(19) 대한민국특허청(KR)
(12) 등록특허공보(B1)

(45) 공고일자 2018년03월28일
(11) 등록번호 10-1842550
(24) 등록일자 2018년03월21일

(51) 국제특허분류(Int. Cl.)
G06F 9/38 (2006.01) *G06F 9/30* (2018.01)
(52) CPC특허분류
G06F 9/3808 (2013.01)
G06F 9/3017 (2013.01)
(21) 출원번호 10-2017-7002473(분할)
(22) 출원일자(국제) 2011년11월22일
심사청구일자 2017년02월24일
(85) 번역문제출일자 2017년01월26일
(65) 공개번호 10-2017-0016514
(43) 공개일자 2017년02월13일
(62) 원출원 특허 10-2014-7016764
원출원일자(국제) 2011년11월22일
심사청구일자 2014년11월25일
(86) 국제출원번호 PCT/US2011/061953
(87) 국제공개번호 WO 2013/077875
국제공개일자 2013년05월30일
(56) 선행기술조사문헌
US5517651 A*
US6065105 A*
US20100161948 A1*
US05517651 A*
*는 심사관에 의하여 인용된 문헌

(73) 특허권자
소프트 머신즈, 인크.
미국 95054 캘리포니아주 산타 클라라 프리덤 서클 3920
(72) 발명자
압달라, 모하마드
미국 95132 캘리포니아주 산 호세 선크레스트 애비뉴 3868
(74) 대리인
양영준, 백만기

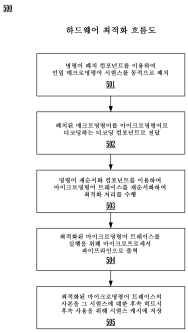
전체 청구항 수 : 총 8 항 심사관 : 김경완

(54) 발명의 명칭 **다중 엔진 마이크로프로세서용 가속 코드 최적화기**

(57) 요 약

마이크로프로세서에서 코드 최적화를 가속화하는 방법이 개시된다. 상기 방법은 명령어 페치 컴포넌트를 이용하여 인입 매크로명령어 시퀀스를 페치하는 단계 및 페치된 매크로명령어를 마이크로명령어로 디코딩하는 디코딩 컴포넌트로 전달하는 단계를 포함한다. 상기 마이크로명령어 시퀀스를 복수의 의존적 코드 그룹을 포함하는 최적화된 마이크로명령어 시퀀스로 재순서화함으로써 최적화 처리가 수행된다. 그 다음에, 복수의 의존적 코드 그룹은 실행을 위해 상기 마이크로프로세서의 복수의 엔진으로 병렬로 출력된다. 최적화된 마이크로명령어 시퀀스에 대한 후속 히트시 후속 사용을 위해 최적화된 마이크로명령어 시퀀스의 사본이 시퀀스 캐시에 저장된다.

대 표 도 - 도5



(52) CPC특허분류
G06F 9/3838 (2013.01)
G06F 9/3853 (2013.01)
G06F 9/3887 (2013.01)

명 세 서

청구범위

청구항 1

마이크로프로세서로서,

인입 매크로명령어 시퀀스를 페치하는 명령어 페치 컴포넌트와,

상기 매크로명령어 시퀀스를 수신하고 마이크로명령어 시퀀스로 디코딩하는 디코딩 컴포넌트와,

상기 마이크로명령어 시퀀스를 액세스하고, SIMD(Single Instruction Multiple Data) 명령어들을 이용하여 의존성 매트릭스를 상기 마이크로명령어 시퀀스에서 추출된 의존성 정보로 채우고(populate), 상기 의존성 매트릭스의 복수의 행들(rows)을 스캔하여 상기 마이크로명령어 시퀀스를 최적화된 마이크로명령어 시퀀스 - 마이크로명령어들은 복수의 의존적 코드 그룹으로 그룹화됨 - 로 재순서화하는 최적화 처리를 수행하는 최적화기 컴포넌트 - SIMD 명령어들을 이용하여 상기 의존성 매트릭스를 채우는 것은 상기 의존성 정보를 결정하기 위해 페어링(pairing) 매치 또는 블로킹 매치를 수행하는 것을 포함함 -와,

상기 복수의 의존적 코드 그룹을 수신하고, 상기 복수의 의존적 코드 그룹을 병렬로 실행하는 복수의 엔진과,

상기 최적화된 마이크로명령어 시퀀스에 대한 후속 히트시 후속 사용을 위해 상기 최적화된 마이크로명령어 시퀀스의 사본을 수신 및 저장하는 시퀀스 캐시를 포함하는

마이크로프로세서.

청구항 2

제1항에 있어서,

상기 최적화 처리는 상기 의존성 매트릭스의 복수의 행들을 스캐닝하여 매칭 명령어들(matching instructions)을 식별하는 것을 포함하는

마이크로프로세서.

청구항 3

제2항에 있어서,

상기 최적화 처리는 상기 매칭 명령어들을 분석하여 상기 매칭 명령어들이 블로킹 의존성(blocking dependency)을 포함하는지를 결정하는 것을 더 포함하고,

상기 블로킹 의존성을 제거하기 위해 재명명(renaming)이 수행되는

마이크로프로세서.

청구항 4

제3항에 있어서,

상기 최적화 처리는 상기 의존성 매트릭스의 각 행의 제1 매치들에 대응하는 명령어들을 대응하는 의존성 그룹으로 이동시키는 것을 더 포함하는

마이크로프로세서.

청구항 5

제1항에 있어서,

최적화된 마이크로명령어 시퀀스들의 사본들을 저장하는 메모리 계층을 더 포함하는

마이크로프로세서.

청구항 6

제5항에 있어서,

상기 메모리 계층은 L1 캐시 및 L2 캐시를 포함하는

마이크로프로세서.

청구항 7

제1항에 있어서,

상기 마이크로명령어 시퀀스의 사본을 수신하고 저장하는 마이크로명령어 캐시를 더 포함하는

마이크로프로세서.

청구항 8

제1항에 있어서,

상기 블로킹 매치는 레지스터 재명명(renaming)을 통해 해결되는

마이크로프로세서.

발명의 설명

기 술 분 야

[0001] 관련 출원에 대한 상호 참조

[0002] 본 출원은 2010년 1월 5일 모하메드 에이. 압달라(Mohammad A. Abdallah)에 의해 "APPARATUS AND METHOD FOR PROCESSING COMPLEX INSTRUCTION FORMATS IN A MULTITHREADED ARCHITECTURE SUPPORTING VARIOUS CONTEXT SWITCH MODES AND VIRTUALIZATION SCHEMES"라는 명칭으로 출원되어 동시계류중인 본 출원과 공동으로 양도된 미국 특허 출원 제2010/0161948호와 관련되며, 이 출원은 그 전체가 본 명세서에서 인용된다.

[0003] 본 출원은 2008년 12월 19일 모하메드 에이. 압달라에 의해 "APPARATUS AND METHOD FOR PROCESSING AN INSTRUCTION MATRIX SPECIFYING PARALLEL IN DEPENDENT OPERATIONS"라는 명칭으로 출원되어 동시계류중인 본 출원과 공동으로 양도된 미국 특허 출원 제2009/0113170호와 관련되며, 이 출원은 그 전체가 본 명세서에서 인용된다.

[0004] 본 출원은 2010년 9월 17일 모하메드 에이. 압달라에 의해 "SINGLE CYCLE MULTI-BRANCH PREDICTION INCLUDING SHADOW CACHE FOR EARLY FAR BRANCH PREDICTION"라는 명칭으로 출원되어 동시계류중인 본 출원과 공동으로 양도된 미국 특허 출원 제61/384,198호와 관련되며, 이 출원은 그 전체가 본 명세서에서 인용된다.

[0005] 본 출원은 2011년 3월 25일 모하메드 에이. 압달라에 의해 "EXECUTING INSTRUCTION SEQUENCE CODE BLOCKS BY USING VIRTUAL CORES INSTANTIATED BY PARTITIONABLE ENGINES"라는 명칭으로 출원되어 동시계류중인 본 출원과 공동으로 양도된 미국 특허 출원 제61/467,944호와 관련되며, 이 출원은 그 전체가 본 명세서에서 인용된다.

[0006] 발명의 분야

[0007] 본 발명은 일반적으로 디지털 컴퓨터 시스템에 관한 것으로, 특히, 명령어 시퀀스(instruction sequence)를 포함하는 명령어를 선택하는 시스템 및 방법에 관한 것이다.

배 경 기 술

[0008] 프로세서는 의존적이거나 완전히 독립적인 다중 태스크를 처리하는 것이 필요하다. 이러한 프로세서의 내부 상태는 일반적으로 프로그램 실행의 각 특정 순간에 상이한 값들을 유지(hold)할 수 있는 레지스터들로 이루어진다. 프로그램 실행의 각 순간에, 내부 상태 이미지는 프로세서의 아키텍처 상태로 불린다.

[0009] 다른 함수(예를 들어, 다른 스레드(thread), 프로세스 또는 프로그램)을 실행하기 위해 코드 실행이 전환된 경우, 머신(machine)/프로세서의 상태는 새로운 함수가 내부 레지스터를 이용하여 그의 새로운 상태를 구축할 수 있도록 저장되어야 한다. 일단 새로운 함수가 끝나면, 그의 상태는 폐기될 수 있고 이전 콘텍스트(context)의

상태가 복원될 것이고 실행이 재개된다. 이러한 전환 처리는 콘텍스트 전환(context switch)으로 불리고 일반적으로 특히 많은 수의 레지스터(예를 들어, 64, 128, 256) 및/또는 비순차적 실행을 이용하는 현대 아키텍처의 경우 수십 또는 수백 사이클을 포함한다.

[0010] 스레드 인지 하드웨어 아키텍처(thread-aware hardware architectures)에서, 하드웨어가 제한된 수의 하드웨어 지원 스레드에 대해 다중 콘텍스트의 상태를 지원하는 것이 일반적이다. 이 경우, 하드웨어는 각 지원 스레드마다 모든 아키텍처 상태 요소(architecture state element)를 이중화한다(duplicate). 이는 새로운 스레드를 실행할 때 콘텍스트 전환에 대한 필요성을 제거한다. 그러나, 이는 여전히 많은 단점, 즉 하드웨어에서 지원되는 각 추가 스레드마다 모든 아키텍처 상태 요소(즉, 레지스터)를 이중화하기 위한 면적, 전력 및 복잡도를 갖는다. 추가로, 만일 소프트웨어 스레드의 개수가 명백히 지원되는 하드웨어 스레드의 개수를 초과한 경우, 여전히 콘텍스트 전환이 수행되어야 한다.

