(19) 대한민국특허청(KR)
(12) 등록특허공보(B1)

(45) 공고일자 2009년03월13일
(11) 등록번호 10-0888675
(24) 등록일자 2009년03월06일

(51) Int. Cl.
G06F 9/46 (2006.01) *G06F 9/28* (2006.01)
(21) 출원번호 10-2007-0016244
(22) 출원일자 2007년02월15일
심사청구일자 2007년02월15일
(65) 공개번호 10-2008-0076392
(43) 공개일자 2008년08월20일
(56) 선행기술조사문헌
KR1020070011434 A*
KR1020070116740 A
US20060150184 A1
KR1020070018066 A
*는 심사관에 의하여 인용된 문헌

(73) 특허권자
고려대학교 산학협력단
서울 성북구 안암동5가1 고려대학교 내
(72) 발명자
오재근
전북 전주시 덕진구 덕진동1가
뉴엔띠홍쨩
서울 성북구 안암동3가 16번지
(뒷면에 계속)
(74) 대리인
현종철

전체 청구항 수 : 총 4 항 심사관 : 윤혜숙

(54) 임베디드 코어에서 멀티 쓰레드 실행을 위해 확장된프로세서 및 임베디드 코어에서 멀티 쓰레드 실행 방법

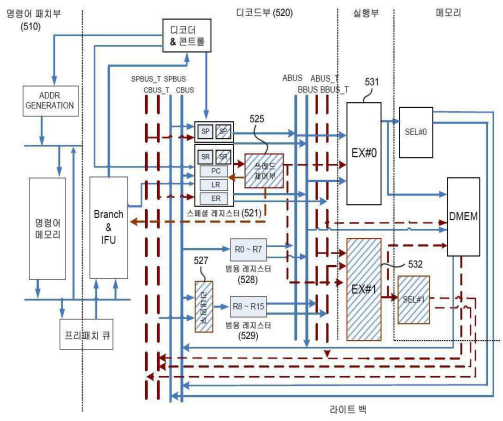
(57) 요 약

임베디드 코어에서 멀티 쓰레드 실행을 위해 확장된 프로세서 및 임베디드 코어에서 멀티 쓰레드 실행 방법이 개시된다.

본 발명은 멀티 쓰레드로 제작된 코드를 명령어 메모리로부터 인출하는 명령어 패치부, 상기 인출된 코드들을 디코드하고, 상기 인출된 코드들의 실행 순서를 설정하는 디코드부, 상기 설정된 실행 순서에 따라 각각의 쓰레드에 대한 코드를 실행하는 복수의 실행부, 상기 디코드부를 위한 명령어 정보를 저장하는 스페셜 레지스터 및 시리얼 코드 영역에서는 전체 영역을 사용하고, 패러럴 코드 영역에서는 각각의 쓰레드를 위해 분할된 영역을 사용하는 범용 레지스터를 포함한다.

본 발명에 의하면, CMP의 성능을 유지하는 상태에서 전력 소모량을 줄이고, 프로세서의 면적을 줄여 제조가 용이하다.

대 표 도 - 도5



(72) 발명자

황석중

경기 파주시 아동동 357-1

김아름

경기 구리시 인창동 개나리아파트 6동 203호

김선욱

경기 남양주시 도농동 부영아파트2단지 203동 1604호

특허청구의 범위

청구항 1

삭제

청구항 2

삭제

청구항 3

삭제

청구항 4

삭제

청구항 5

멀티 쓰레드로 제작된 코드를 명령어 메모리로부터 인출하는 명령어 패치부;

상기 인출된 코드들을 디코드하고, 상기 인출된 코드들의 실행 순서를 설정하는 디코드부;

상기 설정된 실행 순서에 따라 각각의 쓰레드에 대한 코드를 실행하는 복수의 실행부;

상기 디코드부를 위한 명령어 정보를 저장하는 스패셜 레지스터; 및

시리얼 코드 영역에서는 전체 영역이 사용되고, 패러럴 코드 영역에서는 각각의 쓰레드를 위해 분할된 영역이 사용되는 범용 레지스터를 포함하고,

상기 복수의 실행부는

싱글 포트 캐쉬를 이용하고, 2개 이상의 쓰레드가 동시에 상기 싱글 포트 캐쉬에 접근하는 경우, 이웃한 어드레스에 대한 접근을 한번의 접근으로 처리하는 더블링 데이터 캐쉬 인터페이스를 포함하는 임베디드 코어에서 멀티 쓰레드 실행을 위해 확장된 프로세서.

청구항 6

제 5 항에 있어서,

상기 디코드부는

2개 이상의 쓰레드에서 컨트롤 플로우를 분기시키는 명령어가 존재하는 경우 각각의 쓰레드 간의 순서를 설정하는 쓰레드 제어부를 포함하는 것을 특징으로 하는 임베디드 코어에서 멀티 쓰레드 실행을 위해 확장된 프로세서.

청구항 7

삭제

청구항 8

삭제

청구항 9

제 5 항에 있어서,

상기 더블링 데이터 캐쉬 인터페이스는

2개의 32비트 캐쉬 스토어 접근을 1개의 64비트 캐쉬 스토어 접근으로 합치는 컴바이너; 및

상기 싱글 포트 캐쉬로부터 1개의 64비트 캐쉬 로드 접근을 수행한 후 2개의 32비트 캐쉬 로드 접근에 대한 출

력값을 생성하는 복수의 멀티플렉서를 포함하는 임베디드 코어에서 멀티 쓰레드 실행을 위해 확장된 프로세서.

청구항 10

삭제

청구항 11

멀티 쓰레드로 제작된 코드를 명령어 메모리로부터 인출하는 단계;

명령어 정보를 저장하는 스패셜 레지스터를 이용하여 상기 인출된 코드들을 디코드하고, 상기 인출된 코드들에 2개 이상의 쓰레드에서 컨트롤 플로우를 분기시키는 명령어가 존재하는 경우 각각의 쓰레드 간의 순서를 설정하는 단계; 및

싱글 포트 캐쉬를 이용하는 복수의 실행부를 이용하여 상기 설정된 실행 순서에 따라 상기 각각의 쓰레드에 대한 코드를 실행하고, 2개 이상의 쓰레드가 동시에 상기 싱글 포트 캐쉬에 접근하는 경우, 이웃한 어드레스에 대한 접근을 한번의 접근으로 처리하는 단계를 포함하는 임베디드 코어에서 멀티 쓰레드 실행 방법.

명 세 서

발명의 상세한 설명

발명의 목적

발명이 속하는 기술 및 그 분야의 종래기술

<11> 본 발명은 임베디드 프로세서에 관한 것으로, 특히, 임베디드 코어에서 멀티 쓰레드 실행을 위해 확장된 프로세서 및 임베디드 코어에서 멀티 쓰레드 실행 방법에 관한 것이다.

<12> 임베디드 프로세서는 일반적인 데이터 처리용 프로세서와 비교하여 보았을 때 저가, 저전력이 중요한 부분을 차지한다. 수행 프로그램이 ROM의 형태로 저장된다. ROM에 저장되는 내장형 프로세서 분야에서 수행 프로그램의 크기를 효과적으로 줄임으로 다이(die)에 있어서 많은 크기를 차지하는 ROM의 크기를 줄일 수 있으므로 좀더 저가의 내장형 시스템을 공급할 수 있다.

<13> EISC(Extendable Instruction Set Computer)는 수행 프로그램의 크기 및 메모리 접근의 회수를 매우 효과적으로 줄일 수 있도록 설계된 명령어 셋으로서, 확장 명령어를 이용하여 명령어의 즉치 값 및 변위 등의 값을 자유롭게 확장할 수 있는 형태를 지닌 아키텍쳐로서 종래의 CISC와 RISC의 장점을 합친 형태를 취하고 있다.

