면 1 감소시키고 쓰레드의 수행을 즉시 재개시킨다. 이 값이 0 또는 음수이면 그 즉시 이 쓰레드를 중단시킨 후, 이 세마포어의 중단 큐에 삽입한 후, 해당 프로세서에는 다른 쓰레드를 할당(assign)하여 수행시킨다.

<90> 한 쓰레드가 임의의 한 세마포어에 세마포어 포스트를 요청한 경우에는 다음과 같이 동작한다. 하드웨어 운영체제(110)는 세마포어 값을 읽고, 이 값이 0 또는 양수이면 세마포어 값을 요청된 양만큼 증가시킨다. 이 값이 음수이면 요청된 값만큼의 쓰레드를 중단 큐로부터 인출하여 대기 큐에 삽입하고, 작업 스케쥴링을 통해 여러 프로세서 코어(120)에 분산시켜 수행한다.

<91> 본 발명의 일 실시예에 따른 하드웨어 운영체제(110)는 세마포어 포스트, 세마포어 값 갱신, 중단 큐로부터 쓰레드를 인출, 쓰레드를 대기 큐에 삽입, 작업 스케쥴링, 컨텍스트 교환의 과정들이 일괄적으로 하드웨어적으로 빠르게 수행된다.

<92> 여기서, 작업 스케쥴링을 위한 정보, 세마포어 접근 제어를 위한 정보, 각 세마포어 별 중단 큐는 쓰레드 제어 메모리(220)로 구현된다. 쓰레드 제어 메모리(220)는 하나의 온 칩 SRAM으로 구현될 수 있다. 도 3을 참조하면, 쓰레드 제어 메모리(220)의 구조가 도시되어 있다. 쓰레드 제어 메모리(220)에 저장된 작업 스케쥴링을 위한 정보를 쓰레드 디스크립터(Thread Descriptor)라고 부르고, 세마포어 접근 제어를 위한 정보를 세마포어 디스크립터(Semaphore Descriptor)라고 부른다.

<93> 쓰레드 디스크립터(222)는 PRIO, WCNT, N, NEXT의 네 개의 필드로 구성된다. PRIO는 우선순위 값의 상위 비트이고 WCNT는 우선순위 값의 하위 비트이다. WCNT의 비트 수를 NWCNT라고 할 때, 쓰레드의 우선순위 값은 ((PRIO<<NWCNT) | WCNT)이다. 우선순위를 나타내는 정보 중에서 PRIO는 소프트웨어에 의하여 쓰레드에 부여된 우선순위 값이며 쓰레드 관리자(230)에 의해 변하지 않고, WCNT는 공평한 스케쥴링을 위하여 쓰레드 관리자(230)가 값을 변경한다. 예를 들어, 가장 높은 우선순위 값이 p이고, PRIO 값이 p인 쓰레드가 n개 있다면, 이 쓰레드들의 WCNT 값은 다음과 같이 갱신된다. 쓰레드 관리자(230)에 의하여 선택된 쓰레드의 WCNT 값은 0으로 설정하고, 나머지 쓰레드들의 WCNT 값은 1 증가시킨다. N과 NEXT 필드는 중단 큐(226)를 구현하기 위한 것으로 추후 설명하기로 한다.

<94> 세마포어 디스크립터(224)는 E, B, VALUE, NEXT의 네 개의 필드로 구성된다. E는 해당 세마포어 디스크립터가 이미 사용중인 경우 1의 값으로, 그렇지 않은 경우 0의 값으로 설정하는 플래그이다. B는 해당 세마포어가 바이너리 세마포어인 경우 1의 값으로, 카운팅 세마포어인 경우 0의 값으로 설정되는 플래그이다. VALUE는 세마포어 값이 저장되는 필드이다. E, B 및 VALUE의 세 필드에 저장되는 값을 이용해 세마포어 접근을 제어하는 방법(세마포어 대기, 세마포어 포스트 등)은 앞서 설명하였는바 생략하기로 한다.

<95> 중단 큐(226)는 세마포어 디스크립터(224)의 NEXT 필드와 쓰레드 디스크립터(222)의 N, NEXT 필드를 이용해 구현된다. 만일 세마포어 디스크립터(224)의 VALUE가 음수인 경우 한 개 이상의 쓰레드가 중단 큐에 등록되어 있다는 의미로, 첫 번째 쓰레드의 ID가 세마포어 디스크립터(224)의 NEXT 필드에 저장된다. 쓰레드 디스크립터(222)의 N 필드가 1인 경우 다른 쓰레드가 추가적으로 중단 큐에 등록되어 있다는 의미로, 후속 쓰레드의 ID가 쓰레드 디스크립터(222)의 NEXT 필드에 저장된다. 만일 N 필드가 0인 경우에는, 이 쓰레드가 중단 큐에 등록된 마지막 쓰레드라는 의미이다.

<96> 쓰레드 디스크립터(222)와 세마포어 디스크립터(224)를 상술한 것과 같이 표현함으로써, 작업 스케쥴링을 위한 정보, 세마포어 접근 제어를 위한 정보, 중단 큐가 단 하나의 싱글 포트(Single-Port) SRAM을 이용해 구현될 수 있고, 이로써 면적을 크게 줄일 수 있다.

<97> 코프로세서 버스에는 하나 이상의 코프로세서(10)가 연결되어 프로세서 코어(120)의 수행을 도울 수 있다. 코프로세서(10)는 하드웨어 운영체제(110)에 락(Lock), 릴리즈(Release), 프로세서 코어 ID의 세 가지 신호를 전달한다. 프로세서 코어 ID 신호는 현재 코프로세서(10)에 접근한 프로세서 코어(120)의 번호를 의미하고, 락 신호

는 코프로세서(10)에 요청된 작업이 아직 완료되지 않았음을 의미하며, 릴리즈 신호는 요청된 작업이 완료되었음을 의미한다. 하드웨어 운영체제(110)는 락 신호가 인가된 경우 프로세서 코어 ID 신호가 가리키는 프로세서 코어(120)가 수행하고 있는 작업을 중단시키고, 컨텍스트 버퍼(260)에 대기 중인 작업으로 컨텍스트 교환을 수행한다. 이후, 릴리즈 신호가 인가되면, 중지되었던 작업을 컨텍스트 버퍼(260)에 다시 준비하고 현재 작업이 없는 프로세서 코어 혹은 재개할 작업보다 우선순위가 낮은 작업이 수행되고 있는 프로세서 코어를 대상으로 컨텍스트 교환을 수행한다. 위와 같은 기능을 통하여 코프로세서(10)로부터의 인터럽트 요청, 작업의 중단, 그리고 작업의 재개의 과정을 하드웨어적으로 수행하여 성능을 개선할 수 있다.

<98> 도 4는 소프트웨어 운영체제에서의 멀티 쓰레드 프로그램 수행 과정 흐름도이고, 도 5는 본 발명의 일 실시예에 따른 하드웨어 운영체제에서의 멀티 쓰레드 프로그램 수행 과정 흐름도이다.

<99> 도 4를 참조하면, 소프트웨어 운영체제에서의 멀티 쓰레드 프로그램 수행은 단계 S410부터 S470까지의 7단계로 구분된다.