[0011] 이는 많은 수의 스레드를 요구하는 미세 단위로(on a fine granularity basis) 병렬성(parallelism)이 필요함에 따라 흔해진다. 이중화 콘텍스트-상태 하드웨어 스토리지(duplicate context-state hardware storage)를 갖는 하드웨어 스레드 인지 아키텍처는 스레드되지 않은 소프트웨어 코드(non-threaded software code)에 도움이 되지 못하고 단지 스레드된 소프트웨어에 대해 콘텍스트 전환의 수만 줄여준다. 그러나, 이러한 스레드는 일반적으로 조립자(coarse grain) 병렬성을 위해 구축되어, 개시 및 동기화를 위해 소프트 오버헤드(overhead)를 가중시키는 결과를 가져와, 효율적인 스레딩 개시/자동 생성 없이, 함수 호출 및 루프 병렬 실행과 같은 미립자(fine grain) 병렬성을 유지(leave)한다. 이러한 설명된 오버헤드는 비-명백하게/쉽게 병렬화된/스레드된 소프트웨어 코드(non-explicitly/easily parallelized/threaded software codes)에 대해 종래 기술의 컴파일러 또는 사용자 병렬화 기술을 이용한 그러한 코드의 자동 병렬화의 어려움을 수반한다.

발명의 내용

[0012] 일 실시예에서, 본 발명은 마이크로프로세서에서 코드 최적화를 가속화하는 방법으로서 구현된다. 상기 방법은 명령어 페치(fetch) 컴포넌트를 이용하여 인입(incoming) 매크로명령어(macro instruction) 시퀀스를 페치하는 단계 및 페치된 매크로명령어를 마이크로명령어로 디코딩하는 디코딩 컴포넌트로 전달하는 단계를 포함한다. 상기 마이크로명령어 시퀀스를 복수의 의존적 코드 그룹을 포함하는 최적화된 마이크로명령어 시퀀스로 재순서화(reordering)함으로써 최적화 처리가 수행된다. 다음에, 상기 복수의 의존적 코드 그룹은 실행을 위해 상기 마이크로프로세서의 복수의 엔진으로 병렬로 출력된다. 최적화된 마이크로명령어 시퀀스에 대한 후속 히트(hit)시 후속 사용을 위해 최적화된 마이크로명령어 시퀀스의 사본이 시퀀스 캐시에 저장된다.

[0013] 전술한 바는 개요이므로 필요에 의해 세부 사항의 간략화, 일반화 및 생략을 포함하고; 따라서, 이 기술분야의 통상의 기술자는 이 개요가 단지 예시적이며 어떤 방식으로든 제한적인 것으로 의도되지 않음을 인식할 것이다. 오직 특허청구범위에 의해서만 규정된 바와 같은 본 발명의 다른 양태, 발명적 특징, 및 이점은 아래에 기술된 비제한적인 상세한 설명에서 명확해질 것이다.

도면의 간단한 설명

[0014] 본 발명은 첨부 도면에서 제한이 아닌 예시적으로 설명되고, 첨부 도면에서 같은 참조 부호는 유사한 구성 요소를 나타낸다.

도 1은 본 발명의 일 실시예에 따른 마이크로프로세서의 할당/발행 스테이지의 개요도를 도시한다.

도 2는 본 발명의 일 실시예에 따른 최적화 처리를 예시하는 개요도를 도시한다.

도 3은 본 발명의 일 실시예에 따른 다단계의 최적화 처리를 도시한다.

도 4는 본 발명의 일 실시예에 따른 다단계의 최적화 및 명령어 이동 처리를 도시한다.

도 5는 본 발명의 일 실시예에 따른 예시적인 하드웨어 최적화 처리의 단계 흐름도를 도시한다.

도 6은 본 발명의 일 실시예에 따른 대안의 예시적인 하드웨어 최적화 처리의 단계 흐름도를 도시한다.

도 7은 본 발명의 일 실시예에 따른 할당/발행 스테이지의 CAM 매칭 하드웨어 및 우선순위 인코딩 하드웨어의 동작을 도시하는 도면을 도시한다.

도 8은 본 발명의 일 실시예에 따른 분기(branch) 앞에 최적화된 스케줄링을 예시하는 도면을 도시한다.

도 9는 본 발명의 일 실시예에 따른 스토어(store) 앞에 최적화된 스케줄링을 예시하는 도면을 도시한다.

도 10은 본 발명의 일 실시예에 따른 예시적인 소프트웨어 최적화 처리의 도면을 도시한다.

도 11은 본 발명의 일 실시예에 따른 SIMD 소프트웨어 기반 최적화 처리의 흐름도를 도시한다.

도 12는 본 발명의 일 실시예에 따른 예시적인 SIMD 소프트웨어 기반 최적화 처리의 동작 단계 흐름도를 도시한다.

도 13은 본 발명의 일 실시예에 따른 소프트웨어 기반 의존성 브로드캐스팅 처리(software based dependency broadcast process)를 도시한다.

도 14는 본 발명의 일 실시예에 따라 의존성 명령어 그룹화(dependency grouping of instructions)가 가변 제한된 의존성 명령어 그룹(variably bounded groups of dependent instructions)을 구축하는데 어떻게 사용될 수 있는지를 보여주는 예시적인 흐름도를 도시한다.

도 15는 본 발명의 일 실시예에 따른 계층적 명령어 스케줄링을 묘사하는 흐름도를 도시한다.

도 16은 본 발명의 일 실시예에 따른 3 슬롯 의존성 그룹 명령어의 계층적 스케줄링을 묘사하는 흐름도를 도시한다.

도 17은 본 발명의 일 실시예에 따른 3 슬롯 의존성 그룹 명령어의 계층적 이동 윈도우 스케줄링(hierarchical moving window scheduling)을 묘사하는 흐름도를 도시한다.

도 18은 본 발명의 일 실시예에 따라 가변 크기의 의존성 명령어 체인(예를 들어, 가변 제한된 명령어 그룹)이 복수의 컴퓨팅 엔진에 어떻게 할당되는지를 보여준다.

도 19는 본 발명의 일 실시예에 따라 스케줄링 큐에의 블록 할당 및 3 슬롯 의존성 그룹 명령어의 계층적 이동 윈도우 스케줄링을 묘사하는 흐름도를 도시한다.

도 20은 본 발명의 일 실시예에 따라 의존성 코드 블록(예를 들어, 의존성 그룹 또는 의존성 체인)이 엔진에서 어떻게 실행되는지를 도시한다.

도 21은 본 발명의 일 실시예에 따라 다중코어 프로세서를 위한 글로벌 프론트 엔드 페치 & 스케줄러 및 레지스터 파일, 글로벌 상호접속부 및 프래그먼트된 메모리 서브시스템(a fragmented memory subsystem)을 포함하여 복수의 엔진 및 이들의 컴포넌트의 개요도를 도시한다.

도 22는 본 발명의 일 실시예에 따른 복수의 세그먼트, 복수의 세그먼트된 공통 분할 스케줄러 및 상호접속부 및 세그먼트의 포트를 도시한다.

도 23은 본 발명의 일 실시예에 따른 예시적인 마이크로프로세서 파이프라인의 도면을 도시한다.

발명을 실시하기 위한 구체적인 내용

[0015] 비록 본 발명이 일 실시예와 관련하여 설명되었지만, 본 발명은 본 명세서에 기술된 특정 형태로 제한되는 것으로 의도되지 않는다. 반대로, 첨부의 특허청구범위에 의해 규정된 바와 같은 본 발명의 범주 내에 합당하게 포함될 수 있는 바와 같은 대안, 변형, 및 균등물을 망라하는 것으로 의도된다.

[0016] 후술하는 상세한 설명에서, 특정 방법 순서, 구조, 구성 요소, 및 커넥션과 같은 많은 구체적인 세부 사항이 기술되었다. 그러나, 이러한 구체적인 세부 사항 및 다른 구체적인 세부 사항이 본 발명의 실시예를 실시하는데 사용될 필요는 없음이 이해될 것이다. 다른 경우에, 이와 같은 설명을 불필요하게 불명확하게 하지 않도록 하기 위해 공지의 구조, 구성 요소, 또는 커넥션은 생략되었거나, 특별히 구체적으로 설명되지 않았다.

[0017] 본 명세서 내에서 "일 실시예" 또는 "하나의 실시예"라는 언급은 그 실시예와 관련하여 기술된 특정한 특징, 구조, 또는 특성이 본 발명의 적어도 하나의 실시예에 포함된다는 것을 나타내는 것으로 의도된다. 본 명세서 내 여러 곳에서 "일 실시예에서"라는 문구의 출현은 반드시 모두가 동일한 실시예를 언급하는 것은 아니며, 별개의 또는 대안의 실시예도 다른 실시예와 상호 배타적인 것은 아니다. 또한, 일부 실시예에 의해 나타날 수 있지만 다른 실시예에 의해 나타나지 않는 다양한 특징이 기술된다. 유사하게, 일부 실시예에 대한 요건일 수 있지만 다른 실시예에 대해서는 아닌 다양한 요건이 기술된다.

[0018] 다음의 상세한 설명의 일부 부분은 절차, 단계, 논리 블록, 처리, 및 컴퓨터 메모리 내의 데이터 비트에 대한

다른 동작 기호 표현으로 제시된다. 이러한 설명 및 표현은 데이터 처리 기술의 통상의 기술자에 의해 이들의 연구의 본질을 본 기술 분야의 다른 숙련자들에게 가장 효율적으로 전달하기 위해 이용되는 수단이다. 여기에는, 절차, 컴퓨터 실행 단계, 논리 블록, 프로세스 등이 있고, 이들은 일반적으로 원하는 결과를 도출하는 단계 또는 명령어들의 자기 일치(self-consistent) 시퀀스로 인식된다. 이러한 단계는 물리량에 대한 물리적 조작을 필요로 하는 것이다. 일반적으로, 반드시는 아니지만, 이러한 양은 컴퓨터 판독가능한 저장 매체의 전기 또는 자기 신호의 형태를 취하고 컴퓨터 시스템에 저장되고, 전달되고, 결합되고, 비교되고, 다른 방식으로 조작될 수 있다. 주로 일반적인 사용의 이유로, 때때로 이러한 신호를 비트, 값, 요소, 기호, 문자, 항(terms), 또는 숫자 등으로 나타내는 것이 편리한 것으로 입증되었다.