<14> EISC는 기본적으로 RISC가 갖고 있는 간단한 구조의 하드웨어를 취하면서도 CISC의 장점을 추가하여 높은 성능을 갖게 하였고, 코드 밀도가 높아 종래의 RISC 프로세서와 비교하여 약 60%, CISC 프로세서와 비교하여 약 80% 정도로 프로그램 크기가 작다는 장점을 지니고 있다. 따라서, 코드 밀도가 중시되는 내장형 응용 분야에 있어서 강점을 지니고 있다.

<15> 종래의 CISC (Complex Instruction Set Computer) 기반의 마이크로 프로세서는 코드 밀도에 있어서 장점이 있으나 고속화에 문제점을 지니고 있으며, RISC(Reduce Instruction Set Computer)기반의 마이크로 프로세서는 대부분 32-bit 크기의 명령어를 지님으로 프로그램 코드의 크기가 지나치게 커진다는 단점을 지니고 있다. EISC 기반의 32-bit 마이크로 프로세서인 AE32000은 16-bit 크기의 명령어를 이용하여 32-bit 데이터를 효과적으로 처리할 수 있도록 설계되었다.

<16> 이는 ARM사의 Thumb계열의 프로세서나 MIPS16에서도 채택하고 있는 방법이나, EISC 아키텍쳐의 경우 16-bit 크기의 명령어 사용으로 인한 즉치 값(immediate value) 공간의 부족을 이전의 접근 방식과는 다르게 독립적으로 사용 가능한 LERI(Load Extension Register with Immediate data) 명령을 이용하는 것으로 해결하였다.

<17> LERI명령은 2비트 Op-Code와 14비트 즉치 값을 가지는 명령으로서 ER(Extension Register)에 즉치 값을 저장하고, 이후에 즉치 값의 확장이 필요한 명령에서 ER의 값을 인출하여 해당 명령어의 즉치 값과 덧붙이는 구조를 지니고 있다. 이러한 구조의 장점은 짧은 즉치 값 크기로 인하여 발생 가능한 문제를 효과적으로 해결 할 수 있다는 것이나, LERI의 부가로 인하여 코드의 길이가 길어질 수 있다는 문제와 성능의 저하를 가져 올 수 있다는 문제를 지니고 있다. 따라서, EISC기반의 프로세서는 이 LERI를 효과적으로 처리하는 것이 중요하다.

<18> AE32000은 EISC기반의 32-bit 마이크로 프로세서로서 Low-End용 32-bit 마이크로 프로세서인 SE3208의 특징에 부가적으로 더 깊은 파이프라인과 OSI(On-Silicon In Circuit Emulator) 지원 기능, 캐쉬 및 가상 메모리의 지원기능, LERI 명령어 폴딩 기능등을 추가하여 성능을 향상시킨 프로세서이다.

<19> 그러나, 종래의 멀티 쓰레드 환경에서의 코드 생성에 대한 프로세서는 2개의 코어를 가진 CMP(Chip Multi Processor)로 전력 소모량이 커서 임베디드 시스템에 적합하지 못하고, 면적이 크며, 제조가 용이하지 않은 문제점이 있다.

발명이 이루고자 하는 기술적 과제

<20> 따라서, 본 발명이 이루고자 하는 첫번째 기술적 과제는 CMP의 성능을 유지하는 상태에서 전력 소모량을 줄이고, 면적을 줄여 제조가 용이한 임베디드 코어에서 멀티 쓰레드 실행을 위해 확장된 프로세서를 제공하는 데 있다.

<21> 본 발명이 이루고자 하는 두번째 기술적 과제는 상기의 임베디드 코어에서 멀티 쓰레드 실행을 위해 확장된 프로세서에 적용된 임베디드 코어에서 멀티 쓰레드 실행 방법을 제공하는 데 있다.

발명의 구성 및 작용

<22> 상기의 첫번째 기술적 과제를 이루기 위하여, 본 발명은 멀티 쓰레드로 제작된 코드를 명령어 메모리로부터 인출하는 명령어 패치부, 상기 인출된 코드들을 디코드하고, 상기 인출된 코드들의 실행 순서를 설정하는 디코드부, 상기 설정된 실행 순서에 따라 각각의 쓰레드에 대한 코드를 실행하는 복수의 실행부, 상기 디코드부를 위한 명령어 정보를 저장하는 스패셜 레지스터 및 시리얼 코드 영역에서는 전체 영역이 사용되고, 패러럴 코드 영역에서는 각각의 쓰레드를 위해 분할된 영역이 사용되는 범용 레지스터를 포함하는 임베디드 코어에서 멀티 쓰레드 실행을 위해 확장된 프로세서를 제공한다.

<23> 또한, 상기의 첫번째 기술적 과제를 이루기 위하여, 본 발명은 멀티 쓰레드로 제작된 코드를 명령어 메모리로부터 인출하는 명령어 패치부, 상기 인출된 코드들을 디코드하고, 상기 인출된 코드들의 실행 순서를 설정하는 디코드부, 상기 설정된 실행 순서에 따라 각각의 쓰레드에 대한 코드를 실행하는 복수의 실행부, 상기 디코드부를 위한 명령어 정보를 저장하는 스패셜 레지스터 및 시리얼 코드 영역에서는 전체 영역이 사용되고, 패러럴 코드 영역에서는 각각의 쓰레드를 위해 분할된 영역이 사용되는 범용 레지스터를 포함하고, 상기 복수의 실행부는 싱글 포트 캐쉬를 이용하고, 2개 이상의 쓰레드가 동시에 상기 싱글 포트 캐쉬에 접근하는 경우, 이웃한 어드레스에 대한 접근을 한번의 접근으로 처리하는 더블링 데이터 캐쉬 인터페이스를 포함하는 임베디드 코어에서 멀티 쓰레드 실행을 위해 확장된 프로세서를 제공한다.

<24> 한편, 상기의 두번째 기술적 과제를 이루기 위하여, 본 발명은 멀티 쓰레드로 제작된 코드를 명령어 메모리로부터 인출하는 단계, 명령어 정보를 저장하는 스패셜 레지스터를 이용하여 상기 인출된 코드들을 디코드하고, 상기 인출된 코드들에 2개 이상의 쓰레드에서 컨트롤 플로우를 분기시키는 명령어가 존재하는 경우 각각의 쓰레드 간의 순서를 설정하는 단계, 및 싱글 포트 캐쉬를 이용하는 복수의 실행부를 이용하여 상기 설정된 실행 순서에 따라 상기 각각의 쓰레드에 대한 코드를 실행하고, 2개 이상의 쓰레드가 동시에 상기 싱글 포트 캐쉬에 접근하는 경우, 이웃한 어드레스에 대한 접근을 한번의 접근으로 처리하는 단계를 포함하는 임베디드 코어에서 멀티 쓰레드 실행 방법을 제공한다.

<25> 삭제

<26> 이하에서는 도면을 참조하여 본 발명의 바람직한 실시예를 설명하기로 한다. 그러나, 다음에 예시하는 본 발명의 실시예는 여러 가지 다른 형태로 변형될 수 있으며, 본 발명의 범위가 다음에 상술하는 실시예에 한정되는 것은 아니다.

<27> 도 1은 본 발명의 위치를 기술 트리로 나타낸 것이다.

<28> 본 발명의 동기는 다음의 세 가지를 들 수 있다. 첫 번째로 임베디드 어플리케이션은 높은 병렬성을 가지고, 두 번째로 이 병렬성을 가진 코드가 전체 실행시간의 대부분을 차지하며 그리고 마지막으로 쓰레드들은 대부분의 경우 같은 코드를 실행한다는 것이다.

<29> 도 2a는 본 발명이 적용되는 임베디드 어플리케이션에서 같은 코드가 실행되는 경우와 컨트롤 플로우가 달라지

는 경우의 스피드업을 도시한 것이다.

<30> 도 2a 은 Amdahls's Law를 사용하여 계산된 스피드업(speedup)이다. 동일함(Same) 의 경우 전체 실행시간에서 같은 코드 영역이 실행되는 경우의 스피드업이고 다름(Different)의 경우에는 if-else/switch 와 같이 콘트롤 플로우가 달라질 수 있을 경우를 의미한다. 도 2a를 통해 conven 과 viterb 를 제외한 대부분의 벤치마크 프로그램에서 같은 코드 영역을 실행하는 것을 볼 수 있다.