<100> 응용 작업 수행 중(단계 S400) 일단 인터럽트가 발생(단계 S410)하면 수행 중이던 응용 작업을 중단하고 단계 S420부터 단계 S470까지의 각 과정을 순차적으로 수행한 후 응용 작업을 재개한다(단계 S480).

<101> 인터럽트가 발생(단계 S410)하면, 인터럽트 핸들러를 수행한다(단계 S420). 그리고 작업 동기화를 수행하고 대기 큐에 삽입한다(단계 S430). 작업 스케쥴링을 수행(단계 S440)한 후 현재 쓰레드의 컨텍스트를 외부 메모리에 대피(단계 S450)하거나 재개할 쓰레드의 컨텍스트를 외부 메모리로부터 읽어온다(단계 S460). 단계 S450 및 단계 S460는 컨텍스트 교환 과정에 해당하며, 소프트웨어 운영체제의 경우 이 과정 중에 많은 클럭 싸이클을 소모하게 된다.

<102> 도 5를 참조하면, 하드웨어 운영체제에서의 멀티 쓰레드 프로그램 수행은 단계 S510부터 S570까지의 7단계로 구분된다. 하드웨어 운영체제에서의 각 단계는 이에 대응되는 소프트웨어 운영체제에서의 단계에 비해 매우 짧은 시간 내에 완수될 수 있다.

<103> 하드웨어 운영체제에서의 각 단계 중 컨텍스트 교환 직전까지의 단계 S510 내지 S550은 프로세서 코어와 무관하게 수행된다. 쓰레드 관리자에 의해 선택되어 재개할 쓰레드의 컨텍스트를 외부 메모리로부터 모두 읽어 들여 컨텍스트 버퍼에 저장 완료한 후 단계 S560에서 컨텍스트 교환을 개시한다. 현재 수행 중이던 쓰레드의 컨텍스트도 일단 컨텍스트 버퍼에 대피된 후 나중에 하드웨어 운영체제에 의해 외부 메모리에 저장되기 때문에, 프로세서 코어 관점에서 멀티 쓰레딩 오버헤드는 극히 적다.

<104> 이와 같이 하드웨어 운영체제 기반의 멀티 프로세서 시스템에서 각 프로세서 코어들은 좀 더 많은 연산 자원을 응용 작업에 할애할 수 있으므로, 결과적으로 더욱 우수한 성능을 보일 수 있다.

<105> 이하에서는 다른 실시예에 따라 공유형 명령어 캐시(130) 및 공유형 데이터 캐시(140)를 통해 작업 이전 오버헤드를 줄이고 전체 면적을 줄이는 멀티 프로세서 시스템에 대해서 설명하기로 한다.

<106> 프로세서 코어를 원활하게 구동하기 위해서는 앞으로 사용될 확률이 높은 명령어들을 미리 온 칩 메모리에 저장해 둔 명령어 캐시가 필요하다. 하나의 프로세서 코어만을 지원할 수 있는 일반적인 명령어 캐시는 순차적인 명령어들로 구성된 메모리 블록을 온 칩 메모리에 저장하고, 해당 메모리 블록이 캐시에 존재함을 확인할 수 있도록 명령어 주소의 일부분을 태그(Tag)로서 별도의 메모리에 저장한다. 이와 같이 명령어 블록을 저장하는 온 칩 메모리를 캐시 메모리라 하고, 태그를 저장하는 온 칩 메모리를 태그 메모리라 한다. 일반적인 명령어 캐시가 명령어를 프로세서에 제공하는 과정은 (a) 프로세서 코어가 요청한 주소의 명령어의 일부분과 태그 메모리에 저장된 태그를 비교하여, 요청된 명령어가 캐시 메모리에 존재하는지를 확인하는 태그 매칭(Tag Matching) 단계와, (b) 요청된 명령어가 캐시 메모리에 존재하지 않는 경우 외부 메모리로부터 명령어 블록을 읽어와 캐시 메모리에 저장하는 명령어 블록 적재(Instruction Blocking Loading) 단계, (c) 태그 매칭을 통해 캐시 메모리에 존재함이 확인된 경우 혹은 명령어 블록 적재가 끝난 경우, 명령어를 캐시 메모리로부터 읽어오는 명령어 읽기(Instruction Reading) 단계로 구성된다.

<107> 복수의 프로세서 코어(120)에 명령어를 동시에 지원할 때에는 복수의 프로세서 코어(120)가 캐시 메모리에 접근하려고 하는 캐시 메모리 충돌(Cache Memory Conflict) 및/또는 동시에 태그를 매칭하려는 태그 매칭 충돌(Tag Matching Conflict)이 발생한다. 이러한 충돌을 해소하기 위하여 본 발명의 다른 실시예에 따른 멀티 프로세서 시스템(100)은 공유형 명령어 캐시(130)를 포함한다.

<108> 도 6은 공유형 명령어 캐시의 구조를 설명하는 상세도이며, 도 7은 메인 메모리의 데이터 배치 방식을 나타낸

도면이고, 도 8은 기존 캐시 메모리의 데이터 배치 방식을 나타낸 도면이며, 도 9는 본 발명의 일 실시예에 따른 인터리브 명령어 캐시 메모리의 데이터 배치 방식을 나타낸 도면이고, 도 10은 공유형 데이터 캐시의 구조를 설명하는 상세도이다.

<109> 도 6을 참조하면, 복수의 프로세서 코어(120a, 120b, 120c, 120d, 이하 120이라 통칭함), 공유형 명령어 캐시(130), 명령어 버퍼(610), 인터리브 명령어 캐시 메모리(Interleaved Instruction Cache Memory)(620), 태그 메모리(630a, 630b, 이하 630이라 통칭함), 캐시 제어기(640), 블록-필 제어기(650)가 도시되어 있다.

<110> 복수의 프로세서 코어(120)에 명령어를 지원하기 위해서는 태그 매칭과 명령어 읽기가 각 프로세서 코어(120)마다 동시에 수행되어야 한다. 이러한 동시적인 명령어 읽기를 지원하기 위해서 인터리브 명령어 캐시 메모리(620)를 이용한다. 인터리브 명령어 캐시 메모리(620)는 복수의 온 칩 메모리(622a, 622b, 622c, 622d, 이하 622라 통칭함)를 이용해 캐시 메모리를 구성하되, 하나의 명령어 블록을 구성하는 명령어들이 서로 다른 온 칩 메모리에 분산되도록 저장한다.

<111> 인터리브 명령어 캐시 메모리(620)는 내부 통신 네트워크(624)와, 하나 이상의 온 칩 메모리(622)를 포함한다. 내부 통신 네트워크(624)는 캐시 제어기(640)로부터의 온 칩 메모리 선택 신호에 따라 선택된 온 칩 메모리(622)에 명령어를 저장하거나 선택된 온 칩 메모리(622)에 저장된 명령어를 프로세서 코어(120)로 보낸다.

<112> 이 구조는 기존의 세트-어소시에이티브 캐시(Set-associative cache)의 메모리 구조와 유사해 보이지만, 기존의 메모리 구조에서는 하나의 명령어 블록이 하나의 온 칩 메모리에 저장된다는 점에서 다르다. 복수의 프로세서 코어(120)가 요청한 명령어들이 서로 다른 온 칩 메모리에 저장되어 있는 경우, 서로 다른 주소의 명령어를 동시에 읽어서 해당 프로세서 코어에 제공할 수 있다. 그러나 만일 두 개의 프로세서 PA와 PB가 서로 다른 메모리를 참조하는데, 우연히 이들이 같은 온 칩 메모리에 저장되어 있다면, 하나의 프로세서는 수행이 중단되어야 한다.