[0019] 그러나, 이러한 용어 및 유사 용어는 모두 적절한 물리량과 관련될 것이며 단지 이러한 양에 적용된 편리한 표시에 불과하다는 것을 유념해야 할 것이다. 다음의 설명으로부터 명백하듯이 특별히 달리 언급되지 않는 한, 본 발명 전체에 걸쳐, "처리하는" 또는 "액세스하는" 또는 "기록하는" 또는 "저장하는" 또는 "복제하는" 등과 같은 용어를 이용하는 설명은 컴퓨터 시스템, 또는 컴퓨터 시스템의 레지스터 및 메모리 및 다른 컴퓨터 판독가능한 매체 내의 물리(전자)량으로 표현된 데이터를 조작하여 컴퓨터 시스템 메모리 또는 레지스터 또는 다른 그러한 정보 저장, 전송 또는 디스플레이 장치 내의 물리량으로 유사하게 표현된 다른 데이터로 변환하는 유사 전자 컴퓨팅 장치의 동작 및 처리를 나타낸다는 것을 이해할 것이다.

[0020] 일 실시예에서, 본 발명은 마이크로프로세서에서 코드 최적화를 가속화하는 방법으로서 구현된다. 본 방법은 명령어 페치(fetch) 컴포넌트를 이용하여 인입(incoming) 매크로명령어 시퀀스를 페치하는 단계 및 페치된 매크로명령어를 마이크로명령어로 디코딩하는 디코딩 컴포넌트로 전달하는 단계를 포함한다. 마이크로명령어 시퀀스를 복수의 의존적 코드 그룹을 포함하는 최적화된 마이크로명령어 시퀀스로 재순서화(reordering)함으로써 최적화 처리가 수행된다. 최적화된 마이크로명령어 시퀀스는 실행을 위해 마이크로프로세서 파이프라인으로 출력된다. 최적화된 마이크로명령어 시퀀스에 대한 후속 히트(hit)시 후속 사용을 위해 최적화된 마이크로명령어 시퀀스의 사본이 시퀀스 캐시에 저장된다.

[0021] 도 1은 본 발명의 일 실시예에 따른 마이크로프로세서(100)의 할당/발행(issue) 스테이지의 개요도를 도시한다. 도 1에 예시된 바와 같이, 마이크로프로세서(100)는 페치 컴포넌트(101), 네이티브(native) 디코드 컴포넌트(102), 및 명령어 스케줄링 및 최적화 컴포넌트(110) 및 마이크로프로세서의 나머지 파이프라인(105)을 포함한다.

[0022] 도 1의 실시예에서, 페치 컴포넌트(101)에 의해 매크로명령어가 페치되고 네이티브 디코드 컴포넌트(102)에 의해 네이티브 마이크로명령어로 디코드되며, 그 다음에 마이크로명령어를 마이크로명령어 캐시(121) 및 명령어 스케줄링 및 최적화기 컴포넌트(110)로 제공한다. 일 실시예에서, 페치된 매크로명령어는 소정의 분기(branches)를 예측함으로써 어셈블링된(assembled) 명령어 시퀀스를 포함한다.

[0023] 매크로명령어 시퀀스는 네이티브 디코드 컴포넌트(102)에 의해 결과적인 마이크로명령어 시퀀스로 디코딩된다. 다음에, 이러한 마이크로명령어 시퀀스는 멀티플렉서(103)를 통해 명령어 스케줄링 및 최적화 컴포넌트(110)로 전송된다. 명령어 스케줄링 및 최적화기 컴포넌트는, 예를 들어, 더 효율적인 실행을 위해 마이크로명령어 시퀀스의 특정 명령어를 재순서화하여 최적화 처리를 수행함으로써 작동한다. 이는 결과적으로 최적화된 마이크로명령어 시퀀스를 얻고 이것은 그 다음에 멀티플렉서(104)를 통해 나머지 파이프라인(105)(예를 들어, 할당, 디스패치(dispatch), 실행, 및 방출(retirement) 스테이지 등)으로 전달된다. 최적화된 마이크로명령어 시퀀스는 명령어를 더 빠르고 더 효율적으로 실행하게 한다.

[0024] 일 실시예에서, 매크로명령어는 고레벨 명령어 세트 아키텍처로부터의 명령어일 수 있는 반면, 마이크로명령어는 저레벨 머신 명령어이다. 다른 실시예에서, 매크로명령어는 복수의 상이한 명령어 세트 아키텍처(예를 들어, CISC 계열, x86 RISC 계열, MIPS, SPARC, ARM, 가상 계열, 및 JAVA 등)로부터의 게스트(guest) 명령어일 수 있는 반면, 마이크로명령어는 저레벨 머신 명령어 또는 상이한 네이티브 명령어 세트 아키텍처의 명령어이다. 유사하게, 일 실시예에서, 매크로명령어는 어떤 아키텍처의 네이티브 명령어일 수 있고, 마이크로명령어는 재순서화되고 최적화된 그와 동일한 아키텍처의 네이티브 마이크로명령어일 수 있다. 예를 들어, X86 매크로명령어 및 X86 마이크로 코딩된 마이크로명령어이다.

[0025] 일 실시예에서, 빈번히 발생되는 코드(예를 들어, 핫코드(hot code))의 실행 성능을 가속화하기 위해, 빈번히 발생되는 마이크로명령어 시퀀스의 사본이 마이크로명령어 캐시(121)에 캐시되고 빈번히 발생되는 최적화된 마이크로명령어 시퀀스의 사본이 시퀀스 캐시(122) 내에 캐시된다. 코드가 페치되고, 디코딩되고, 최적화되고, 실행됨에 따라, 특정 최적화된 마이크로명령어 시퀀스는 도시된 축출(eviction) 및 채움(fill) 경로(130)를 통

해 시퀀스 캐시의 크기에 따라 축출되거나 페치될 수 있다. 이러한 축출 및 채움 경로는 최적화된 마이크로명령어 시퀀스를 마이크로프로세서의 메모리 계층(예를 들어, L1 캐시, L2 캐시, 또는 특수 캐시 가능 메모리 범위 등)으로 및 그로부터 전달하게 해준다.

[0026] 일 실시예에서, 마이크로명령어 캐시(121)는 생략될 수 있음이 주목되어야 한다. 이러한 실시예에서, 핫코드의 가속은 시퀀스 캐시(122) 내에 최적화된 마이크로명령어 시퀀스의 저장에 의해 제공된다. 예를 들어, 마이크로명령어 캐시(121)를 생략하여 절약된 공간은, 예를 들어, 더 큰 시퀀스 캐시(122)를 구현하는데 사용될 수 있다.

[0027] 도 2는 본 발명의 일 실시예에 따른 최적화 처리를 예시하는 개요도를 도시한다. 도 2의 좌측은, 예를 들어, 네이티브 디코드 컴포넌트(102) 또는 마이크로명령어 캐시(121)에서 수신된 것으로서 인입 마이크로명령어 시퀀스를 보여준다. 처음에 이들 명령어의 수신시, 이들은 최적화되지 않는다.

[0028] 최적화 처리의 한가지 목적은 서로 의존하는 명령어를 찾고 그 명령어를 식별하며 이들이 좀 더 효율적으로 실행할 수 있도록 이들을 이들 각각의 의존성 그룹(dependency groups)으로 이동시키는 것이다. 일 실시예에서, 의존성 명령어 그룹은 집약성(locality)을 위해 이들 각각의 소스 및 목적지가 함께 그룹화되기 때문에 이들이 좀 더 효율적으로 실행할 수 있도록 함께 디스패치(dispatch)될 수 있다. 이러한 최적화 처리는 비순차적 프로세서뿐만 아니라 순차적 프로세서 둘 다에서 사용될 수 있음이 주목되어야 한다. 예를 들어, 순차적 프로세서 내에서, 명령어는 순차적으로 디스패치된다. 그러나, 전술한 바와 같이 이들은 의존성 명령어들이 각 그룹에 배치되어 그룹들이 독립적으로 실행할 수 있도록 이동될 수 있다.

[0029] 예를 들어, 인입 명령어는, 로드(loads), 연산(operations) 및 스토어(stores)를 포함한다. 예를 들어, 명령어 1은 소스 레지스터(예를 들어, 레지스터 9 및 레지스터 9를 추가하고 그 결과를 레지스터 5에 저장하는 연산을 포함한다. 따라서, 레지스터 5는 목적지이고 레지스터 9 및 레지스터 5는 소스이다. 이러한 방식으로, 16개 명령어의 시퀀스는 도시된 바와 같이 목적지 레지스터 및 소스 레지스터를 포함한다.

[0030] 도 2의 실시예는 하나의 그룹에 속하는 명령어들이 서로 의존하는 의존성 그룹을 생성하는 명령어의 재순서화를 구현한다. 이를 달성하기 위해, 16개의 인입 명령어의 로드 및 스토어에 대해 위험성 체크를 수행하는 알고리즘이 실행된다. 예를 들어, 스토어는 의존성 체크 없이 이전 로드를 이동할 수 없다. 스토어는 이전 스토어를 통과할 수 없다. 로드는 의존성 체크 없이 이전 스토어를 통과할 수 없다. 로드는 로드를 통과할 수 있다. 명령어는 재명명(renaming) 기술을 이용하여 사전 경로 예측 분기(prior path predicted branch)(예를 들어, 동적으로 구축된 분기)를 통과할 수 있다. 비-동적으로 예측된 분기의 경우, 명령어의 이동은 분기의 범위를 고려할 필요가 있다. 전술한 각 규칙은 가상 의존성을 추가함으로써(예를 들어, 명령어에 가상 소스 또는 목적지를 인위적으로 추가하여 규칙을 시행함으로써) 구현될 수 있다.

[0031] 계속해서 도 2를 참조하면, 전술한 바와 같이, 최적화 처리의 목적은 의존성 명령어의 위치를 파악하고 이들을 공통 의존성 그룹으로 이동하는 것이다. 이러한 처리는 위험성 체크 알고리즘에 따라 행해져야 한다. 최적화 알고리즘은 명령어 의존성을 찾는 것이다. 명령어 의존성은 사실 의존성(true dependencies), 출력 의존성(output dependencies) 및 반 의존성(anti-dependencies)을 더 포함한다.