<31> 도 2b는 본 발명이 적용되는 임베디드 어플리케이션에서 시리얼과 패러럴 영역에서 실제 실행되는 명령어의 분포를 도시한 것이다.

<32> 도 2b는 각 벤치마크 프로그램에서 명령어가 패러럴 영역에 속하는지 시리얼 코드 영역에 속하는 지를 보여주는 분포이다. 그래프에서 병렬-브랜치(*P_branch*)에 해당하는 명령어가 매우 적은 것을 볼 수 있으며 컨트롤 플로우가 달라질 가능성 또한 매우 작다는 것을 알 수 있다.

<33> 도 3은 본 발명의 일 실시 예에 따른 임베디드 코어에서 멀티 쓰레드 실행을 위해 확장된 프로세서의 블럭도이다.

<34> 명령어 패치부(310)는 멀티 쓰레드로 제작된 코드를 명령어 메모리로부터 인출한다.

<35> 디코드부(320)는 인출된 코드들을 디코드하고, 인출된 코드들의 실행 순서를 설정한다.

<36> 복수의 실행부(331, 332)는 디코드부(320)에서 설정된 실행 순서에 따라 각각의 쓰레드에 대한 코드를 실행한다.

<37> 스패셜 레지스터(321)는 디코드부(320)를 위한 명령어 정보를 저장한다. 구성의 일 예로, 스패셜 레지스터(321)는 복수개로 구성될 수 있다.

<38> 범용 레지스터(329)는 시리얼 코드 영역에서는 전체 영역을 사용하고, 패러럴 코드 영역에서는 각각의 쓰레드를 위해 분할된 영역을 사용한다. 이때, 패러럴 코드 영역에서 복수의 실행부(331, 332)가 범용 레지스터를 2개 이상으로 분할된 영역으로 인식하게 하는 리네임부가 범용 레지스터(329)를 제어할 수 있다.

<39> 도 4는 본 발명의 다른 실시 예에 따른 임베디드 코어에서 멀티 쓰레드 실행을 위해 확장된 프로세서의 블럭도이다.

<40> 명령어 패치부(410)는 멀티 쓰레드로 제작된 코드를 명령어 메모리로부터 인출한다.

<41> 디코드부(420)는 인출된 코드들을 디코드하고, 인출된 코드들의 실행 순서를 설정한다.

<42> 쓰레드 제어부(425)는 2개 이상의 쓰레드에서 컨트롤 플로우를 분기시키는 명령어가 존재하는 경우 각각의 쓰레드의 순서를 설정하는 를 포함할 수 있다. 이때, 디코드부(420)는 병렬 실행을 시작, 병렬 실행을 종료, 변수에 락을 걸어서 다른 쓰레드가 사용하지 못하게 함, 변수의 락을 해제, 2개 이상의 쓰레드가 다른 컨트롤 플로우일 때 동기화, 쓰레드의 아이디를 제공을 포함하는 명령어 셋을 저장하고, 명령어 셋을 이용하여 인출된 코드들을 해석할 수 있다.

<43> 복수의 실행부(431, 432)는 디코드부(420)에서 설정된 실행 순서에 따라 각각의 쓰레드에 대한 코드를 실행한다. 이때, 복수의 실행부(431, 432)는 도 6과 같은 더블링 데이터 캐쉬 인터페이스를 포함할 수 있다.

<44> 스패셜 레지스터(421)는 디코드부(420)를 위한 명령어 정보를 저장한다. 구성의 일 예로, 스패셜 레지스터(421)는 복수개로 구성될 수 있다.

<45> 범용 레지스터(429)는 시리얼 코드 영역에서는 전체 영역을 사용하고, 패러럴 코드 영역에서는 각각의 쓰레드를 위해 분할된 영역을 사용한다. 이때, 패러럴 코드 영역에서 복수의 실행부(431, 432)가 범용 레지스터를 2개 이상으로 분할된 영역으로 인식하게 하는 리네임부가 범용 레지스터(429)를 제어할 수 있다.

<46> 도 5는 본 발명의 또다른 실시 예에 따른 임베디드 코어에서 멀티 쓰레드 실행을 위해 확장된 프로세서의 블럭도이다.

<47> 임베디드 코어에서 멀티 쓰레드 실행을 하기 위해서는 CMP와 같은 멀티 프로세서가 필요하다. 이를 위해서 AE32000C 코어를 확장하여 멀티 쓰레드 환경을 구현한다. AE32000C 코어는 임베디드 코어로 임베디드 오디오, 비디오 등 다양한 분야에서 사용된다.

<48> AE32000C 프로세서는 기본적으로 하나의 코어만을 갖는다. 그리고 한 프로그램을 멀티 쓰레드 코드로 만들

경우, 2개의 쓰레드가 수행될 때 2개의 쓰레드가 대부분의 경우 같은 코드를 수행한다.

<49> AE32000C 코어는 fetch, decode, execution, mem 그리고 WB(write back) 의 5개 단계로 나누어진 파이프 라인 구조를 갖는다.

<50> 첫번째, 명령어 인출(Instruction Fetch, IF stage) 단계이다. 명령어 메모리에 접근하여 명령어를 가지고 온다. backend 파이프라인의 동작과 무관하게 명령어 큐(도 5의 프리패치 큐)가 찰 때까지 선인출(prefetch)을 수행한다. 또한, LERI명령으로 인하여 발생할 수 있는 성능 저하를 최소화하기 위하여 명령어 폴딩을 수행한다.

<51> 두번째, 명령어 디코드(Instruction Decode/Operand Fetch, ID/OF) 단계이다. 인출해 온 명령어에 대한 해석을 수행하며, 명령어 해석에 따라 파이프라인의 각 데이터 패스 요소에 필요한 제어 신호를 생성하고, 필요한 오퍼랜드를 인출한다.

<52> ID 스테이지에서는 대부분의 제어가 수행되며, 특히 인터럽트에 대한 반응을 수행하여 파이프를 비우고 제어 상태 머신으로 전환하는 것이나, 분기를 수행하는 등의 동작이 수행된다. 오퍼랜드 인출(operand fetch)의 경우에는 데이터간의 의존성 및 하드웨어 의존성등 점검하여 피할 수 없는 의존성의 경우 버블 스테이지(bubble stage)를 발생시키며, 피할 수 있는 경우에는 포워딩(forwarding)으로서 의존성 문제를 해결하는 동작 또한 수행된다. 또한, ID스테이지에서는 보조 프로세서와의 제어가 수행된다.

<53> 세번째, 실행(Exection, EX) 단계이다. 실행 단계에서는 프로세서에서 수행하는 대부분의 연산을 처리하며, 데이터 접근을 위한 주소를 생성하는 작업을 수행한다. 일반적인 환경에서 사용되는 MAC연산을 포함한 모든 연산은 1 사이클(cycle) 내에 연산이 가능하나, DSP Extension중 SIMD Extension된 상태의 MAC연산은 pipelined SIMD MAC으로 서 구현되어 있으므로, 같은 스루풋(throughput)을 지니지만 레이턴시(latency)에 차이가 있다.

<54> 네번째, 메모리 접근(Memory Access, MEM) 단계이다. 데이터 메모리에 접근하여 데이터를 READ/WRITE하는 동작을 수행한다. 외부 데이터/캐쉬의 레이턴시가 있는 경우에 프로세서는 in-order 파이프라인을 채용하고 있으므로 파이프라인이 정지되어야 한다.

<55> 다섯번째, 라이트 백(Write-Back, WB) 단계이다. 연산의 결과 혹은 메모리 접근을 통하여 취득한 새로운 데이터를 레지스터 파일에 반영하는 단계이다.