<113> 이 때, 명령어들은 순차적으로 읽힐 가능성이 매우 높기 때문에, 기존의 세트-어소시에이티브 캐시 메모리 방식에서는 한 프로세서가 명령어 블록을 모두 참조하고 다음 명령어 블록으로 넘어가거나, 분기 명령을 통해 다른 명령어 블록을 참조하게 될 때까지 계속 충돌이 발생하고 성능을 지연시킨다. 이에 반하여, 인터리브 명령어 캐시 메모리 구조에서는 최초 한 번만 시간 지연이 발생하고, 그 다음부터는 온 칩 메모리 접근 충돌이 자연 해소된다. 이에 관한 설명을 도 8을 통해 좀 더 상세하게 설명하면 다음과 같다.

<114> 도 8을 참조하면, 기존의 세트-어소시에이티브 캐시(Set-associative cache)가 도시되어 있다. 기존의 세트-어소시에이티브 캐시는 동시적인 데이터 읽기/쓰기(Read/Write)가 가능하도록 여러 개의 메모리 셀들(810, 820, 830, 840)을 병렬 연결하여 사용한다. 여기서, 각 메모리 셀(810, 820, 830, 840)은 SRAM으로 구현된다.

<115> 하지만, 도 9를 참조하면, 본 발명의 실시예에 따른 인터리브 명령어 캐시 메모리는 도 8에 도시된 세트-어소시에이티브 캐시와 같이 여러 개의 메모리 셀들(910, 920, 930, 940)이 병렬 연결되어 있다. 여기서, 각 메모리 셀(910, 920, 930, 940)은 SRAM으로 구현된다. 하지만, 복수의 프로세서 코어(120)의 동시적인 접근으로 인한 충돌을 최소화하기 위하여 각 메모리 셀(910, 920, 930, 940)에 데이터를 배치하는 방식이 다르다.

<116> 메인 메모리(700)의 데이터들은 N개의 워드로 구성된 블록단위로 캐시 메모리에 저장된다. 도 7 내지 도 9에서는 하나의 블록이 4개의 워드로 구성된다고 가정한다. 또한, 메인 메모리(700)의 A, B, C, D 블록들은 각각 Way0, Way1, Way2, Way3에 캐싱(caching)되어 있다고 가정한다.

<117> 일반적인 세트-어소시에이티브 캐시 구조에서는 각 경로(way) 별로 하나의 SRAM을 할당한다. 이는 SRAM의 타이밍 특성 때문이다. 태그 메모리와 캐시 메모리 구현에 이용되는 동기식 SRAM(Synchronous SRAM)은 주소(address)를 인가하면, 다음 클록 싸이클에 데이터가 출력된다. 따라서, 태그 매칭(Tag matching)은 두 싸이클에 걸쳐 이루어진다. 첫 번째 사이클에는 모든 태그 메모리에 주소를 인가하고, 다음 싸이클에 태그 메모리로부터 태그 정보들을 받아 이 중에서 히트(hit)인 경로를 찾아낸다. 만일, 태그 매칭이 완료되어 데이터가 저장되어 있는 경로를 알아낸 다음 캐시 메모리에 주소를 인가하고, 그 다음 싸이클에 데이터를 읽으면 총 세 클록 싸이클이 소요되며, 성능이 크게 떨어진다. 따라서, 대부분의 캐시들은 태그 매칭의 첫 번째 싸이클에 모든 캐시 메모리에도 주소를 인가하고, 그 다음 싸이클에 데이터가 저장된 경로를 알아내면, 해당 캐시 메모리가 출력한 데이터를 프로세서 코어에 제공하도록 한다. 즉, 모든 경로들이 동시에 읽혀야 하므로, 각 경로마다 하나의 SRAM을 할당한다(도 8 참조).

<118> 하지만, 이러한 방식은 복수의 프로세서 코어를 지원하는 공유형 캐시에는 적합하지 않다. 일반적으로 캐시, 특히 명령어 캐시는 순차적으로 데이터를 요청하는 비율이 높다. 예를 들어, 한 프로세서 코어가 Way0에 임의의

순간에 접근하였다면, 그 다음 싸이클에도 Way0로부터 그 데이터를 읽어올 가능성이 매우 높다. 제1 프로세서 코어와 제2 프로세서 코어가 동시에 Way0에 접근한다고 가정하면, 하나의 프로세서 코어는 데이터를 받지 못해 지연(stall)되어야 한다. 하지만, 그 다음 싸이클에도 제1 프로세서 코어와 제2 프로세서 코어는 계속 충돌이 나게 되고, 심각한 성능 저하를 유발하는 문제점이 있다.

<119> 이를 해소하기 위하여, 본 발명의 일 실시예에서는 도 9에 도시된 것과 같이 데이터를 배치한다. 우선, 블록을 구성하는 각 워드들은 서로 다른 메모리 셀에 배치한다. 그리고 캐시 메모리가 4개의 메모리 셀(910, 920, 930. 940), 즉 M0, M1, M2, M3로 구성되어 있고, 네 개의 경로, 즉 Way0, Way1, Way2, Way3가 존재한다면, Way0를 구성하는 워드들은 순서대로 M0, M1, M2, M3에, Way1를 구성하는 워드들은 M1, M2, M3, M0에, Way2를 구성하는 워드들은 순서대로 M2, M3, M0, M1에, Way3를 구성하는 워드들은 M3, M0, M1, M2에 배치한다. 이러한 배치에 의하면, 첫 번째 싸이클에는 제1 프로세서 코어와 제2 프로세서 코어가 충돌나서 한 프로세서 코어가 지연되더라도, 그 다음 싸이클에는 충돌이 해소된다.

<120> 또한, 공유형 명령어 캐시(130)는 동시적인 태그 매칭을 지원해야 한다. 그러나, 태그 메모리(630)의 경우에는 상술한 인터리브 구조로 구현할 수가 없다. 그리고 동시적인 태그 매칭을 지원하기 위하여 복수의 태그 메모리 사본을 관리하는 것은 면적 면에서 비효율적이다.

<121> 명령어의 경우 순차적으로 접근되는 경향이 매우 강하다. 따라서, 공유형 명령어 캐시(130)는 한 명령어 블록에 처음 접근하려는 경우에만 태그 매칭을 수행하고, 해당 명령어 블록에 속한 명령어들을 순차적으로 읽어오는 경우에는 태그 매칭을 건너뛰는 태그 매칭 스키핑(Tag Matching Skipping) 방식을 이용한다. 태그 매칭 스키핑 방식에 의할 경우 태그 매칭의 빈도를 줄일 수 있는 효과가 있다.

<122> 공유형 명령어 캐시(130)는 프로세서 코어(120)가 분기 명령을 수행한 경우, 프로세서 코어(120)가 순차적인 주소를 가진 명령어를 요청했으나 해당 명령어가 다른 명령어 블록에 저장되어 있는 경우에만 태그 매칭을 수행한다.