[0032] 이러한 알고리즘은 먼저 사실 의존성을 찾음으로써 시작한다. 사실 의존성을 식별하기 위해, 16개의 명령어 시퀀스의 각 목적지는 16개의 명령어 시퀀스에서 나중에 일어나는 다른 후속 소스와 비교된다. 이전 명령어에 정말로 의존하는(truly dependent) 후속 명령어는 "_1"로 표시되어 이들의 사실 의존성을 나타낸다. 이는 도 2에서 16개의 명령어 시퀀스에 걸쳐 좌측에서 우측으로 진행하는 명령어 번호로 도시된다. 예를 들어, 명령어 번호 4를 고려하면, 목적지 레지스터 R3는 후속 명령어의 소스와 비교되고, 각 후속 소스는 "_1"로 표시되어 그 명령어의 사실 의존성을 나타낸다. 이 경우, 명령어 6, 명령어 7, 명령어 11, 및 명령어 15는 "_1"로 표시된다.

[0033] 다음에, 알고리즘은 출력 의존성을 찾는다. 출력 의존성을 식별하기 위해, 각 목적지는 다른 후속 명령어의 목적지와 비교된다. 그리고 16개 명령어 각각에 대해, 매치하는 각 후속 목적지는 "1_"로 표시된다(예를 들어, 때때로 적색 1로 나타냄).

[0034] 다음에, 알고리즘은 반 의존성을 찾는다. 반 의존성을 식별하기 위해, 16개 명령어 각각에 대해, 각 소스는 이전 명령어의 소스와 비교되어 매치를 식별한다. 매치가 일어나면, 고려 중에 있는 명령어는 그 자체를 "1_"로 표시된다(예를 들어, 때때로 적색 1로 나타냄).

[0035] 이러한 방식으로, 알고리즘은 16개 명령어의 시퀀스에 대해 행(rows) 및 열(columns)의 의존성 매트릭스를 채운

다(populate). 의존성 매트릭스는 16개 명령어 각각에 대해 서로 다른 형태의 의존성을 나타내는 표시를 포함한다. 일 실시예에서, 의존성 매트릭스는 한 사이클 내에 CAM 매칭 하드웨어 및 적절한 브로드캐스팅 로직을 이용하여 채워진다. 예를 들어, 목적지는 나머지 명령어를 통해 하향 브로드캐스팅되어 후속 명령어의 소스(예를 들어, 사실 의존성) 및 후속 명령어의 목적지(예를 들어, 출력 의존성)와 비교되는 반면, 목적지는 이전 명령어를 통해 상향 브로드캐스팅되어 이전 명령어의 소스(예를 들어, 반 의존성)와 비교될 수 있다.

[0036] 최적화 알고리즘은 의존성 매트릭스를 이용하여 어느 명령어를 함께 공통 의존성 그룹으로 이동할지 선택한다. 서로 정말로 의존하는 명령어들이 동일한 그룹으로 이동하는 것이 바람직하다. 레지스터 재명명은 반 의존성을 제거하여 그 반 의존성 명령어를 이동시키는데 사용된다. 이동은 전술한 규칙 및 위험성 체크에 따라 행해진다. 예를 들어, 스토어는 의존성 체크 없이 이전 로드를 이동할 수 없다. 스토어는 이전 스토어를 통과할 수 없다. 로드는 의존성 체크 없이 이전 스토어를 통과할 수 없다. 로드는 로드를 통과할 수 있다. 명령어는 재명명 기술을 이용하여 사전 경로 예측 분기(예를 들어, 동적으로 구축된 분기)를 통과할 수 있다. 비-동적으로 예측된 분기의 경우, 명령어의 이동은 분기의 범위를 고려할 필요가 있다. 설명에 주목한다.

[0037] 일 실시예에서, 어느 명령어를 다른 명령어와 그룹화하도록 이동시킬지 판단하기 위해 우선순위(priority) 인코더가 구현될 수 있다. 우선순위 인코더는 의존성 매트릭스에 의해 제공된 정보에 따라 작동할 것이다.

[0038] 도 3 및 도 4는 본 발명의 일 실시예에 따른 다단계의 최적화 처리를 도시한다. 일 실시예에서, 최적화 처리는 명령어가 제1 패스(pass)에서 이들의 의존성 열을 이동시킴으로써 이동된 후, 명령어를 이동시킬 새로운 기회에 대해 의존성 매트릭스가 다시 채워지고 다시 검사된다는 점에서 반복적이다. 일 실시예에서, 이러한 의존성 매트릭스 채움(population) 처리는 세 번 반복된다. 이에 대해서는 도 4에 도시되고, 이 도면은 이동된 다음 검사되어 다른 명령어를 이동시킬 기회를 다시 찾는 명령어를 보여준다. 16개 명령어 각각의 우측 상의 번호 시퀀스는 그 명령어가 처리를 시작했을 때에 있던 그룹 및 그 명령어가 처리의 마지막에 있던 그룹을 보여주며, 이때 중간 그룹 번호가 그 사이에 있다. 예를 들어, 도 4는 명령어 6이 처음에 그룹 4에 있었지만 이동하여 그룹 1에 있는 것을 도시한다.

[0039] 이러한 방식으로, 도 2 내지 도 4는 본 발명의 일 실시예에 따른 최적화 알고리즘의 동작을 예시한다. 비록 도 2 내지 도 4가 할당/발행 스테이지를 예시하지만, 이러한 기능성은 또한 로컬 스케줄러/디스패치 스테이지에서도 구현될 수 있음이 주목되어야 한다.

[0040] 도 5는 본 발명의 일 실시예에 따른 예시적인 하드웨어 최적화 처리(500)의 단계 흐름도를 도시한다. 도 5에 도시된 바와 같이, 흐름도는 본 발명의 일 실시예에 따라 마이크로프로세서의 할당/발행 스테이지에서 구현된 바와 같은 최적화 처리의 동작 단계를 도시한다.

[0041] 처리(500)는 단계(501)에서 시작하고, 이 단계에서 명령어 페치 컴포넌트(예를 들어, 도 1의 페치 컴포넌트(20))를 이용하여 인입 매크로명령어 시퀀스가 페치된다. 전술한 바와 같이, 페치된 명령어는 특정 명령어 분기를 예측함으로써 어셈블링된 시퀀스를 포함한다.

[0042] 단계(502)에서, 페치된 매크로명령어는 마이크로명령어로 디코딩하는 디코딩 컴포넌트로 전달된다. 매크로명령어 시퀀스는 분기 예측에 따라 마이크로명령어 시퀀스로 디코딩된다. 일 실시예에서, 마이크로명령어 시퀀스는 이어서 마이크로명령어 캐시에 저장된다.

[0043] 다음에, 단계(503)에서, 시퀀스를 포함하는 마이크로명령어를 의존성 그룹으로 재순서화함으로써 마이크로명령어 시퀀스에 대해 최적화 처리가 수행된다. 재순서화는 명령어 재순서화 컴포넌트(예를 들어, 명령어 스케줄링 및 최적화기 컴포넌트(110))에 의해 구현된다. 이러한 처리는 도 2 내지 도 4에서 설명된다.

[0044] 단계(504)에서, 최적화된 마이크로명령어 시퀀스는 실행을 위해 마이크로프로세서 파이프라인으로 출력된다. 전술한 바와 같이, 최적화된 마이크로명령어 시퀀스는 실행을 위해 머신의 나머지 부분(예를 들어, 나머지 파이프라인(105))으로 전달된다.

[0045] 그 다음에, 단계(505)에서, 최적화된 마이크로명령어 시퀀스의 사본이 그 시퀀스에 대한 후속 히트시 후속 사용을 위해 시퀀스 캐시에 저장된다. 이러한 방식으로, 시퀀스 캐시는 그 시퀀스에 대한 후속 히트시 최적화된 마이크로명령어 시퀀스에의 액세스를 가능하게 하고, 이로써 핫 코드를 가속화한다.

[0046] 도 6은 본 발명의 일 실시예에 따른 대안의 예시적인 하드웨어 최적화 처리(600)의 단계 흐름도를 도시한다. 도 6에 도시된 바와 같이, 흐름도는 본 발명의 대안의 실시예에 따라 마이크로프로세서의 할당/발행 스테이지에서 구현되는 바와 같은 최적화 처리의 동작 단계를 도시한다.

[0047] 처리(600)는 단계(601)에서 시작하고, 이 단계에서 명령어 페치 컴포넌트(예를 들어, 도 1의 페치 컴포넌트(20))를 이용하여 인입 매크로명령어 시퀀스가 페치된다. 전술한 바와 같이, 페치된 명령어는 특정 명령어 분기를 예측함으로써 어셈블링된 시퀀스를 포함한다.

[0048] 단계(602)에서, 페치된 매크로명령어는 마이크로명령어로 디코딩하는 디코딩 컴포넌트로 전달된다. 매크로명령어 시퀀스는 분기 예측에 따라 마이크로명령어 시퀀스로 디코딩된다. 일 실시예에서, 마이크로명령어 시퀀스는 이어서 마이크로명령어 캐시에 저장된다.

[0049] 단계(603)에서, 디코딩된 마이크로명령어는 마이크로명령어 시퀀스 캐시 내의 시퀀스로 저장된다. 마이크로명령어 캐시 내의 시퀀스는 기본적인 블록 경계에 따라 시작하도록 형성된다. 이때 이러한 시퀀스는 최적화되지 않는다.

[0050] 다음에, 단계(604)에서, 시퀀스를 포함하는 마이크로명령어를 의존성 그룹으로 재순서화함으로써 마이크로명령어 시퀀스에 대해 최적화 처리가 수행된다. 재순서화는 명령어 재순서화 컴포넌트(예를 들어, 명령어 스케줄링 및 최적화기 컴포넌트(110))에 의해 구현된다. 이러한 처리는 도 2 내지 도 4에서 설명된다.

[0051] 단계(605)에서, 최적화된 마이크로명령어 시퀀스는 실행을 위해 마이크로프로세서 파이프라인으로 출력된다. 전술한 바와 같이, 최적화된 마이크로명령어 시퀀스는 실행을 위해 머신의 나머지 부분(예를 들어, 나머지 파이프라인(105))으로 전달된다.