<56> 도 5에서, 실제 멀티 쓰레드가 실행되는 영역에서 대부분의 같은 코드 영역이 실행되기 때문에 명령어 패치부(510, Fetch Unit) 및 디코드부(Decode)는 하나를 사용하고, 실행부(531, 532, Execution), 메모리(Memory) 및 라이트 백(Write back) 유닛의 경우 멀티 쓰레드를 위하여 하나씩 더 추가된다.

<57> 도 5의 코어는 16개의 범용 레지스터(528, 529)를 갖는다. 일반적인 멀티 쓰레드 프로그램에서는 2개의 쓰레드를 지원하기 위해 원래의 레지스터의 두 배의 레지스터(32개)를 갖는다. 그러나, 도 5에서는 전력 소모와 리소스를 줄이기 위하여 레지스터 파일을 추가하지 않고 16개의 레지스터를 8개씩(528, 529) 둘로 나누어서 사용한다. 이때, 리네임부(527)가 16개의 레지스터를 8개씩 둘로 나눈다. 리네임부(527)는 도 5와 같이, 추가된 쓰레드의 앞에 위치하여 명령어 레지스터 오퍼랜드(instruction register operands)를 r0-r7에서 r8-r15로 변경한다.

<58> 즉, 시리얼(Serial) 코드 영역에서는 16개의 레지스터를 모두 사용하고 패러럴(Parallel) 코드 영역에서는 각 쓰레드가 8개의 레지스터를 사용한다.

<59> 스패셜 레지스터(521, Special Purpose Register, SPR)의 경우에는 각 쓰레드를 위해 레지스터 파일을 추가한다.

<60> 디코드부(520, Decode stage)에서는 패러럴 코드 영역에서 2개의 쓰레드를 제어하기 위해 쓰레드 제어부(525, Thread control unit)를 추가해서 if-else와 같이 2개의 쓰레드에서 컨트롤 플로우가 달라질 경우를 제어한다.

<61> 도 6은 도 5에 적용되는 더블링 데이터 캐쉬 인터페이스의 블럭도이다.

<62> 멀티 쓰레드를 지원하기 위해서는 데이터 캐쉬가 듀얼 포트로 접근이 가능해야 한다. 그러나, 본 발명에서는 코어 면적 및 전력 소모를 줄이기 위해 듀얼 포트를 사용하지 않고 코어와 데이터 캐쉬 사이에 인터페이스 유닛을 추가한다. 패러럴 코드 영역을 실행할 때, 캐쉬에서 데이터를 읽거나 쓸 때, 각 쓰레드가 순차적으로 접근할 수 있도록 제어한다. 이 경우 듀얼 포트 캐쉬를 사용할 때보다 메모리 접근 부하가 커지지만 인터페이스를 이용하여 순차적으로 캐쉬에 접근하기 때문에 전력 소모와 캐쉬의 크기를 줄일 수 있다는 장점이 있다.

<63> 확장된 코어에서는 싱글 포트 캐쉬를 사용하기 때문에 load/sotre가 많이 발생할 경우에 성능이 많이 저하될 수 있다. 이런 싱글 포트 캐쉬의 문제점을 해결하기 위해서 도 6과 같은 더블링(Doubling) 데이터 캐쉬 인터페이스를 사용한다. 멀티 쓰레드 프로그램에서 두 개의 쓰레드가 동시에 load/store를 할 경우에 두 쓰레드에서 접근하려는 어드레스가 인접해 있을 가능성이 크다. 이 경우 각 요청을 여러 번의 캐쉬 접근으로 처리하는 것이 아니라 한 번의 64bit 접근으로 load/sotre를 수행할 수 있다.

<64> 즉, 더블링 데이터 캐쉬 인터페이스는 2개의 32비트 캐쉬 스토어 접근을 1개의 64비트 캐쉬 스토어 접근으로 합치는 컴바이너(548) 및 싱글 포트 캐쉬로부터 1개의 64비트 캐쉬 로드 접근을 수행한 후 2개의 32비트 캐쉬 로드 접근에 대한 출력값을 생성하는 복수의 멀티플렉서(541, 542)를 포함한다.

<65> 병렬 실행에서, 컨트롤러(549)는 각 쓰레드의 접근 주소(ADR0, ADR1)를 비교하여 캐쉬에 접근하는 방법을 결정한다. 접근하려는 데이터들이 동일한 캐쉬 라인에 존재하면, 메모리 컨트롤러는 한번의 메모리 접근에서 캐쉬로부터 로드된 전체 쓰레드의 데이터(LD0, LD1)를 이용하여 각각의 쓰레드에 데이터를 분배하거나, 제공되는 데이터(ST0, ST1)을 합쳐서 두 배 크기의 데이터를 캐쉬에 저장한다. 접근하려는 데이터들이 동일한 캐쉬 라인에 존재하지 않으면, 두번의 메모리 접근이 인터리브드 방식(interleaved manner)으로 순차적으로 핸들된다. 버퍼는 데이터 인터리빙(data interleaving)을 위해 사용된다. 직렬 실행에서는 컨트롤러(549)는 단지 모든 캐쉬 신호를 액티브한 쓰레드에 중계한다.

<66> 멀티 쓰레드 프로그램을 구현하기 위하여 하드웨어를 추가하는 것 외에 컴파일러를 확장한다. 각각의 쓰레드는 자신의 ID(identification number)를 참조하여 실행하게 각 쓰레드는 자신의 ID를 알아야 한다. 이를 위하여 각 쓰레드 ID를 레지스터에 저장을 하고 fork 를 이용하여 ID값을 초기화 한다. 시리얼 영역을 실행할 때 fork 함수는 각 쓰레드를 위한 스택 영역을 할당하고, 함수의 매개 변수들을 스택에 복사하며 복사가 끝난 후에 쓰레드 #1을 실행한다. join 은 fork 안에 포함되어 있고, 병렬 실행 영역이 끝난 후 쓰레드 #1을 스톨(stall)을 시키고 쓰레드 #0를 계속적으로 실행시킨다.

<67> 그러나 병렬 코드 안에 if-else/switch 와 같은 조건 분기가 있을 경우에 두개의 쓰레드는 같은 코드 영역을 실행하지 않을 수 있다. 본 발명에서 사용한 CMP 구조에서는 한 번에 오직 하나의 명령어만 읽어 들이고 해석할 수 있기 때문에 두개의 쓰레드가 다른 코드 영역을 실행할 경우 동시에 처리할 수 없게 된다. 이런 문제점은 오직 조건 분기에서만 시작된다. 반대로 두개의 쓰레드가 서로 다른 컨트롤 플로우를 실행하다가 다시 같은 코드 영역을 실행하게 되는데 이런 부분을 알기 위해서 컴파일러에서 새로운 명령어를 정의하고 API를 추가한다. 그리고 위에서 설명한 바와 같이 쓰레드를 지원하기 위한 함수들을 라이브러리에 포함시킨다. 멀티 쓰레드를 위해서 새롭게 추가되는 명령어는 표 1과 같다.

표 1

<68>

새로운 명령어	기능
fork	병렬 실행을 시작
join	병렬 실행을 종료
acquire	변수에 락(lock)을 걸어서 다른 쓰레드가 사용하지 못함
release	변수의 락(lock)을 해제
barrier	동기화(두개의 쓰레드가 다른 컨트롤 플로우일 때 같은 PC에서 동기화)
thread_id	쓰레드의 id를제공

<69> 도 7은 본 발명의 또다른 실시 예에 따른 임베디드 코어에서 멀티 쓰레드 실행 방법의 흐름도이다.

<70> 먼저, 멀티 쓰레드로 제작된 코드를 명령어 메모리로부터 인출한다(710 과정).

<71> 다음, 명령어 정보를 저장하는 스패셜 레지스터를 이용하여 인출된 코드들을 디코드하고, 인출된 코드들에 2개 이상의 쓰레드에서 컨트롤 플로우를 분기시키는 명령어가 존재하는 경우 각각의 쓰레드의 순서를 설정한다(720 과정).