<123> 또한, 공유형 명령어 캐시(130)는 각 프로세서 코어 별로 명령어 버퍼(610)를 더 포함할 수 있다. 인터리브 명령어 캐시 메모리(620)를 이용하는 경우 복수의 프로세서 코어가 동일한 온 칩 메모리에 접근한다면 단 하나의 프로세서 코어에만 명령어를 공급할 수 있고 나머지 프로세서 코어들은 명령어를 공급받지 못해 수행이 중단되므로 성능이 저하된다. 또한, 태그 매칭 스키핑 방식은 서로 다른 프로세서 코어에서 분기 명령을 동시에 수행하는 경우에는, 단 하나의 프로세서 코어에 대해서만 태그 매칭을 수행할 수 있고, 나머지 프로세서 코어들은 태그 매칭을 하지 못하여 수행이 중단되므로 성능이 저하된다. 명령어 버퍼(610)는 이러한 문제점들을 해결한다.

<124> 공유형 명령어 캐시(130)는 각 프로세서 코어(120)가 명령어를 요청하기 전에 미리 명령어를 인터리브 명령어 캐시 메모리(620)로부터 읽어와 명령어 버퍼(610)에 저장하고 있다. 그리고 프로세서 코어(120)가 명령어를 요청하되 해당 명령어가 명령어 버퍼(610)에 저장되어 있는 경우, 명령어 버퍼(610)에서 제공한다. 명령어 버퍼(610)를 내장한 경우와 그렇지 않은 경우를 비교할 때 캐시 메모리의 충돌과 태그 매칭 충돌은 같은 횟수로 발생하지만, 명령어 버퍼(610)를 내장한 경우에는 충돌이 발생하더라도 명령어 버퍼(610)에서 프로세서 코어(120)로 명령어를 제공할 수 있으므로 성능이 개선되는 효과가 있다.

<125> 명령어 버퍼(610)는 분기 분별기(Branch Checker)(612)를 포함할 수 있다. 분기 분별기(612)를 통하여 프로세서 코어(120)가 분기 명령을 수행하였는지를 알아낸다. 만일 프로세서 코어(120)가 분기 명령을 수행한 경우, 명령어 버퍼(610)를 비우고 캐시 제어기(640)에 분기 상태 신호(Branch Status Signal)을 통해 알린다. 명령어 버퍼(610)를 늘리는 경우 좀 더 많은 명령어들이 미리 펫치될 수 있으므로 성능이 개선된다. 하지만 이를 위해서는 온 칩 메모리(622)와 명령어 버퍼(610) 간 대역폭(Bandwidth)을 명령어 버퍼(610)와 프로세서 코어(120) 간 대역폭보다 증가시켜야 한다. 도 6에는 프로세서 코어(120)와 명령어 버퍼(610) 간에는 32-bit 회선으로 연결되고 명령어 버퍼(610)와 온 칩 메모리(622) 간에는 64-bit 회선으로 연결된 경우를 도시하고 있으나, 이는 일 실시예에 불과하며 다른 구현도 가능하다.

<126> 캐시 제어기(640)는 캐시 메모리 충돌과 태그 매칭 충돌이 발생한 경우, 명령어 펫치 요청 중에서 명령어 버퍼(610)에 저장된 명령어의 개수가 적은 요청들을 우선적으로 지원한다. 이를 위하여 명령어 버퍼(610)들은 저장된 명령어들의 개수를 적절하게 인코딩하여 프리-펫치 압력 신호(Pre-fetch Pressure Signal)를 통해 캐시 제어기(640)에 전달한다. 캐시 제어기(640)는 분기 상태 신호와 프리-펫치 압력 신호들을 이용해 어떤 프로세서 코어(120)를 위해 명령어를 읽을 것인지를 결정한 후, 각 온 칩 메모리(622) 별로 온 칩 메모리 선택 신호를 보낸

다. 이에 따라 각 온 칩 메모리(622)로부터 적절한 명령어가 읽혀 해당 명령어 버퍼(610)에 삽입된다. 캐시 제어기(640)는 태그 매칭도 수행하는데 동시에 여러 개의 주소에 대한 태그 매칭이 필요한 경우 명령어 버퍼(610)에 명령어가 전혀 없는 순차적인 명령어 요청에 대한 태그 매칭을 가장 먼저 수행하고, 분기에 의한 명령어 요청에 대한 태그 매칭을 그 다음으로 수행하며, 그 외의 요청에 대한 태그 매칭을 맨 나중에 수행한다.

<127> 캐시 제어기(640)는 태그 매칭 결과 요청된 명령어가 인터리브 명령어 캐시 메모리(620)에 존재하지 않음을 확인한 경우, 명령어 블록 요청 신호를 통해 블록-필 제어기(650)에 이 사실을 통지한다. 블록-필 제어기(650)는 시스템 버스를 통해 외부 메모리로부터 명령어 블록을 읽어와 인터리브 명령어 캐시 메모리(620)에 저장한다.

<128> 프로세서 코어(120)의 원활한 구동을 위해서는 명령어 캐시와 함께 앞으로 읽거나 쓸 확률이 높은 데이터들을 온 칩 메모리에 저장해 둔 데이터 캐시가 필수적이다. 공유형 데이터 캐시(140)는 복수의 프로세서 코어(120)의 데이터 읽기 또는 쓰기 요청을 동시에 지원할 수 있는 데이터 캐시이다. 도 10을 참조하면, 공유형 데이터 캐시(140), 복수의 프로세서 코어(120), 인터리브 데이터 캐시 메모리(1020), 캐시 제어기(1040), 블록-필 제어기(1050), 히스토리 버퍼(1010), 히스토리 테스터(1015), 태그 메모리(1030a, 1030b, 1030c, 1030d, 이하 1030이라 통칭함)가 도시되어 있다.

<129> 데이터 접근은 크게 두 가지 유형으로 분류할 수 있다. 한 가지는 스택 메모리(Stack Memory) 접근이고 다른 하나는 힙 메모리(Heap Memory) 접근이다.

<130> 스택 메모리에는 지역 변수, 프로시저(혹은 함수) 호출시 서브-프로그램에 전달하는 인수들, 그리고 컴파일러가 레지스터 값을 대피시키기 위한 임시 메모리로 사용된다. 스택 메모리는 각 프로세서 코어(혹은 프로세서 코어 상에서 수행되는 작업)마다 고유한 영역이 지정된다. 일반적으로 약 50 ~ 70%의 데이터 접근이 스택 메모리 접근이다.

<131> 힙 메모리는 소프트웨어에 의하여 동적으로 할당되는 변수 혹은 배열 변수가 배치되는 영역이다. 한 소프트웨어 프로그램(혹은 한 작업)은 매우 많은 동적 변수들을 운용하지만, 짧은 순간에는 소수의 동적 변수들이 자주 이용된다.

<132> 즉, 스택 메모리의 경우 프로세서 코어의 스택 포인터 전후의 주소로 접근되는 빈도가 매우 높고, 힙 메모리의 경우에도 최근에 접근되었던 동적 변수가 다시 접근될 확률이 높다. 이를 고려하여 본 발명의 일 실시예에 따른 공유형 데이터 캐시(140)는 히스토리 기반 계층적 태그 매칭 방식를 적용한다.

<133> 히스토리 기반 계층적 태그 매칭 방식은 다음과 같다.