[0052] 그 다음에, 단계(606)에서, 최적화된 마이크로명령어 시퀀스의 사본이 그 시퀀스에 대한 후속 히트시 후속 사용을 위해 시퀀스 캐시에 저장된다. 이러한 방식으로, 시퀀스 캐시는 그 시퀀스에 대한 후속 히트시 최적화된 마이크로명령어 시퀀스에의 액세스를 가능하게 하고, 이로써 핫 코드를 가속화한다.

[0053] 도 7은 본 발명의 일 실시예에 따른 할당/발행 스테이지의 CAM 매칭 하드웨어 및 우선순위 인코딩 하드웨어의 동작을 도시하는 도면을 도시한다. 도 7에 도시된 바와 같이, 명령어의 목적지는 좌측으로부터 CAM 어레이로 브로드캐스팅된다. 세 개의 예시적인 명령어 목적지가 도시된다. 더 밝은 음영 CAM(예를 들어, 녹색)은 사실 의존성 매치 및 출력 의존성 매치에 대한 것이고, 따라서 그 목적지는 하향 브로드캐스팅된다. 더 어두운 음영 CAM(예를 들어, 청색)은 반 의존성 매치에 대한 것이고, 따라서 그 목적지는 상향 브로드캐스팅된다. 이들 매치는 전술한 바와 같이 의존성 매트릭스를 채운다. 우선순위 인코더는 우측에 도시되고, 이들은 CAM의 행을 스캔하여 첫 번째 매치, "_1" 또는 "1_"를 찾음으로써 작동한다. 도 2 내지 도 4의 설명에서 전술한 바와 같이, 처리는 반복적인 것으로 구현될 수 있다. 예를 들어, 만일 "_1"이 "1_"에 의해 차단되면, 그 목적지는 재명명되고 이동될 수 있다.

[0054] 도 8은 본 발명의 일 실시예에 따른 분기 앞에 최적화된 스케줄링 명령어를 예시하는 도면을 도시한다. 도 8에 예시된 바와 같이, 전통적인 저스트 인 타임(just-in-time; JIT) 컴파일러 예와 함께 하드웨어 최적화된 예가 도시된다. 도 8의 좌측은 "Branch C to L1"이라는 미획득 분기 바이어스(branch biased untaken)를 포함하는 원래의 최적화되지 않은 코드를 보여준다. 도 8의 중간 열은 전통적인 저스트 인 타임 컴파일러 최적화를 보여주고, 여기서 레지스터는 재명명되고 명령어는 분기 앞으로 이동된다. 이러한 예에서, 저스트 인 타임 컴파일러는 분기 바이어스 판단이 잘못된 경우(예를 들어, 분기가 실제로는 미획득이 아니라 획득된 경우)를 설명하는 보상 코드를 삽입한다. 반대로, 도 8의 우측 열은 하드웨어 언롤된(unrolled) 최적화를 보여준다. 이 경우, 레지스터가 재명명되고 명령어가 분기 앞으로 이동된다. 그러나, 어떤 보상 코드도 삽입되지 않음이 주목되어야 한다. 하드웨어는 분기 바이어스 판단이 사실인지 여부를 추적한다. 잘못 예측된 분기의 경우, 하드웨어는 정확한 명령어 시퀀스를 실행하기 위해 그의 상태를 자동으로 롤백(roll back)한다. 하드웨어 최적화기 해결책은 분기가 잘못 예측된 경우에 하드웨어가 잘못 예측된 명령어 시퀀스를 플러싱(flushing)하면서 메모리 내의 원래 코드로 점프하고 그로부터 정확한 시퀀스를 실행하기 때문에 보상 코드의 사용을 피할 수 있다.

[0055] 도 9는 본 발명의 일 실시예에 따른 스토어 앞에 로드의 최적화된 스케줄링을 예시하는 도면을 도시한다. 도 9에 예시된 바와 같이, 전통적인 저스트 인 타임 컴파일러 예와 함께 하드웨어 최적화된 예가 도시된다. 도 9의 좌측은 스토어 "R3<- LD[R5]"를 포함하여 최적화되지 않은 원래의 코드를 보여준다. 도 9의 중간 열은 전통적인 저스트 인 타임 컴파일러 최적화를 보여주고, 여기서 레지스터는 재명명되고 로드는 스토어 앞으로 이동된다. 이 예에서, 저스트 인 타임 컴파일러는 로드 명령어의 어드레스가 스토어 명령어의 어드레스를 엘리어싱(alias)하는 경우(예를 들어, 스토어 앞에 로드 이동이 적절하지 않은 경우)를 설명하는 보상 코드를 삽입한다. 반대로, 도 9의 우측 열은 하드웨어 언롤된 최적화를 보여준다. 이 경우, 레지스터는 재명명되고 로드는 또한 스토어 앞으로 이동된다. 그러나, 어떤 보상 코드도 삽입되지 않음이 주목되어야 한다. 스토어 앞으

로 로드를 이동시키는 것이 잘못된 경우, 하드웨어는 정확한 명령어 시퀀스를 실행하기 위해 그의 상태를 자동으로 롤백한다. 하드웨어 최적화기 해결책은 어드레스 엘리어스 체크 분기(address alias-check branch)가 잘못 예측된 경우 하드웨어는 잘못 예측된 명령어 시퀀스를 플러싱하면서 메모리 내의 원래 코드로 점프하고 그로부터 정확한 시퀀스를 실행하기 때문에 보상 코드의 사용을 피할 수 있다. 이 경우, 시퀀스는 엘리어싱이 없다고 가정한다. 일 실시예에서, 도 9에 도식화된 기능성은 도 1의 명령어 스케줄링 및 최적화기 컴포넌트(110)에 의해 구현될 수 있음이 주목되어야 한다. 마찬가지로, 일 실시예에서, 도 9에 도식화된 기능성은 아래의 도 10에 설명된 소프트웨어 최적화기(1000)에 의해서도 구현될 수 있음이 주목되어야 한다.

[0056] 추가적으로, 동적으로 언롤된 시퀀스에 대해, 명령어는 재명명을 이용하여 사전 경로 예측 분기(예를 들어, 동적으로 구축된 분기)를 통과할 수 있음이 주목되어야 한다. 비-동적으로 예측된 분기의 경우, 명령어의 이동은 분기의 범위를 고려해야 한다. 루프는 원하는 범위까지 언롤될 수 있고 최적화는 시퀀스 전체에 걸쳐 적용될 수 있다. 예를 들어, 이는 분기 전체에 걸쳐 이동하는 명령어의 목적지 레지스터를 재명명함으로써 구현될 수 있다. 이러한 특징의 이점들 중 하나는 분기의 범위의 어떤 보상 코드 또는 광범위한 분석도 필요하지 않다는 사실이다. 따라서, 이러한 특징은 속도를 크게 높이고 최적화 처리를 간략화한다.

[0057] 분기 예측 및 명령어 시퀀스의 어셈블링에 관한 추가 정보는 2010년 9월 17일 모하메드 에이. 압달라(Mohammad A. Abdallah)에 의해 "SINGLE CYCLE MULTI-BRANCH PREDICTION INCLUDING SHADOW CACHE FOR EARLY FAR BRANCH PREDICTION"이라는 명칭으로 출원되어 본 출원과 공동으로 양도된 미국 특허 출원 제61/384,198호에서 찾아볼 수 있으며, 이 출원은 그 전체가 본 명세서에서 인용된다.

[0058] 도 10은 본 발명의 일 실시예에 따른 예시적인 소프트웨어 최적화 처리의 도면을 도시한다. 도 10의 실시예에서, 명령어 스케줄링 및 최적화기 컴포넌트(예를 들어, 도 1의 컴포넌트(110))는 소프트웨어 기반 최적화기(1000)로 대체된다.

[0059] 도 10의 실시예에서, 소프트웨어 최적화기(1000)는 하드웨어 기반 명령어 스케줄링 및 최적화기 컴포넌트(110)에 의해 수행된 최적화 처리를 수행한다. 소프웨어 최적화기는 최적화된 시퀀스의 사본을 메모리 계층(예를 들어, L1, L2, 시스템 메모리)에 유지한다. 이렇게 하면 소프트웨어 최적화기가 시퀀스 캐시에 저장된 것과 비교하여 최적화된 시퀀스의 집합(collection)을 훨씬 더 크게 유지하는 것이 가능해진다.

[0060] 소프트웨어 최적화기(1000)는 최적화에 입력되고 최적화 처리로부터 출력된 것으로서 메모리 계층에 상주하는 코드를 포함할 수 있음이 주목되어야 한다.

[0061] 일 실시예에서, 마이크로명령어 캐시는 생략될 수 있음이 주목되어야 한다. 이러한 실시예에서, 단지 최적화된 마이크로명령어 시퀀스만 캐시된다.

[0062] 도 11은 본 발명의 일 실시예에 따른 SIMD 소프트웨어 기반 최적화 처리의 흐름도를 도시한다. 도 11의 윗부분은 소프트웨어 기반 최적화기가 입력 명령어 시퀀스의 각 명령어를 어떻게 검사하는지를 보여준다. 도 11은 SIMD 비교가 하나 를 다수에 매칭(예를 들어, SIMD 바이트가 제1 소스 "Src1"를 모든 제2 소스 바이트 "Src2"와 비교)하는데 어떻게 사용될 수 있는지를 보여준다. 일 실시예에서, Src1은 임의의 명령어의 목적지 레지스터를 포함하고 Src2는 각 다른 후속 명령어로부터의 하나의 소스를 포함한다. 매칭은 모든 목적지에 대해 모든 후속 명령어 소스와 이루어진다(예를 들어, 사실 의존성 체크). 이는 해당 명령어에 대해 원하는 그룹을 나타내는 페어링(pairing) 매치이다. 매칭은 각 목적지와 모든 후속 명령어 목적지 사이에서 이루어진다(예를 들어, 출력 의존성 체크). 이는 재명명으로 해결(resolve)될 수 있는 블로킹 매치이다. 매칭은 각 목적지와 모든 이전의 명령어 소스 사이에서 이루어진다(예를 들어, 반 의존성 체크). 이는 재명명에 의해 해결될 수 있는 블로킹 매치이다. 그 결과는 의존성 매트픽스의 행 및 열을 채우는데 사용된다.