<72> 다음, 시리얼 코드 영역에서는 범용 레지스터의 전체 영역을 할당하고, 패러럴 코드 영역에서는 범용 레지스터를 2개 이상으로 분할한 영역을 각각의 쓰레드를 위해 할당한다(730 과정).

<73> 마지막으로, 복수의 실행부를 이용하여 설정된 실행 순서에 따라 각각의 쓰레드에 대한 코드를 실행한다(740 과정).

<74> 도 8은 본 발명의 또다른 실시 예에 따른 임베디드 코어에서 멀티 쓰레드 실행 방법의 흐름도이다.

<75> 먼저, 멀티 쓰레드로 제작된 코드를 명령어 메모리로부터 인출한다(810 과정).

<76> 다음, 명령어 정보를 저장하는 스패셜 레지스터를 이용하여 인출된 코드들을 디코드하고, 인출된 코드들에 2개 이상의 쓰레드에서 컨트롤 플로우를 분기시키는 명령어가 존재하는 경우 각각의 쓰레드의 순서를 설정한다(820 과정).

<77> 다음, 시리얼 코드 영역에서는 범용 레지스터의 전체 영역을 할당하고, 패러럴 코드 영역에서는 범용 레지스터를 2개 이상으로 분할한 영역을 각각의 쓰레드를 위해 할당한다(830 과정).

<78> 다음, 싱글 포트 캐쉬를 이용하는 복수의 실행부를 이용하여 설정된 실행 순서에 따라 각각의 쓰레드에 대한 코드를 실행한다(840 과정).

<79> 이때, 각각의 쓰레드에 대한 코드를 실행하고, 2개 이상의 쓰레드가 동시에 싱글 포트 캐쉬에 접근하는 경우, 이웃한 어드레스에 대한 접근인지 판단한다(850 과정).

<80> 이때, 이웃한 어드레스에 대한 접근이면, 한번의 접근으로 처리하고(851 과정), 이웃한 어드레스에 대한 접근이 아니면, 각각의 접근을 순차적으로 처리한다(852 과정).

<81> 마지막으로, 인출된 코드에 따라 프로그램이 종료되는지 판단하고(860 과정), 종료되지 않으면, 다음 코드들을 인출하여 위와 같은 과정을 반복한다(810-852 과정). 이때, 인출된 코드에 따라 프로그램이 종료되면, 모든 과정을 종료한다.

<82> 본 발명에 따른 새로운 병렬 구조의 성능 데이터를 얻고 이를 종래의 구조의 성능과 비교하기 위해서, EEMBC 벤치마크를 사용하였다. EEMBC는 총 34개의 벤치마크로 구성되어 있고 이 중에서 11개의 코드를 병렬화할 수 있었다. 또한, 구성의 일 예로, 하드웨어는 ADChips 사의 EISC AE32000C 를 확장하여 사용할 수 있다.

<83> 도 9는 본 발명에 따른 명령어 부하를 도시한 것이다.

<84> 도 9는 확장된 코어에서 멀티 쓰레드 프로그램을 실행하기 위해서 벤치마크 프로그램을 컴파일했을 때 실제 실행되는 총 명령어의 수가 얼마나 증가하는지를 보여준다. 실제 병렬 코드 영역에서는 16개의 레지스터를 두개의 그룹으로 나누어서 8개씩 사용하기 때문에 실제 실행되는 명령어의 수가 증가하는 것을 볼 수 있다. 그러나, 이와 같은 레지스터의 부족에 따른 문제(register spill)는 16개의 레지스터를 모두 사용하여 해결할 수 있는 문제이다. 병렬화 부하(parallelization overhead)는 병렬 코드를 만들 때 fork를 사용하여 각 쓰레드를 위해 스택을 만들고 매개 변수를 복사하게 되는데 이런 일련의 일들로 인해서 생기는 것으로, 도 9에서 보는 것과 같이 conven을 제외한 다른 벤치마크에서의 병렬화 부하는 매우 적다.

<85> 표 2는 새로운 구조에서 얼마나 성능이 향상됐는가를 보여준다. 그림 5 에서 보았듯이 실제 실행되는 명령어의 수가 평균적으로 17.5%가 증가함에도 불구하고 싱글 포트 캐쉬에서 20.1%의 성능향상을 얻을 수 있었고 듀얼 포트 캐쉬를 사용할 경우 평균 36.5%, 최대 82.3%의 성능향상을 얻을 수 있었다.

<86> 표 2 확장된 구조의 speedup싱글 포트의 경우 캐쉬에 접근할 때 두개의 쓰레드가 동시에 데이터를 읽거나 쓸 수 없기 때문에 성능이 많이 떨어지게 된다. 하지만 두 쓰레드가 인접한 메모리(메모리의 차이가 4)인 어드레스에 접근할 경우에 캐쉬에서 2개의 쓰레드를 위한 데이터를 한 번에 처리하게 되고(Doubling) 이를 통해 성능을 향상시킬 수 있게 된다.

표 2

Benchmark	Unoptimized				Optimized			
	Single port	Doubling		Dual port	Single port	Doubling		Dual port
		outer	inner			outer	inner	
aifftr	1.04	1.10	1.10	1.18	1.13	1.14	1.14	1.18
aiifft	1.04	1.08	1.08	1.19	1.12	1.13	1.13	1.19
basefp	1.16	1.16	1.16	1.31	1.16	1.16	1.16	1.31
idctrn	1.34	1.39	1.48	1.51	1.33	1.37	1.48	1.51
autcor	1.15	1.24	1.24	1.43	1.15	1.24	1.24	1.43
conven	0.78	0.83	0.83	0.87	1.00	1.00	1.00	1.00
fft	1.04	1.13	1.16	1.26	1.04	1.13	1.16	1.26
viterb	0.91	0.92	0.92	1.07	1.00	1.00	1.00	1.07
bezierfloat	1.34	1.34	1.34	1.69	1.34	1.34	1.57	1.82
bezierfixed	1.57	1.57	1.57	1.82	1.57	1.57	1.57	1.82
rgbhpg	1.57	1.36	1.55	1.61	1.36	1.36	1.55	1.61
average	1.16	1.19	1.22	1.36	1.20	1.22	1.25	1.37

<87>

<88> 멀티 쓰레드 프로세서를 구현하기 위해서 종래의 EISC 코어에 새로운 로직을 추가하였다.

<89> 본 발명에 따른 전력을 측정하기 위해서 Synopsys 사의 Design Compiler와 PrimePower를 사용하였다. Design Compiler를 이용하여 AE32000C의 Verilog 코드를 합성하고 PrimePower를 이용하여 전력을 측정했다. 전력 소모를 측정하기 위한 공정 설정은 공급 전압 3.3V에 0.35um CMOS process이고 클럭 주파수는 66MHz로 가정하였다.

<90> 표 3은 새로운 로직을 추가함으로써 생기는 부하를 나타낸 것이다. 실제 실행부(Execution unit)를 추가했기 때문에 실행부의 크기는 92.5%만큼 증가하고 전력 소모는 81.29%만큼 증가하였다. 디코드부(Decode unit)의 경우에 명령어를 해석하고 각 명령어에 맞는 오퍼랜드(operand, 명령어를 실행하기 위한 인자들)가 두개의 쓰레드에서 실행되기 때문에 35.10%만큼 크기가 증가하고 전력 소모는 31.05%만큼 증가하였다. 메모리(Memory) 유닛의 경우에도 두개의 쓰레드를 처리해야 하기 때문에 15.09 %만큼 증가하였다. 전체적으로 하드웨어의 크기는 46.98% 증가하였고 전력 소비량은 24.23%만큼 증가하였다. 이는 실제 ARM의 MPCore(Multi Core Processor)와 비교할 때 매우 작은 값이다. ARM MPCore의 경우 2 개의 프로세서를 사용할 때 125%만큼 크기가 증가하고 전력 소비도 117%증가한다.