<134> 첫 번째 단계로, 최근에 접근되었던 스택 메모리와 힙 메모리의 태그를 히스토리 버퍼(1010)에 저장한다. 두 번째 단계로, 새로운 데이터 접근 요청이 있는 경우 히스토리 테스터(1015)는 일단 히스토리 버퍼(1010)에 저장된 태그와 요청된 주소를 비교하여 해당 데이터가 인터리브 데이터 캐시 메모리(1020)에 존재하는지를 확인한다. 세 번째 단계로, 만일 두 번째 단계에서 아직 판별이 끝나지 않은 경우 태그 메모리(1030)에서 태그를 읽어와 태그 매칭을 시도한다. 히스토리 버퍼(1010)는 각 프로세서 코어(120)마다 하나씩 존재하고, 상당수의 데이터 접근 요청이 두 번째 단계에서 캐시-히트임이 밝혀지기 때문에 태그 매칭 충돌이 현저하게 감소하는 효과가 있다.

<135> 히스토리 버퍼(1010)는 최근에 접근된 주소들의 태그가 다수 저장되어 있다. 다른 실시예에서는 응용에 따라 스택 메모리 접근의 빈도와 힙 메모리 접근의 빈도가 다르기 때문에, 히스토리 버퍼(1010)는 스택 메모리 접근과 힙 메모리 접근을 구별하여, 각각 다른 개수의 태그를 저장할 수도 있다. 데이터 접근 중에서 스택 메모리 접근과 힙 메모리 접근을 판별하는 방법 중 하나는 현재의 스택 포인터로부터 일정 범위 내의 데이터 접근은 스택 메모리 접근으로 간주하고, 그 이외의 것은 힙 메모리 접근으로 간주하면 면적 면에서 큰 오버헤드 없이 구현할 수 있다.

<136> 공유형 데이터 캐시(140)의 다른 구조는 공유형 명령어 캐시(130)와 기능이 유사한 바 상세한 설명은 생략한다.

<137> 한편, 공유형 명령어 캐시(130)의 구조가 도 6에 도시되어 있다. 2-경로 방식 세트 어소시에이티브 캐시의 예를 나타내고 있어 2개의 태그 메모리가 캐시 제어기(1040)에 연결되어 있다. 이는 하나의 실시예에 불과하며, 만일 공유형 명령어 캐시(130)가 4-경로 방식 세트 어소시에이티브 캐시인 경우에는 도 10에 도시된 공유형 데이터 캐시(140)와 같이 4개의 태그 메모리가 캐시 제어기(1040)에 연결될 수 있다.

<138> 또한, 본 발명의 또 다른 실시예에 따르면, 멀티 프로세서 시스템은 하드웨어 운영체제, 공유형 명령어 캐시와 공유형 데이터 캐시를 모두 포함한다. 하드웨어 운영체제를 통해 멀티 쓰레딩 오버헤드를 줄이고, 공유형 명령

어 캐시와 공유형 데이터 캐시를 통해 작업 이전 오버헤드와 시스템 면적을 줄일 수 있다.

<139> 상기에서는 본 발명의 바람직한 실시예를 참조하여 설명하였지만, 해당 기술 분야에서 통상의 지식을 가진 자라면 하기의 특허 청구의 범위에 기재된 본 발명의 사상 및 영역으로부터 벗어나지 않는 범위 내에서 본 발명을 다양하게 수정 및 변경시킬 수 있음을 이해할 수 있을 것이다.

도면의 간단한 설명

<140> 도 1은 본 발명의 일 실시예에 따른 멀티 프로세서 시스템의 구조를 나타낸 개략도.

<141> 도 2는 하드웨어 운영체제의 구조를 설명하는 상세도.

<142> 도 3은 쓰레드 제어 메모리의 구조를 설명하는 상세도.

<143> 도 4는 소프트웨어 운영체제에서의 멀티 쓰레드 프로그램 수행 과정 흐름도.

<144> 도 5는 본 발명의 일 실시예에 따른 하드웨어 운영체제에서의 멀티 쓰레드 프로그램 수행 과정 흐름도.

<145> 도 6은 공유형 명령어 캐시의 구조를 설명하는 상세도.

<146> 도 7은 메인 메모리의 데이터 배치 방식을 나타낸 도면.

<147> 도 8은 기존 캐시 메모리의 데이터 배치 방식을 나타낸 도면.

<148> 도 9는 본 발명의 일 실시예에 따른 인터리브 명령어 캐시 메모리의 데이터 배치 방식을 나타낸 도면.

<149> 도 10은 공유형 데이터 캐시의 구조를 설명하는 상세도.