[0063] 도 12는 본 발명의 일 실시예에 따른 예시적인 SIMD 소프트웨어 기반 최적화 처리(1200)의 동작 단계 흐름도를 도시한다. 처리(1200)는 도 9의 흐름도의 문맥에서 설명된다.

[0064] 단계(1201)에서, 입력 명령어 시퀀스는 메모리에 인스턴스화된(instantiated) 소프트웨어 기반 최적화기를 이용하여 액세스된다.

[0065] 단계(1202)에서, 의존성 매트릭스는 SIMD 명령어를 이용하여, SIMD 비교 명령어 시퀀스를 이용하여 입력 명령어 시퀀스에서 추출된 의존성 정보로 채워진다.

[0066] 단계(1203)에서, 제1 매치에 대해 매트릭스의 행들이 우측에서 좌측으로 스캔된다(예를 들어, 의존성 표시).

[0067] 단계(1204)에서, 각 제1 매치가 분석되어 그 매치의 유형을 결정한다.

[0068] 단계(1205)에서, 만일 제1 표시된 매치가 블로킹 의존성이면, 이 목적지에 대해 재명명이 행해진다.

[0069] 단계(1206)에서, 매트릭스의 각 행에 대해 모든 제1 매치가 식별되고 그 매치에 대해 대응하는 열이 주어진 의존성 그룹으로 이동된다.

[0070] 단계(1207)에서, 스캐닝 처리가 여러 번 반복되어 입력 시퀀스를 포함하는 명령어를 재순서화하여 최적화된 출력 시퀀스를 생성한다.

[0071] 단계(1208)에서, 최적화된 명령어 시퀀스는 실행을 위해 마이크로프로세서의 실행 파이프라인으로 출력된다.

[0072] 단계(1209)에서, 최적화된 출력 시퀀스는 후속 사용을 위해(예를 들어, 핫코드를 가속화하기 위해) 시퀀스 캐시에 저장된다.

[0073] 소프트웨어 최적화는 SIMD 명령어의 사용과 함께 연속적으로 행해질 수 있음이 주목되어야 한다. 예를 들어, 최적화는 한번에 명령어의 소스 및 목적지를 스캐닝하는 하나의 명령어를 처리함으로써(예를 들어, 시퀀스에서 이전 명령어부터 후속 명령어까지) 구현될 수 있다. 소프트웨어는 전술한 최적화 알고리즘 및 SIMD 명령어에 따라 SIMD 명령어를 이용하여 현재 명령어 소스 및 목적지를 이전 명령어 소스 및 목적지와 병렬로 비교한다(예를 들어, 사실 의존성, 출력 의존성 및 반 의존성을 검출한다).

[0074] 도 13은 본 발명의 일 실시예에 따른 소프트웨어 기반 의존성 브로드캐스팅 처리를 도시한다. 도 13의 실시예는 전술한 바와 같이 완전 병렬(full parallel) 하드웨어 구현의 비용 부담없이 명령어 그룹을 처리하는 예시적인 소프트웨어 스케줄링 처리의 흐름도를 도시한다. 그러나, 도 13의 실시예는 여전히 SIMD를 이용하여 더 작은 명령어 그룹들을 병렬로 처리할 수 있다.

[0075] 도 13의 소프트웨어 스케줄링 처리는 다음과 같이 진행된다. 먼저, 처리는 세 개의 레지스터를 초기화한다. 처리는 명령어 번호를 취하고 이들을 제1 레지스터에 로드한다. 다음에, 처리는 목적지 레지스터 번호를 취하고 이들을 제2 레지스터에 로드한다. 다음에, 처리는 제1 레지스터 내의 값들을 취하고 이들을 제2 레지스터 내의 위치 번호에 따라 제3 결과 레지스터 내의 위치로 브로드캐스팅한다. 다음에, 처리는 제2 레지스터 내에서 좌측에서 우측으로 진행하여 오버라이트(over write)하고, 맨 좌측 값은 브로드캐스팅이 결과 레지스터 내의 동일 위치로 진행하는 경우에 우측 값을 오버라이트할 것이다. 제3 레지스터 내에서 기록되지 않은 위치는 바이패스된다. 이러한 정보는 의존성 매트릭스를 채우는데 사용된다.

[0076] 도 13의 실시예는 입력 명령어 시퀀스가 복수의 그룹으로 처리될 수 있는 방식을 또한 보여준다. 예를 들어, 16개의 명령어 입력 시퀀스는 8개 명령어의 제1 그룹 및 8개 명령어의 제2 그룹으로 처리될 수 있다. 제1 그룹에서, 명령어 번호는 제1 레지스터에 로드되고, 명령어 목적지 번호는 제2 레지스터에 로드되며, 제1 레지스터 내의 값들은 제2 레지스터 내의 위치 번호에 따라 제3 레지스터(예를 들어, 결과 레지스터) 내의 위치로 브로드캐스팅된다(예를 들어, 그룹 브로드캐스팅). 제3 레지스터 내에서 기록되지 않은 위치는 바이패스된다. 이제 제3 레지스터는 제2 그룹의 처리를 위한 기준(base)이 된다. 예를 들어, 그룹 1의 결과 레지스터는 이제 그룹 2의 처리를 위한 결과 레지스터가 된다.

[0077] 제2 그룹에서, 명령어 번호는 제1 레지스터에 로드되고, 명령어 목적지 번호는 제2 레지스터에 로드되며, 제1 레지스터 내의 값들은 제2 레지스터 내의 위치 번호에 따라 제3 레지스터(예를 들어, 결과 레지스터) 내의 위치로 브로드캐스팅된다. 제3 레지스터 내의 위치는 제1 그룹의 처리 동안 기록된 결과를 오버라이트할 수 있다. 제3 레지스터 내에서 기록되지 않은 위치는 바이패스된다. 이러한 방식으로, 제2 그룹은 제1 그룹으로부터의 기준을 업데이트하고, 그럼으로써 제3 그룹 등의 처리를 위한 새로운 기준을 생성한다.

[0078] 제2 그룹 내의 명령어는 제1 그룹의 처리에서 생성된 의존성 정보를 승계(inherit)할 수 있다. 결과 레지스터 내의 의존성을 업데이트하기 위해 제2 그룹 전체가 처리되지 않아도 된다는 점이 주목되어야 한다. 예를 들어, 명령어 12에 대한 의존성은 제1 그룹의 처리에서, 그 다음 제2 그룹에서 명령어 11까지 명령어 처리에서 생성될 수 있다. 이는 결과 레지스터를 명령어 12까지의 상태로 업데이트한다. 일 실시예에서, 제2 그룹의 나머지 명령어(예를 들어, 명령어 12 내지 16)에 대한 업데이트를 막기 위해 마스크가 사용될 수 있다. 명령어 12에 대한 의존성을 판단하기 위해, R2 및 R5에 대해 결과 레지스터가 검사된다. R5는 명령어 1로 업데이트될 것이고, R2는 명령어 11로 업데이트될 것이다. 그룹 2가 모두 처리된 경우, R2는 명령어 15로 업데이트될 것임이 주목되어야 한다.

[0079] 추가로, 제2 그룹의 모든 명령어(예를 들어, 명령어 9-16)는 서로 독립적으로 처리될 수 있음이 주목되어야 한

다. 이러한 경우, 제2 그룹의 명령어들은 단지 제1 그룹의 결과 레지스터에만 의존한다. 일단 결과 레지스터가 제1 그룹의 처리로부터 업데이트되면 제2 그룹의 명령어들은 병렬로 처리될 수 있다. 이러한 방식으로, 명령어 그룹들은 차례대로(one after another) 병렬로 처리될 수 있다. 일 실시예에서, 각 그룹은 SIMD 명령어(예를 들어, SIMD 브로드캐스팅 명령어)를 이용하여 처리되고, 그럼으로써 상기 각 그룹의 모든 명령어들을 병렬로 처리할 수 있다.

[0080] 도 14는 본 발명의 일 실시예에 따라 명령어의 의존성 그룹화가 가변 제한된 의존성 명령어 그룹을 구축하는데 어떻게 사용될 수 있는지를 보여주는 예시적인 흐름도를 도시한다. 도 2 내지 도 4의 설명에서, 그룹 크기는, 그 경우 그룹 당 세 개의 명령어로 제한된다. 도 14는 명령어들이 어떻게 가변 크기의 그룹들로 재순서화될 수 있고, 그 다음에 복수의 컴퓨팅 엔진에 할당될 수 있는지를 도시한다. 예를 들어, 도 14는 4개의 엔진을 도시한다. 그룹들은 자신들의 특성에 따라 가변 크기를 가질 수 있기 때문에, 엔진 1에는, 예를 들어, 엔진 2보다 더 큰 그룹이 할당될 수 있다. 이는, 예를 들어, 엔진 2가 그 그룹 내 다른 명령어들에 특별히 의존하지 않는 명령어를 갖는 경우에 일어날 수 있다.

[0081] 도 15는 본 발명의 일 실시예에 따른 계층적 명령어 스케줄링을 묘사하는 흐름도를 도시한다. 전술한 바와 같이, 명령어 의존성 그룹화는 가변 제한된 그룹들을 구축하는데 사용될 수 있다. 도 15는 의존성 그룹 내에 다양한 의존성 레벨이 존재하는 특징을 도시한다. 예를 들어, 명령어 1은 이러한 명령어 시퀀스 내의 어떤 다른 명령어에도 의존하지 않으므로, 명령어 1은 L0 의존성 레벨이 된다. 그러나, 명령어 4는 명령어 1에 의존하므로, 명령어 4는 L1 의존성 레벨이 된다. 이러한 방식으로, 명령어 시퀀스의 각 명령어에는 도시된 바와 같이 의존성 레벨이 할당된다.

[0082] 각 명령어의 의존성 레벨은 자원이 의존성 명령어 실행에 확실하게 이용가능한 방식으로 제2 레벨의 계층적 스케줄러에 의해 명령어를 디스패치하는데 사용된다. 예를 들어, 일 실시예에서, L0 명령어는 제2 레벨의 스케줄러(1-4)에 의해 처리되는 명령어 큐(queues)에 로드된다. L0 명령어는 이들이 각 큐 앞에 있도록 로드되고, L1 명령어는 이들이 각 큐에서 후속하도록 로드되고, L2 명령어는 이들에 후속하도록 하는 등으로 로드된다. 이는 도 15에서 L0에서 Ln까지의 의존성 레벨로 도시된다. 스케줄러(1-4)의 계층적 스케줄링은 유리하게 시간 집약성(locality-in-time) 및 명령어 간(instruction-to-instruction) 의존성을 이용하여 스케줄링 판단을 최적의 방식으로 내리게 한다.