표 3

컴포넌트		하드웨어 복잡도(Gates)			소비 전력(mW)		
		기본 Core	확장 Core	증가량	기본 Core	확장 Core	증가량
Core	Fetch	8226	8104	-1.48%	27.35	27.26	-0.33
	Decode and operand fetch	14840	20049	35.10%	35.01	48.88	31.05%
	Execution	20308	39001	92.05%	23.34	42.53	81.29%
	Memory and wirte back	10364	11928	15.09%	48.71	51.45	5.36%
	Others	228	237	3.95%	0	0	0%
	Total	53966	79319	46.98%	134.53	167.12	24.23%
Doubling cache interface		7665	13289	73.37%	36374	65.23	77.54%

<91>

<92> 본 발명은 도면에 도시된 일 실시예를 참고로 하여 설명하였으나 이는 예시적인 것에 불과하며 당해 분야에서

통상의 지식을 가진 자라면 이로부터 다양한 변형 및 실시예의 변형이 가능하다는 점을 이해할 것이다. 그러나, 이와 같은 변형은 본 발명의 기술적 보호범위내에 있다고 보아야 한다. 따라서, 본 발명의 진정한 기술적 보호범위는 첨부된 특허청구범위의 기술적 사상에 의해서 정해져야 할 것이다.

발명의 효과

<93> 상술한 바와 같이, 본 발명에 의하면, CMP의 성능을 유지하는 상태에서 전력 소모량을 줄이고, 프로세서의 면적을 줄여 제조가 용이한 효과가 있다.

도면의 간단한 설명

<1> 도 1은 본 발명의 위치를 기술 트리로 나타낸 것이다.

<2> 도 2a는 본 발명이 적용되는 임베디드 어플리케이션에서 같은 코드가 실행되는 경우와 컨트롤 플로우가 달라지는 경우의 스피드업을 도시한 것이다.

<3> 도 2b는 본 발명이 적용되는 임베디드 어플리케이션에서 시리얼과 패러럴 영역에서 실제 실행되는 명령어의 분포를 도시한 것이다.

<4> 도 3은 본 발명의 일 실시 예에 따른 임베디드 코어에서 멀티 쓰레드 실행을 위해 확장된 프로세서의 블럭도이다.

<5> 도 4는 본 발명의 다른 실시 예에 따른 임베디드 코어에서 멀티 쓰레드 실행을 위해 확장된 프로세서의 블럭도이다.

<6> 도 5는 본 발명의 또다른 실시 예에 따른 임베디드 코어에서 멀티 쓰레드 실행을 위해 확장된 프로세서의 블럭도이다.

<7> 도 6은 도 5에 적용되는 더블링 데이터 캐쉬 인터페이스의 블럭도이다.

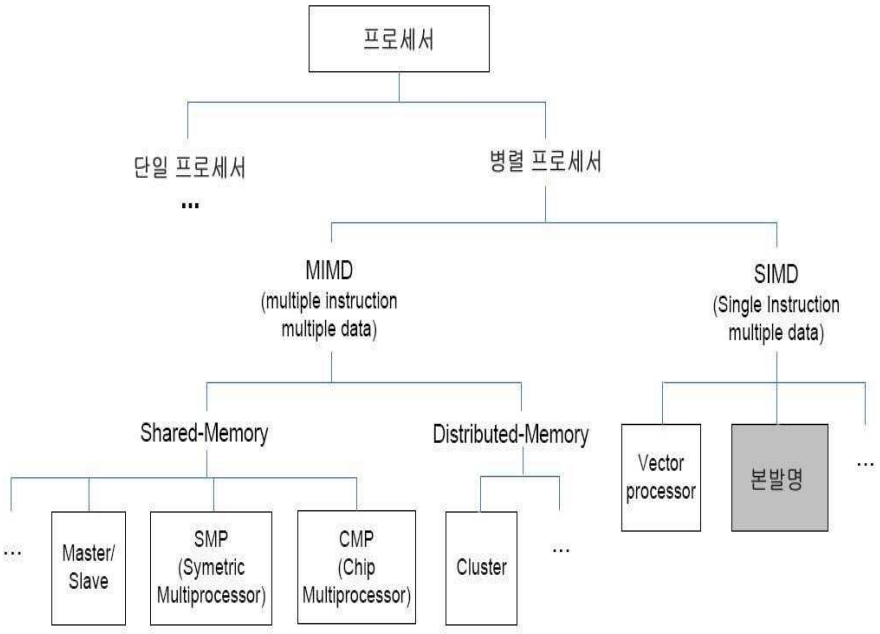
<8> 도 7은 본 발명의 또다른 실시 예에 따른 임베디드 코어에서 멀티 쓰레드 실행 방법의 흐름도이다.

<9> 도 8은 본 발명의 또다른 실시 예에 따른 임베디드 코어에서 멀티 쓰레드 실행 방법의 흐름도이다.

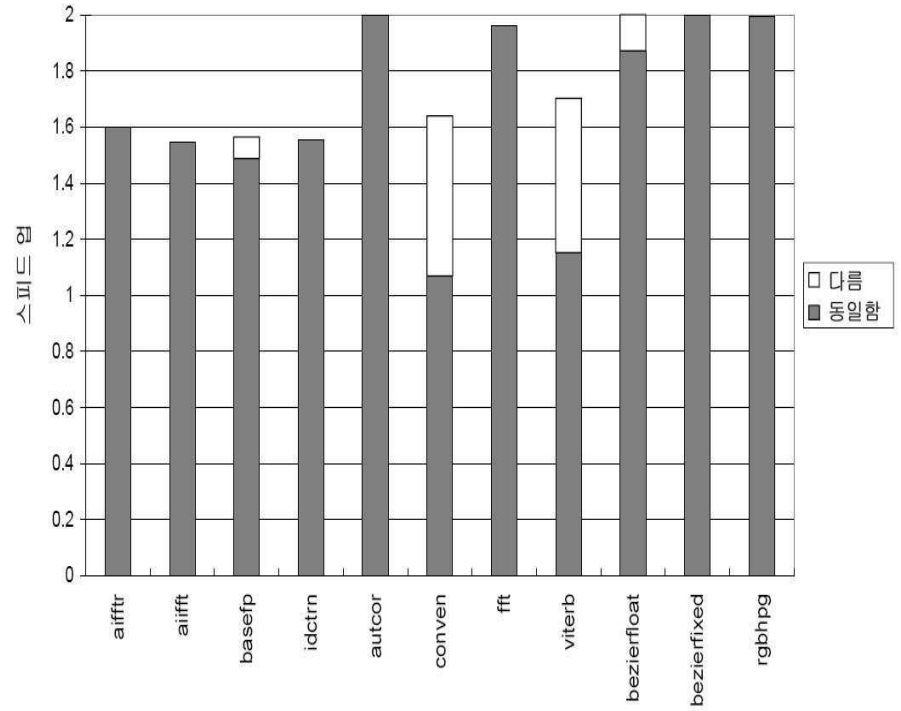
<10> 도 9는 본 발명에 따른 명령어 부하를 도시한 것이다.