<150> <도면의 주요부분에 대한 부호의 설명>

<151> 10 : 코프로세서 100 : 멀티 프로세서 시스템

<152> 110 : 하드웨어 운영체제 120 : 프로세서 코어

<153> 122 : 레지스터 파일 124 : 컨텍스트 제어기

<154> 130 : 공유형 명령어 캐시 140 : 공유형 데이터 캐시

<155> 210 : 주 제어기 220 : 쓰레드 제어 메모리

<156> 230 : 쓰레드 관리자 240 : 컨텍스트 메모리

<157> 250 : 컨텍스트 관리자 260 : 컨텍스트 버퍼

<158> 610 : 명령어 버퍼 620 : 인터리브 명령어 캐시 메모리

<159> 630 : 태그 메모리 640 : 캐시 제어기

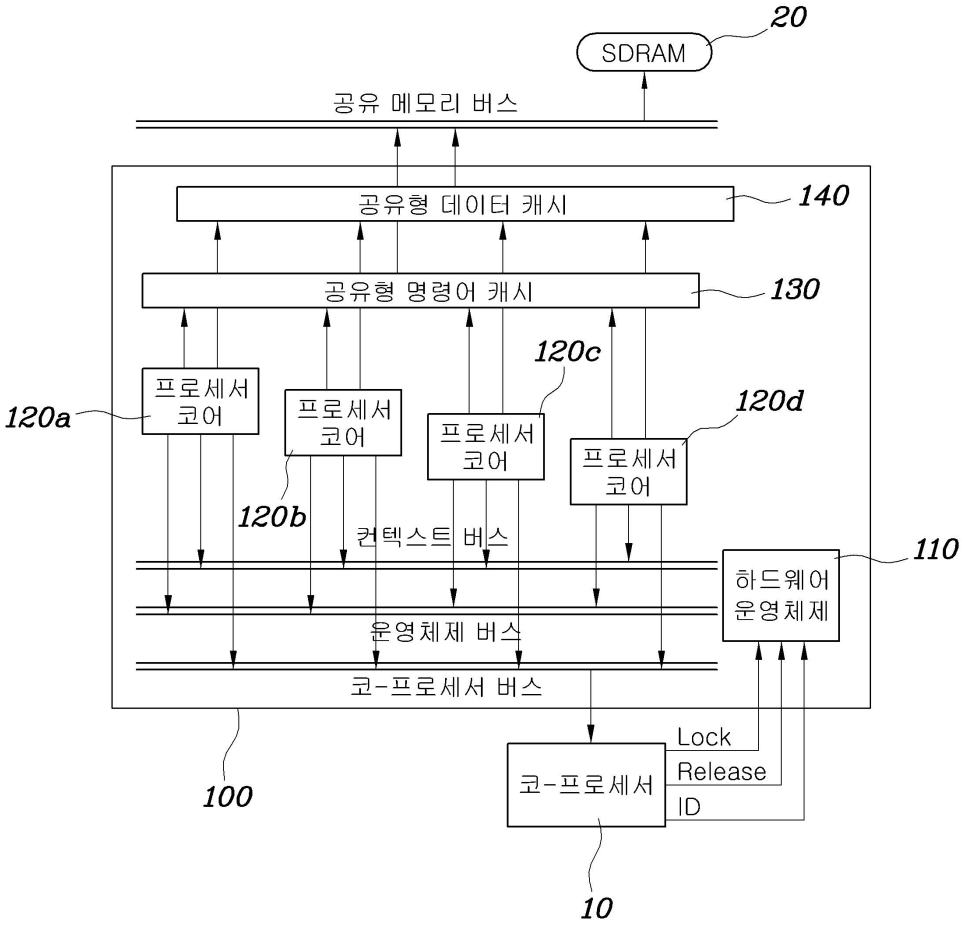
<160> 650 : 블록-필 제어기 1010 : 히스토리 버퍼

<161> 1020 : 인터리브 데이터 캐시 메모리 1030 : 태그 메모리

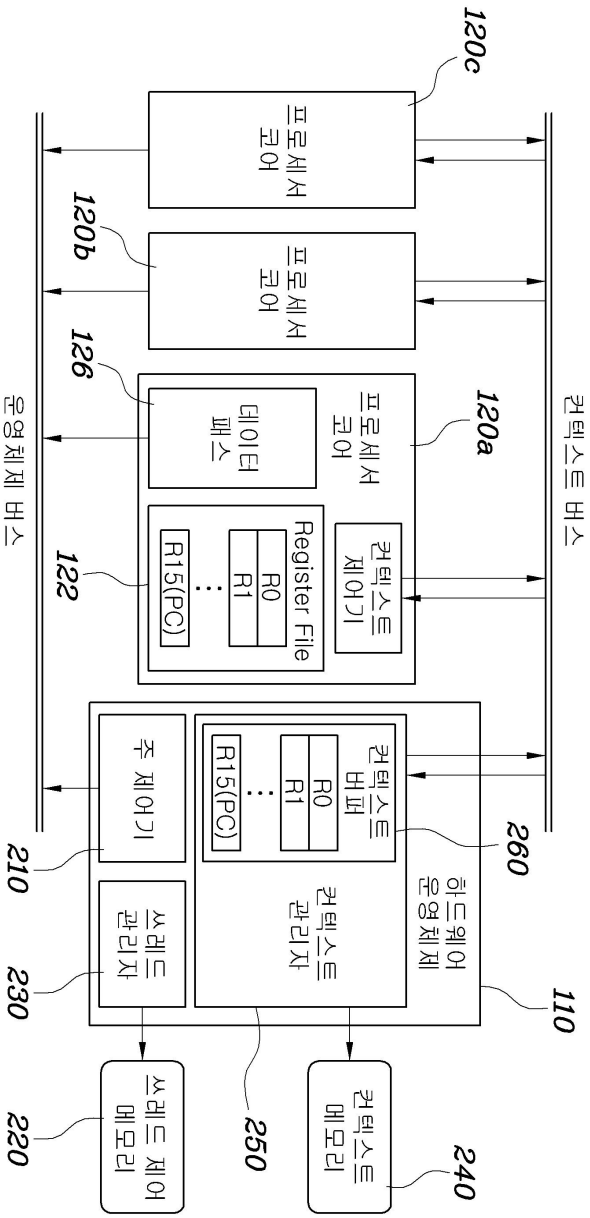
<162> 1040 : 캐시 제어기 1050 : 블록-필 제어기

도면

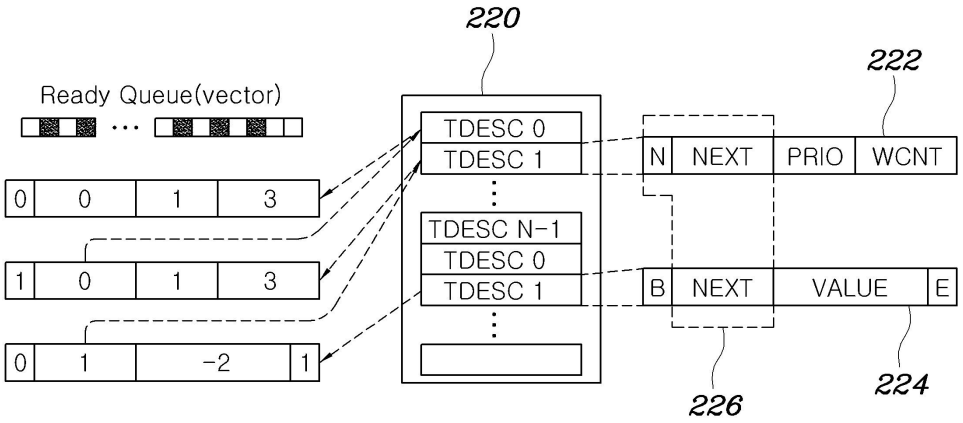
도면1


20
SDRAM
공유 메모리 버스
140
공유형 데이터 캐시
130
공유형 명령어 캐시
120c
120a
프로세서 코어
프로세서 코어
프로세서 코어
프로세서 코어
120d
120b
컨텍스트 버스
110
하드웨어 운영체제
운영체제 버스
코-프로세서 버스
Lock
Release
ID
코-프로세서
100
10

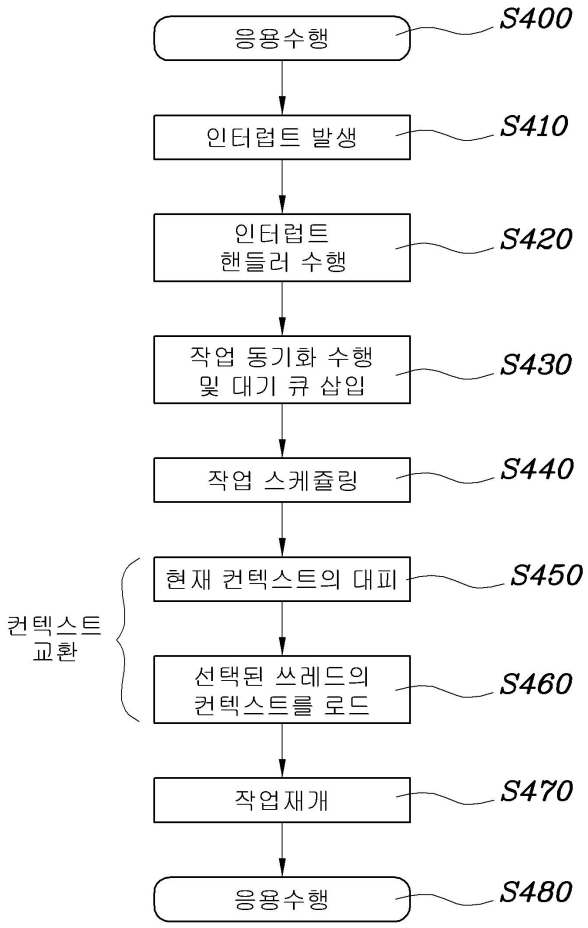
도면2


120c
프로세서 코어
120b
프로세서 코어
126
데이터 패스
120a
프로세서 코어
Register File
R0
R1
R15(PC)
122
컨텍스트 제어기
운영체제 버스
컨텍스트 버스
주 제어기
210
쓰레드 관리자
230
컨텍스트 버퍼
R0
R1
R15(PC)
260
컨텍스트 관리자
하드웨어 운영체제
110
쓰레드 제어 메모리
220
250
컨텍스트 메모리
240