[0083] 이러한 방식으로, 본 발명의 실시예는 명령어 시퀀스의 명령어들에 대한 의존성 그룹 슬롯 할당을 시사한다. 예를 들어, 비순차적 마이크로명령어를 구현하기 위해, 명령어 시퀀스의 명령어 디스패칭은 비순차적이다. 일 실시예에서, 각 사이클마다, 명령어 준비도(readiness)가 체크된다. 어떤 명령어가 의존하는 모든 명령어가 이전에 디스패치하였다면 그 명령어는 준비가 된 것이다. 스케줄러 구조는 그 의존성을 체크함으로써 작동한다. 일 실시예에서, 스케줄러는 통합된 스케줄러이고 그 통합된 스케줄러 구조에서 모든 의존성 체크가 수행된다. 다른 실시예에서, 스케줄러 기능성은 복수의 엔진의 실행 유닛의 디스패치 큐 전체에 걸쳐 분산된다. 따라서, 일 실시예에서, 스케줄러는 통합형인 반면, 다른 실시예에서 스케줄러는 분산형이다. 이들 두 해결책에 따라, 사이클마다 디스패치 명령어의 목적지에 대해 각 명령어 소스가 체크된다.

[0084] 따라서, 도 15는 본 발명의 실시예에 의해 수행된 계층적 스케줄링을 도시한다. 전술한 바와 같이, 먼저 의존성 체인(예를 들어, 의존성 그룹)을 형성하기 위해 명령어들이 그룹화된다. 이러한 의존성 체인의 형성은 소프트웨어 또는 하드웨어에 의해 정적으로 또는 동적으로 행해질 수 있다. 일단 이러한 의존성 체인이 형성되었다면, 이들은 엔진으로 분산/디스패치될 수 있다. 이러한 방식으로, 의존성에 의한 그룹화는 순차적으로 형성된 그룹들에 대해 비순차적 스케줄링을 가능하게 한다. 의존성에 의한 그룹화는 또한 전체 의존성 그룹들을 복수의 엔진(예를 들어, 코어 또는 스레드)에 분산한다. 의존성에 의한 그룹화는 또한 전술한 바와 같이 계층적 스케줄링을 용이하게 하기도 하며, 이 경우 의존성 명령어는 제1 단계에서 그룹화된 다음 제2 단계에서 스케줄된다.

[0085] 도 14 내지 도 19에 도식화된 기능성은 명령어들을 그룹화하는 어떤 방법과도 관계없이(예를 들어, 그룹화 기능성이 하드웨어, 소프트웨어 등으로 구현되든) 작동할 수 있음이 주목되어야 한다. 또한, 도 14 내지 도 19에 도시된 의존성 그룹은 독립형 그룹들의 매트릭스를 포함할 수 있으며, 여기서 각 그룹은 의존성 명령어를 더 포함한다. 추가로, 스케줄러는 또한 엔진일 수 있음이 주목되어야 한다. 이러한 실시예에서, 각각의 스케줄러(1-4)는 (예를 들어, 도 22에 도시된 바와 같이 각 세그먼트가 공통 분할 스케줄러를 포함하는 경우) 그의 각 엔진 내에 포함될 수 있다.

[0086] 도 16은 본 발명의 일 실시예에 따른 3 슬롯 의존성 그룹 명령어의 계층적 스케줄링을 묘사하는 흐름도를 도시

한다. 전술한 바와 같이, 명령어 의존성 그룹화는 가변 제한된 그룹들을 구축하는데 사용될 수 있다. 이 실시예에서, 의존성 그룹은 세 개의 슬롯을 포함한다. 도 16은 3 슬롯 의존성 그룹 내에서조차도 다양한 의존성 레벨을 도시한다. 전술한 바와 같이, 명령어 1은 이러한 명령어 시퀀스 내에서 어떤 다른 명령어에도 의존하지 않으므로, 명령어 1은 L0 의존성 레벨이 된다. 그러나, 명령어 4는 명령어 1에 의존하므로, 명령어 4는 L1 의존성 레벨이 된다. 이러한 방식으로, 명령어 시퀀스의 각 명령어에는 도시된 바와 같이 의존성 레벨이 할당된다.

[0087] 전술한 바와 같이, 각 명령어의 의존성 레벨은 자원이 의존성 명령어 실행에 확실하게 이용가능한 방식으로 제2 레벨의 계층적 스케줄러에 의해 명령어를 디스패치하는데 사용된다. L0 명령어는 제2 레벨의 스케줄러(1-4)에 의해 처리되는 명령어 큐에 로드된다. 도 16에서 L0에서 Ln까지의 의존성 레벨로 도시된 바와 같이, L0 명령어는 이들이 각 큐 앞에 있도록 로드되고, L1 명령어는 이들이 각 큐에서 후속하도록 로드되고, L2 명령어는 이들에 후속하도록 하는 등으로 로드된다. 그룹 번호 4(예를 들어, 위에서부터 네 번째 그룹)는 그것이 별개의 그룹이더라도 L2에서 시작한다는 것이 주목되어야 한다. 이는 명령어 7이 명령어 4에 의존하고, 이는 명령어 1에 의존함으로써, 명령어 7에 L2 의존성을 부여할 수 있기 때문이다.

[0088] 이러한 방식으로, 도 16은 매 세 개의 의존성 명령어가 스케줄러(1-4) 중 주어진 하나의 스케줄러에서 어떻게 함께 스케줄되는지를 보여준다. 스케줄된 제2 레벨의 그룹은 제1 레벨의 그룹 뒤에 있고, 그 다음 그 그룹들은 회전된다.

[0089] 도 17은 본 발명의 일 실시예에 따른 3 슬롯 의존성 그룹 명령어의 계층적 이동 윈도우(moving window) 스케줄링을 묘사하는 흐름도를 도시한다. 이 실시예에서, 3 슬롯 의존성 그룹에 대한 계층적 스케줄링은 통합된 이동 윈도우 스케줄러를 통해 구현된다. 이동 윈도우 스케줄러는 큐에서 명령어를 처리하여 자원이 의존성 명령어 실행에 확실하게 이용가능한 방식으로 명령어를 디스패치한다. 전술한 바와 같이, L0 명령어는 제2 레벨의 스케줄러(1-4)에 의해 처리된 명령어 큐에 로드된다. 도 17에서 L0에서 Ln까지의 의존성 레벨로 도시된 바와 같이, L0 명령어는 이들이 각 큐 앞에 있도록 로드되고, L1 명령어는 이들이 각 큐에서 후속하도록 로드되고, L2 명령어는 이들에 후속하도록 하는 등으로 로드된다. 이동 윈도우는 L0 명령어들이 어떻게 각 큐에서 디스패치될 수 있는지를 예시하며, L0 명령어들은 하나의 큐에서 다른 것보다 더 많을 수 있다. 이러한 방식으로, 이동 윈도우 스케줄러는 큐가 도 17에 예시된 바와 같이 좌측에서 우측으로 흐름에 따라 명령어를 디스패치한다.

[0090] 도 18은 본 발명의 일 실시예에 따라 가변 크기의 의존성 명령어 체인(예를 들어, 가변 제한된 명령어 그룹)이 복수의 컴퓨팅 엔진에 어떻게 할당되는지를 보여준다.

[0091] 도 18에 도시된 바와 같이, 프로세서는 명령어 스케줄러 컴포넌트(10) 및 복수의 엔진(11-14)을 포함한다. 명령어 스케줄러 컴포넌트는 이들의 각 엔진에서 의존성 코드 블록(예를 들어, 가변 제한 그룹)의 실행을 지원하기 위해 코드 블록 및 승계 벡터(inheritance vectors)를 생성한다. 각 의존성 코드 블록은 동일한 논리 코어/스레드에 또는 서로 다른 논리 코어/스레드에 속할 수 있다. 명령어 스케줄러 컴포넌트는 의존성 코드 블록을 처리하여 각 승계 벡터를 생성할 것이다. 이러한 의존성 코드 블록 및 각 승계 벡터는 도시된 바와 같이 특정 엔진(11-14)에 할당된다. 글로벌 상호접속부(interconnect)(30)는 각 엔진(11-14) 전반에 걸쳐 필요한 통신을 지원한다. 명령어 의존성 그룹화가 도 14의 설명에서 전술한 바와 같은 의존성 명령어들의 가변 제한된 그룹들을 구축하는 기능성은 도 18의 실시예의 명령어 스케줄러 컴포넌트(10)에 의해 구현된다는 점이 주목되어야 한다.

[0092] 도 19는 본 발명의 일 실시예에 따라 스케줄링 큐에의 블록 할당 및 3 슬롯 의존성 그룹 명령어의 계층적 이동 윈도우 스케줄링을 묘사하는 흐름도를 도시한다. 전술한 바와 같이, 3 슬롯 의존성 그룹에 대한 계층적 스케줄링은 통합된 이동 윈도우 스케줄러를 통해 구현될 수 있다. 도 19는 의존성 그룹이 어떻게 스케줄링 큐에 로드된 블록이 되는지를 보여준다. 도 19의 실시예에서, 두 개의 독립 그룹이 각 큐에서 절반 블록으로 로드될 수 있다. 이는 도 19의 윗부분에서 도시되며, 여기서 그룹 1이 하나의 절반 블록을 형성하고 그룹 4가 제1 스케줄링 큐에 로드된 또 다른 절반 블록을 형성한다.

[0093] 전술한 바와 같이, 이동 윈도우 스케줄러는 큐에서 명령어를 처리하여 자원이 의존성 명령어 실행에 확실하게 이용가능한 방식으로 명령어를 디스패치한다. 도 19의 아래 부분은 L0 명령어가 어떻게 제2 레벨의 스케줄러에 의해 처리된 명령어 큐에 로드되는지를 보여준다.