도면

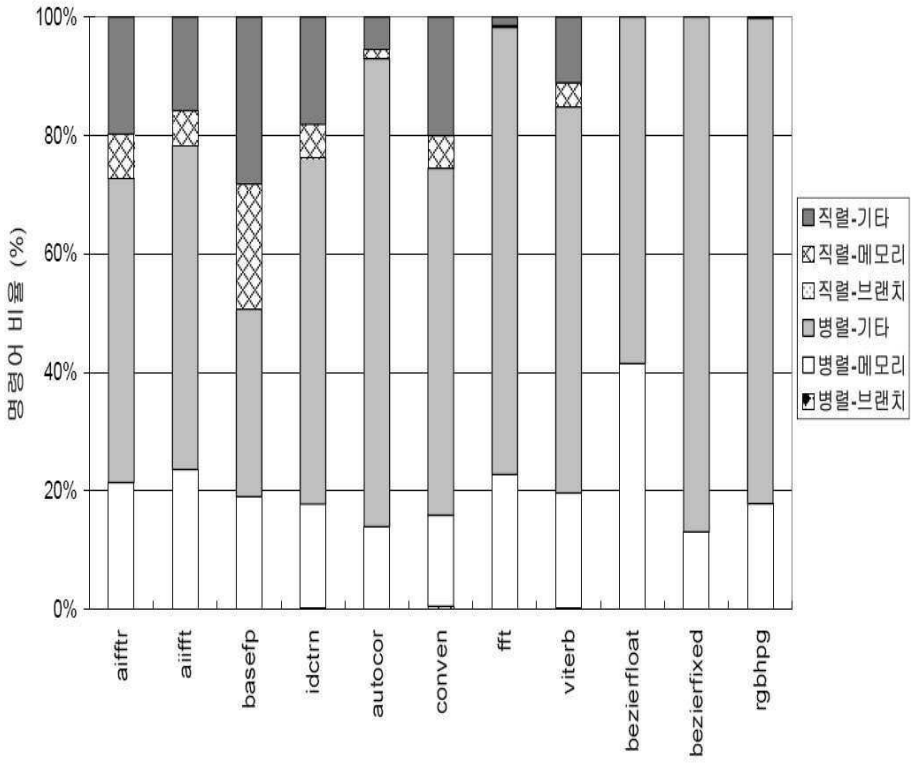
도면1



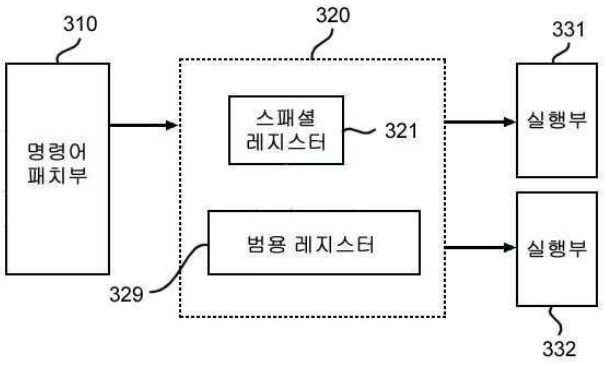
도면2a


스피드 업
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
aifftr
aiifft
basefp
idctrn
autcor
conven
fft
viterb
bezierfloat
bezierfixed
rgbhpg
다름
동일함

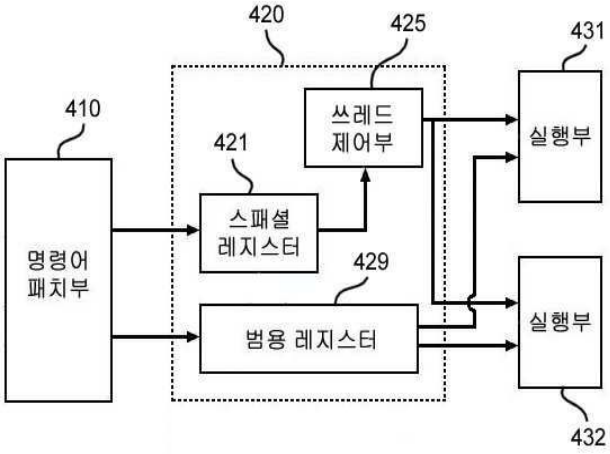
도면2b


명령어 비율 (%)
100%
80%
60%
40%
20%
0%
aifftr
aiifft
basefp
idctrn
autocor
conven
fft
viterb
bezierfloat
bezierfixed
rgbhpg
직렬-기타
직렬-메모리
직렬-브랜치
병렬-기타
병렬-메모리
병렬-브랜치