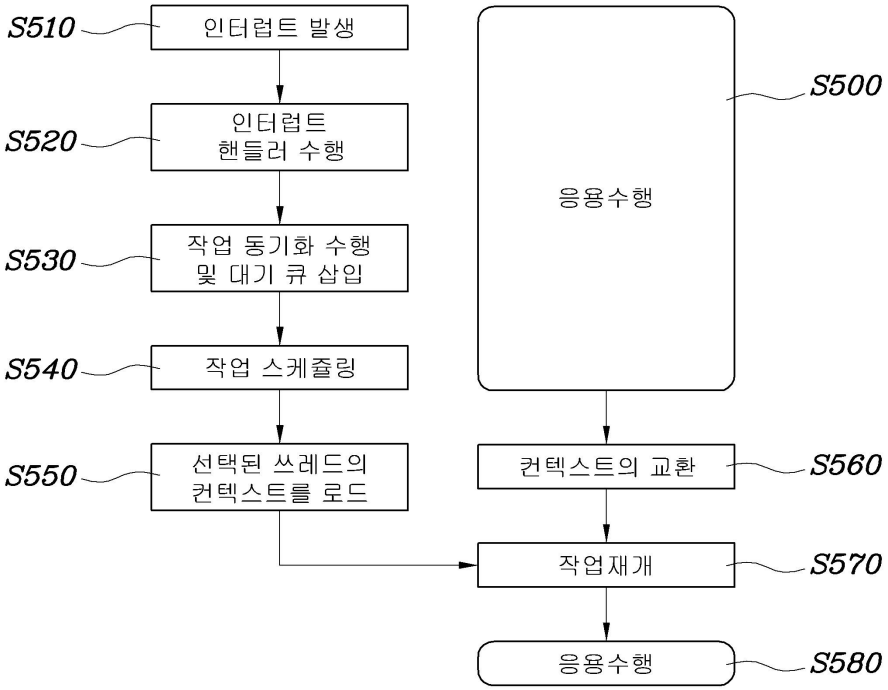
도면3


220
222
Ready Queue(vector)
TDESC 0
TDESC 1
TDESC N-1
TDESC 0
TDESC 1
N
NEXT
PRIO
WCNT
B
NEXT
VALUE
E
0 0 1 3
1 0 1 3
0 1 -2 1
226
224

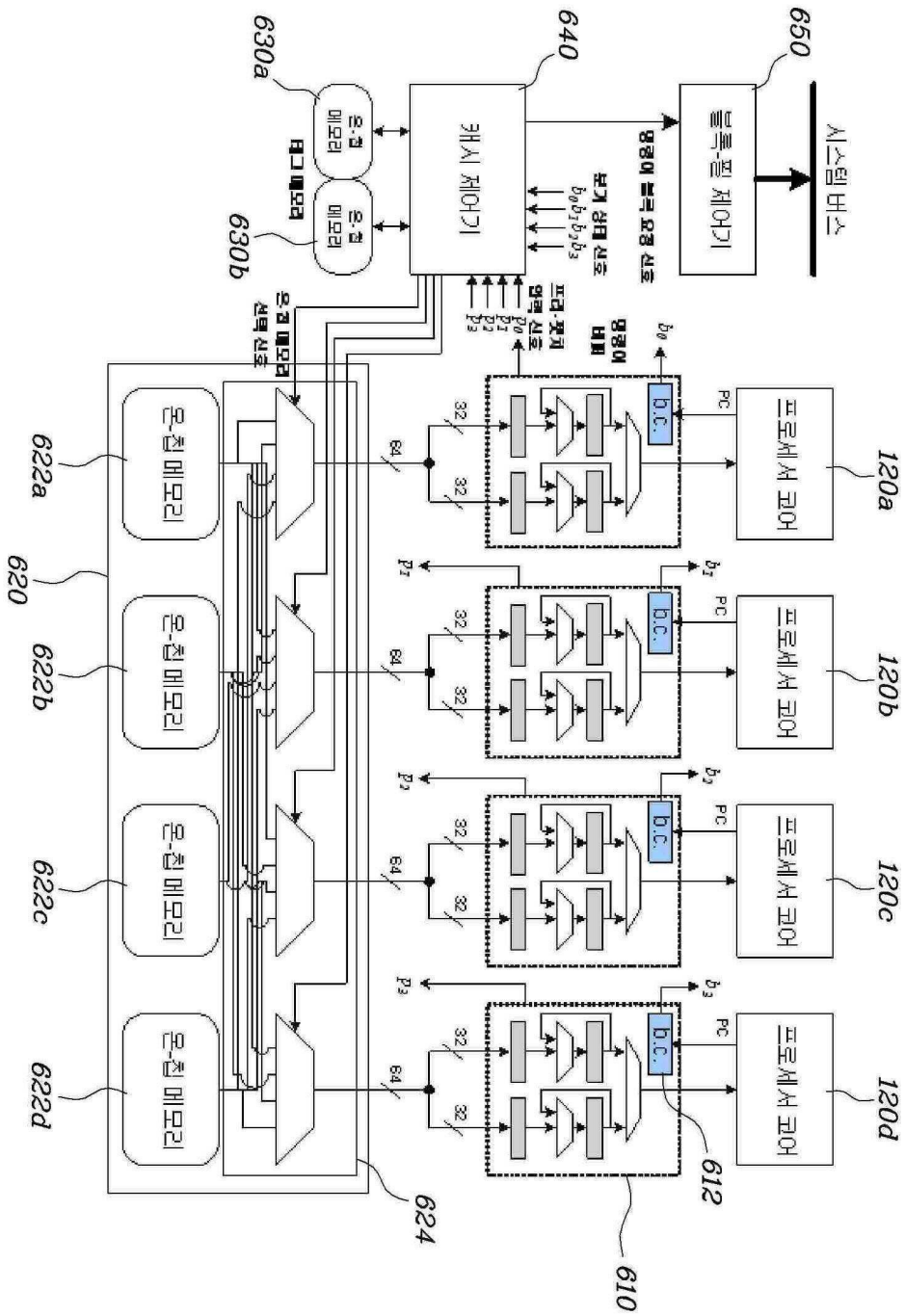
도면4


응용수행
S400
인터럽트 발생
S410
인터럽트
핸들러 수행
S420
작업 동기화 수행
및 대기 큐 삽입
S430
작업 스케쥴링
S440
현재 컨텍스트의 대피
S450
컨텍스트
교환
선택된 쓰레드의
컨텍스트를 로드
S460
작업재개
S470
응용수행
S480

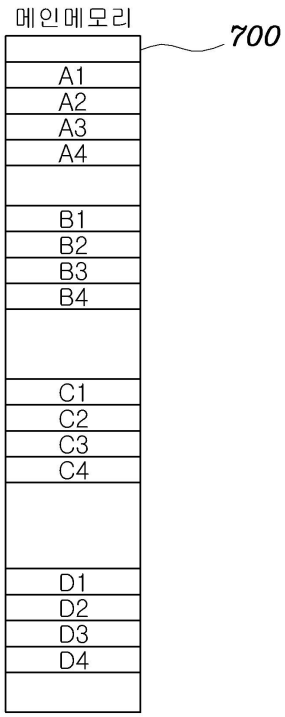
도면5


S510
인터럽트 발생
S520
인터럽트
핸들러 수행
S530
작업 동기화 수행
및 대기 큐 삽입
S540
작업 스케쥴링
S550
선택된 쓰레드의
컨텍스트를 로드
S500
응용수행
컨텍스트의 교환
S560
작업재개
S570
응용수행
S580

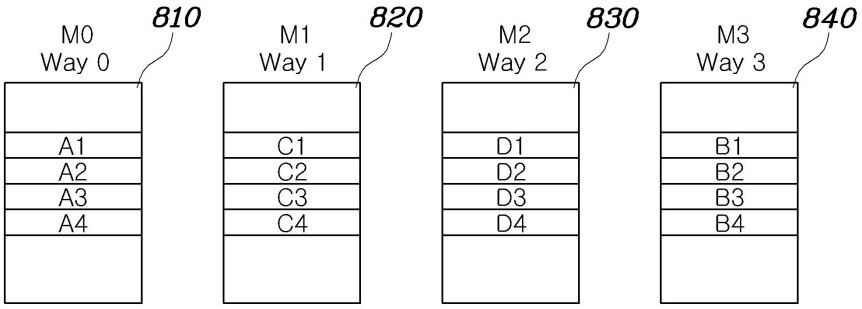
도면6


630a
640
650
630b
캐시 제어기
블록킹 제어기
시스템 버스
온-칩 메모리
프로세서 코어
120a
120b
120c
120d
622a
622b
622c
622d
620
624
610
612
b.c.
PC

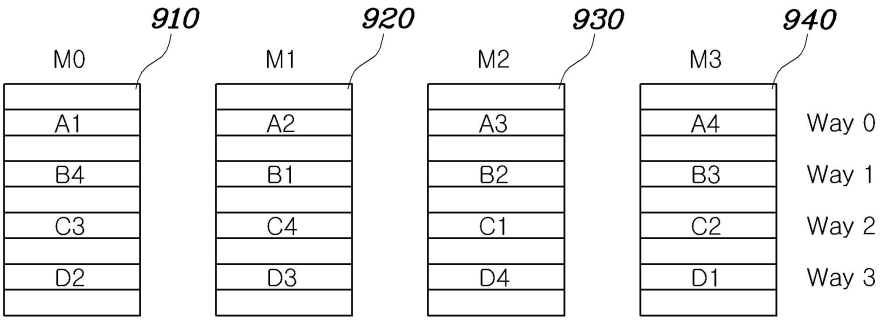
도면7


메인메모리
700
A1
A2
A3
A4
B1
B2
B3
B4
C1
C2
C3
C4
D1
D2
D3
D4

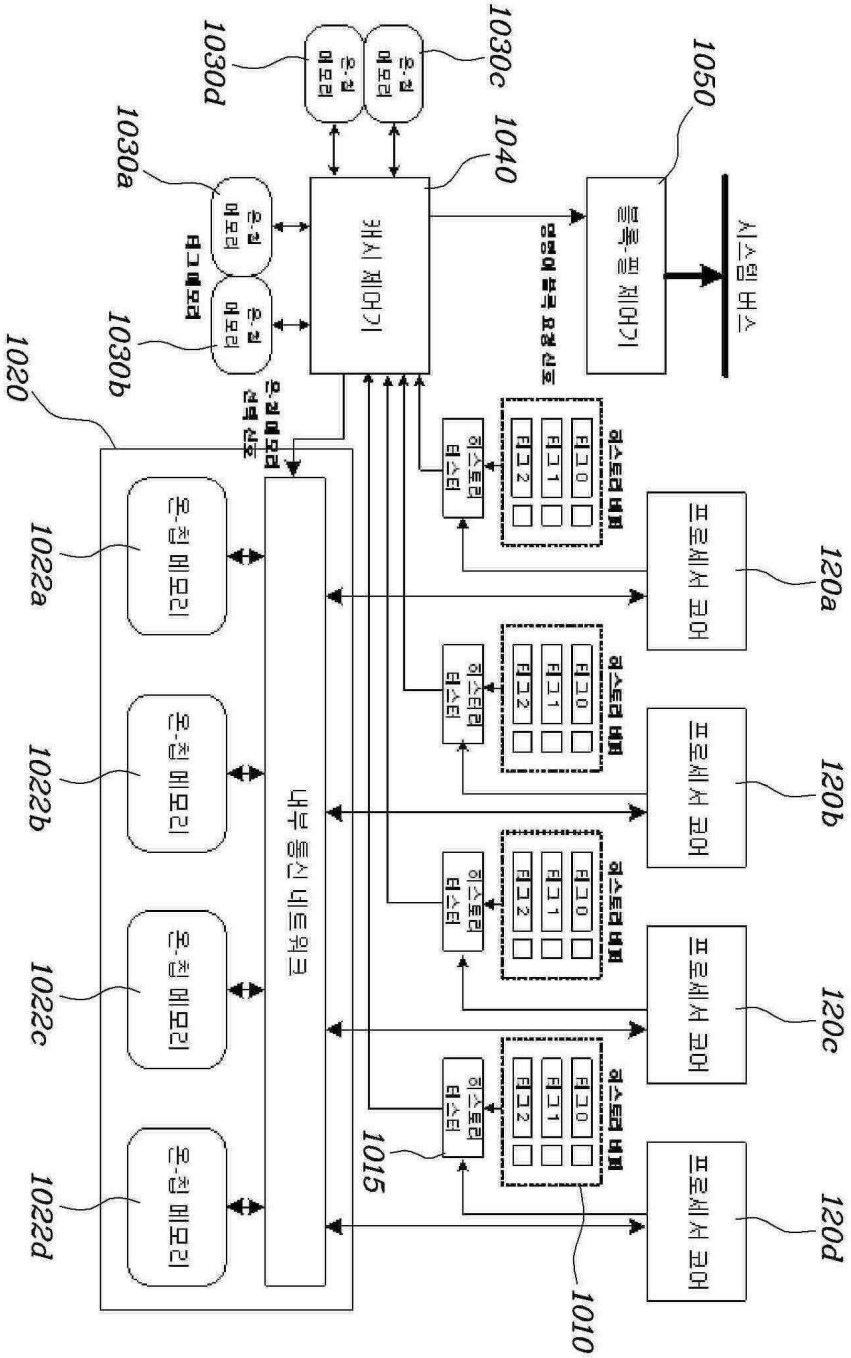
도면8


M0
Way 0
810
A1
A2
A3
A4
M1
Way 1
820
C1
C2
C3
C4
M2
Way 2
830
D1
D2
D3
D4
M3
Way 3
840
B1
B2
B3
B4

도면9


M0
910
A1
B4
C3
D2
M1
920
A2
B1
C4
D3
M2
930
A3
B2
C1
D4
M3
940
A4
B3
C2
D1
Way 0
Way 1
Way 2
Way 3

도면10


1050
1030c
1030d
1040
1030a
1030b
1020
1022a
1022b
1022c
1022d
1015
1010
120a
120b
120c
120d
시스템 버스
블록-킬 제어기
캐시 제어기
온-칩 메모리
내부 통신 네트워크
프로세서 코어
히스토리 테스터
히스토리 버퍼
태그0
태그1
태그2