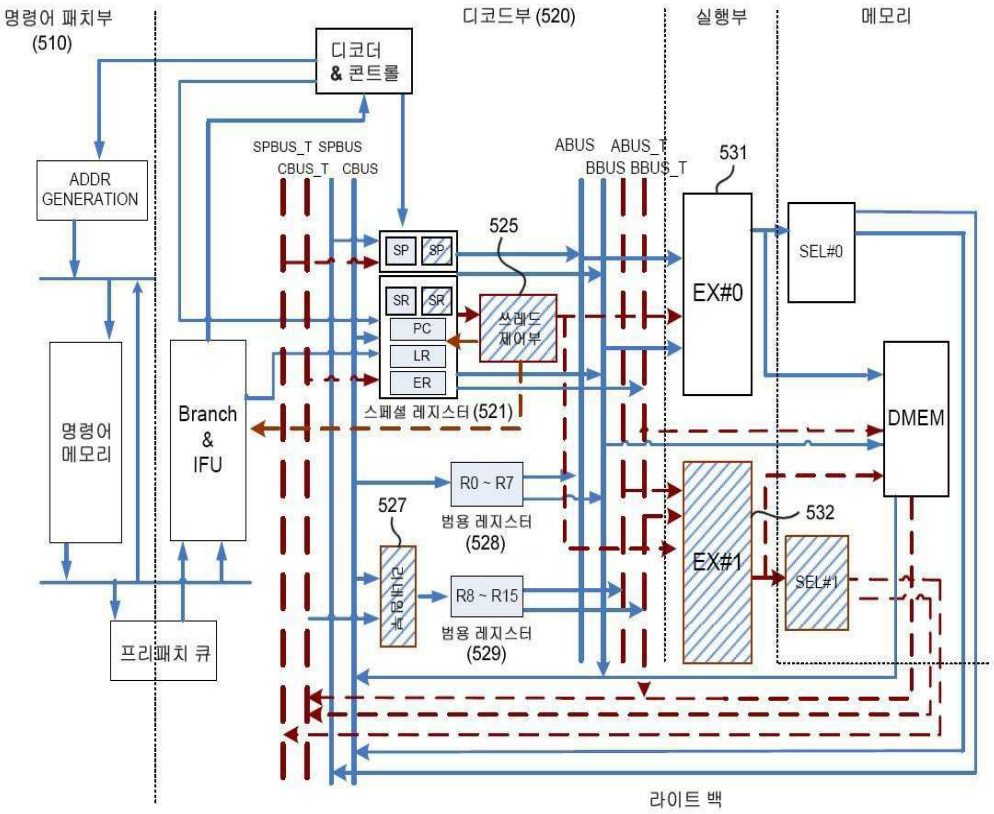
도면3


310
320
331
명령어
패치부
스패셜
레지스터
321
실행부
범용 레지스터
329
실행부
332

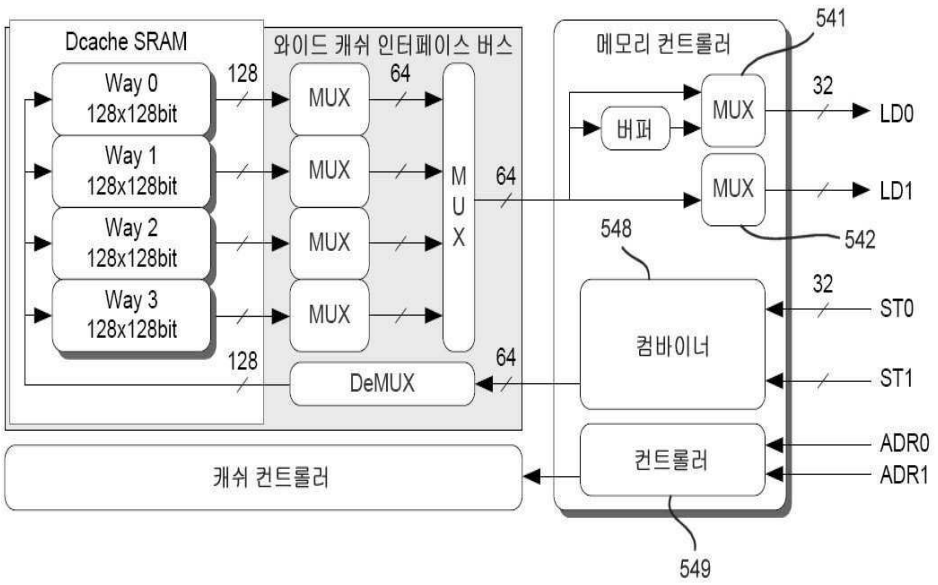
도면4


420
425
431
410
쓰레드
제어부
421
실행부
명령어
패치부
스패셜
레지스터
429
범용 레지스터
실행부
432

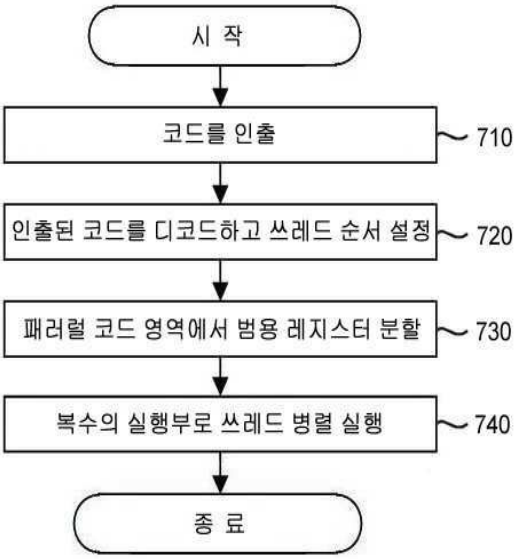
도면5


명령어 패치부
(510)
디코더
& 콘트롤
디코드부 (520)
실행부
메모리
ADDR
GENERATION
SPBUS_T SPBUS
CBUS_T CBUS
ABUS ABUS_T
BBUS BBUS_T
531
525
SP
SR
PC
LR
ER
쓰레드
제어부
스페셜 레지스터 (521)
EX#0
SEL#0
명령어
메모리
Branch
&
IFU
DMEM
R0 ~ R7
범용 레지스터
(528)
527
532
EX#1
SEL#1
R8 ~ R15
범용 레지스터
(529)
프리패치 큐
라이트 백

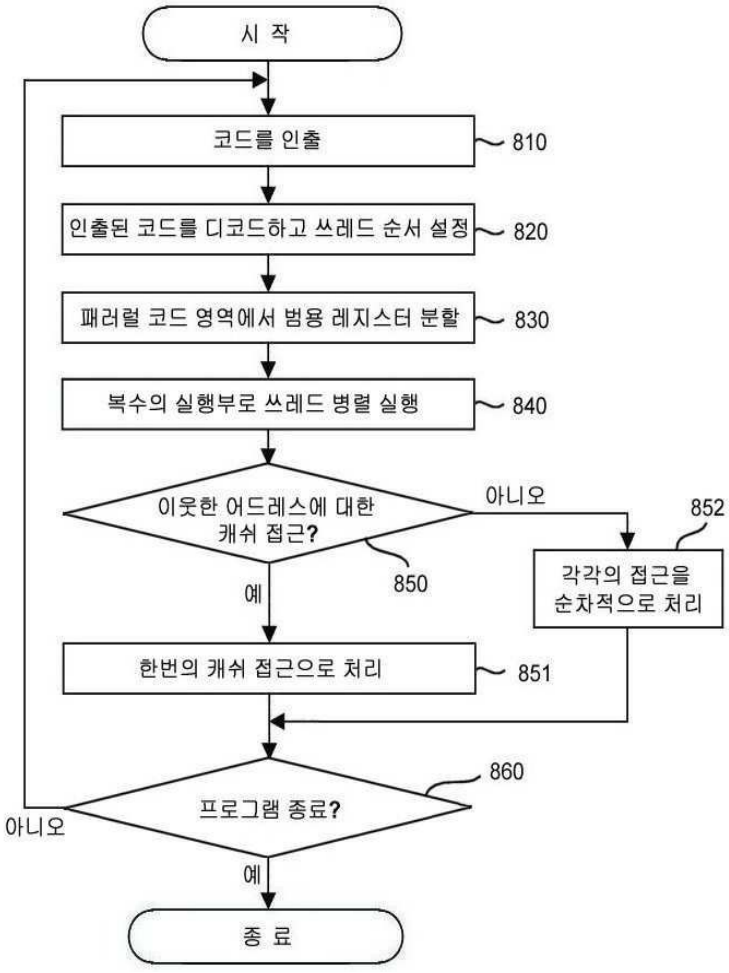
도면6


Dcache SRAM
와이드 캐쉬 인터페이스 버스
메모리 컨트롤러
541
Way 0
128x128bit
Way 1
128x128bit
Way 2
128x128bit
Way 3
128x128bit
128
MUX
64
M
U
X
버퍼
32
LD0
LD1
542
548
컴바이너
ST0
ST1
DeMUX
ADR0
ADR1
컨트롤러
캐쉬 컨트롤러
549

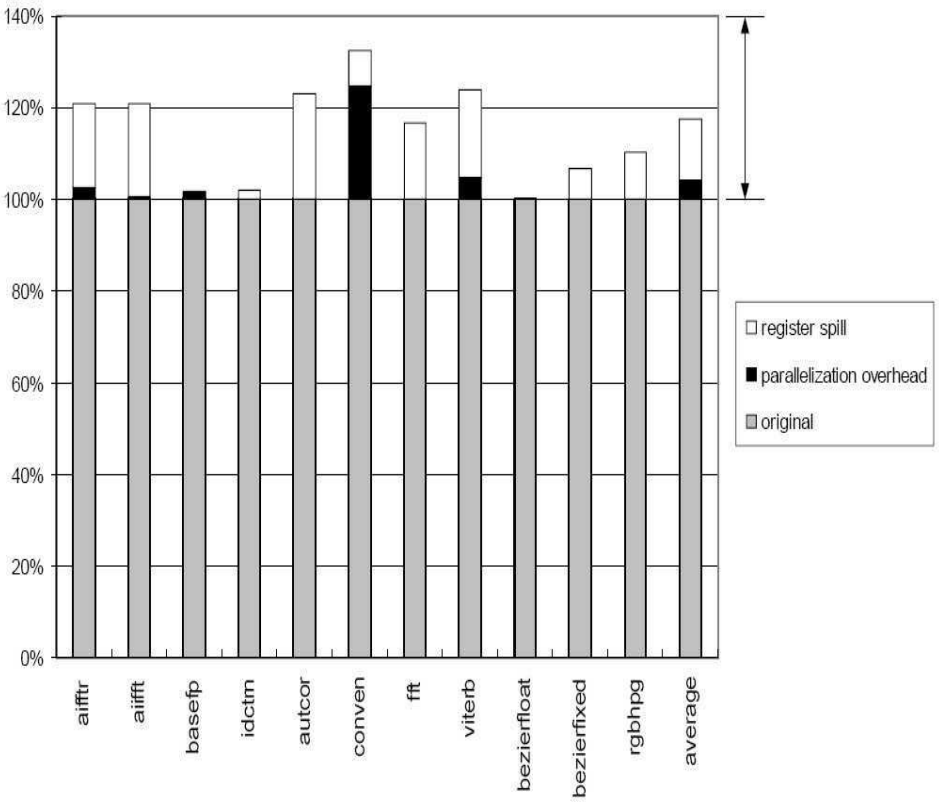
도면7


시 작
코드를 인출
710
인출된 코드를 디코드하고 쓰레드 순서 설정
720
패러럴 코드 영역에서 범용 레지스터 분할
730
복수의 실행부로 쓰레드 병렬 실행
740
종 료

도면8


시 작
코드를 인출
810
인출된 코드를 디코드하고 쓰레드 순서 설정
820
패러럴 코드 영역에서 범용 레지스터 분할
830
복수의 실행부로 쓰레드 병렬 실행
840
이웃한 어드레스에 대한 캐쉬 접근?
850
아니오
852
각각의 접근을 순차적으로 처리
예
한번의 캐쉬 접근으로 처리
851
프로그램 종료?
860
아니오
예
종 료

도면9


140%
120%
100%
80%
60%
40%
20%
0%
aifftr
aiifft
basefp
idctm
autcor
conven
fft
viterb
bezierfloat
bezierfixed
rgbhpg
average
register spill
parallelization overhead
original