[0094] 도 20은 본 발명의 일 실시예에 따라 의존성 코드 블록(예를 들어, 의존성 그룹 또는 의존성 체인)이 엔진(11-14)에서 어떻게 실행되는지를 보여준다. 전술한 바와 같이, 명령어 스케줄러 컴포넌트는 이들의 각 엔진에서

의존성 코드 블록(예를 들어, 가변 제한 그룹, 3 슬롯 그룹 등)의 실행을 지원하기 위해 코드 블록 및 승계 벡터를 생성한다. 도 19에서 전술한 바와 같이, 도 20은 두 개의 독립 그룹이 코드 블록으로서 각 엔진에 어떻게 로드될 수 있는지를 더 보여준다. 도 20은 이들 코드 블록이 어떻게 엔진(11-14)으로 디스패치되는지를 보여주며, 여기서 의존성 명령어는 각 엔진의 스택된(stacked)(예를 들어, 직렬 접속된) 실행 유닛에서 실행한다. 예를 들어, 제1 의존성 그룹, 또는 코드 블록에서, 도 20의 좌측 상부에서, 명령어는 엔진(11)으로 디스패치되고 여기서 이들은 L0이 L1의 상부에 스택되고 이는 L2 상에 더 스택되도록 이들의 의존성의 순서로 실행 유닛 상에 스택된다. 그렇게 할 때, L0의 결과는 L1의 실행 유닛으로 흐르고 그 다음에 L2의 실행으로 흐를 수 있다.

[0095] 이러한 방식으로, 도 20에 도시된 의존성 그룹들은 독립 그룹들의 매트릭스를 포함할 수 있고, 여기서 각 그룹은 의존성 명령어를 더 포함한다. 독립적인 그룹들의 이득은 이들을 병렬로 디스패치 및 실행할 수 있는 능력 및 엔진들 간의 상호접속부에 걸친 통신의 필요성이 최소화된다는 특성이다. 추가로, 엔진(11-14)에 도시된 실행 유닛이 CPU 또는 GPU를 포함할 수 있다는 것이 주목되어야 한다.

[0096] 본 발명의 실시예에 따르면, 명령어들이 이들의 의존성에 따라 의존성 그룹 또는 블록 또는 명령어 매트릭스로 추상화(abstract)된다는 점이 인식되어야 한다. 명령어들을 이들의 의존성에 따라 그룹화하면 더 큰 명령어 윈도우(예를 들어, 더 큰 입력 명령어 시퀀스)를 이용하여 더 간략화된 스케줄링 처리를 용이하게 한다. 전술한 바와 같은 그룹화는 명령어 변화를 제거하고 그러한 변화를 균일하게 추상화하며, 그럼으로써 간단하고, 균일하고 획일적인 스케줄링 의사 결정(decision-making)을 구현할 수 있도록 한다. 전술한 그룹화 기능성은 스케줄러의 복잡도를 증가시키지 않고 스케줄러의 처리율을 증대시킨다. 예를 들어, 네 개의 엔진에 대한 스케줄러에서, 스케줄러는 각 그룹이 세 개의 명령어를 갖는 네 개의 그룹을 디스패치할 수 있다. 그렇게 할 때, 스케줄러는 12개의 명령어를 디스패치하면서 단지 슈퍼 스케일러 복잡도(super scaler complexity)를 갖는 네 개의 레인(lanes)만을 처리한다. 또한, 각 블록은 디스페치된 명령어의 개수를 더 증가시키는 병렬의 독립 그룹을 포함할 수 있다.

[0097] 도 21은 본 발명의 일 실시예에 따라 다중코어 프로세서를 위한 글로벌 프론트 엔드 페치 & 스케줄러 및 레지스터 파일, 글로벌 상호접속부 및 프래그먼트된 메모리 서브시스템을 포함하여 복수의 엔진 및 이들의 컴포넌트의 개요도를 도시한다. 도 21에 도시된 바와 같이, 네 개의 메모리 프래그먼트(101-104)가 도시된다. 메모리 프래그멘테이션 계층(memory fragmentation hierarchy)은 각 캐시 계층(예를 들어, L1 캐시, L2 캐시, 및 로드 스토어 버퍼)에 걸쳐 동일하다. 각 L1 캐시, 각 L2 캐시 및 각 로드 스토어 버퍼 사이에서 메모리 글로벌 상호접속부(110a)를 통해 데이터가 교환될 수 있다.

[0098] 메모리 글로벌 상호접속부는 복수의 코어(예를 들어, 어드레스 연산 및 실행 유닛(121-124))가 프래그먼트된 캐시 계층(예를 들어, L1 캐시, 로드 스토어 버퍼 및 L2 캐시)에서 어느 지점에서든 저장될 수 있는 데이터를 액세스하게 하는 라우팅 매트릭스를 포함한다. 도 21은 또한 각 프래그먼트(101-104)를 메모리 글로벌 상호접속부(110a)를 통해 어드레스 연산 및 실행 유닛(121-124)에 의해 액세스할 수 있는 방식을 도시한다.

[0099] 실행 글로벌 상호접속부(110b)는 유사하게 복수의 코어(예를 들어, 어드레스 연산 및 실행 유닛(121-124))가 세그먼트된 레지스터 파일들 중 어떤 것에라도 저장될 수 있는 데이터를 액세스하게 하는 라우팅 매트릭스를 포함한다. 따라서, 코어는 메모리 글로벌 상호접속부(110a) 또는 실행 글로벌 상호접속부(110b)를 통해 프래그먼트들 중 어떤 것에라도 저장된 데이터 및 세그먼트들 중 어떤 것에라도 저장된 데이터에 액세스할 수 있다.

[0100] 도 21은 머신 전체도를 갖고 레지스터 파일 세그먼트 및 프래그먼트된 메모리 서브시스템의 이용을 관리하는 글로벌 프론트 엔드 페치 & 스케줄러를 더 도시한다. 어드레스 생성은 프래그먼트 정의에 대한 기준을 포함한다. 글로벌 프론트 엔드 페치 & 스케줄러는 명령어 시퀀스를 각 세그먼트에 할당함으로써 작동한다.

[0101] 도 22는 본 발명의 일 실시예에 따른 복수의 세그먼트, 복수의 세그먼트된 공통 분할 스케줄러 및 상호접속부 및 세그먼트 포트를 도시한다. 도 22에 도시된 바와 같이, 각 세그먼트는 공통 분할 스케줄러를 포함하여 도시된다. 공통 분할 스케줄러는 그의 각 세그먼트 내에서 명령어를 스케줄링함으로써 작동한다. 이러한 명령어는 다시 글로벌 프론트 엔드 페치 및 스케줄러로부터 수신되었다. 이 실시예에서, 공통 분할 스케줄러는 글로벌 프론트 엔드 페치 및 스케줄러와 협력하여 작동하도록 구성된다. 세그먼트는 또한 피연산자(operand)/결과 버퍼, 스레드된 레지스터 파일, 및 공통 분할 또는 스케줄러에의 판독/기록 액세스를 제공하는 4개의 판독 기록 포트를 포함하여 도시된다.

[0102] 일 실시예에서, 상호접속부를 이용하기 위해 비중앙집중형 액세스 처리가 구현되고 로컬 상호접속부는 각 경쟁 자원, 이 경우, 각 세그먼트의 포트에의 예약 가산기(reservation adder) 및 임계 리미터(threshold limiter)

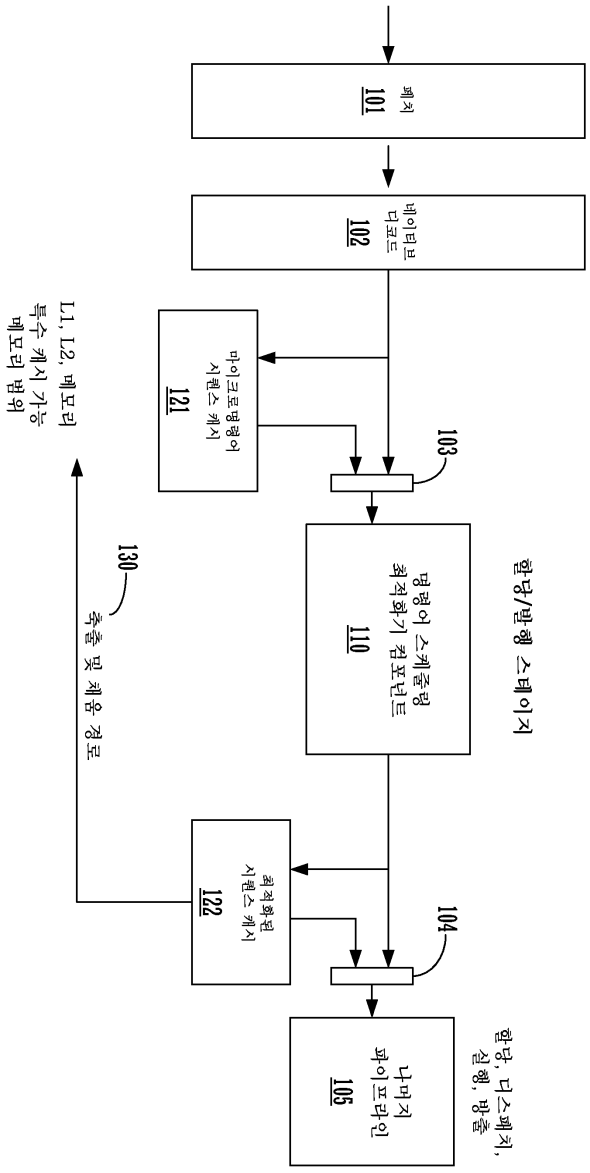
제어 액세스를 이용한다. 이러한 실시예에서, 자원에 액세스하기 위해, 코어는 필요한 버스를 예약하고 필요한 포트를 예약할 필요가 있다.

[0103] 도 23은 본 발명의 일 실시예에 따른 예시적인 마이크로프로세서 파이프라인(2300)의 도면을 도시한다. 마이크로프로세서 파이프라인(2300)은 전술한 바와 같이 실행을 포함하는 명령어를 식별 및 추출하는 처리의 기능성을 구현하는 페치 모듈(2301)을 포함한다. 도 23의 실시예에서, 페치 모듈 다음에는 디코드 모듈(2302), 할당 모듈(2303), 디스패치 모듈(2304), 실행 모듈(2305) 및 방출 모듈(2306)이 온다. 마이크로프로세서 파이프라인(2300)은 전술한 본 발명의 실시예의 기능성을 구현하는 파이프라인의 일례에 불과함이 주목되어야 한다. 이 기술분야의 통상의 기술자는 전술한 디코드 모듈의 기능성을 포함하는 다른 마이크로프로세서 파이프라인도 구현될 수 있음을 인식할 것이다.

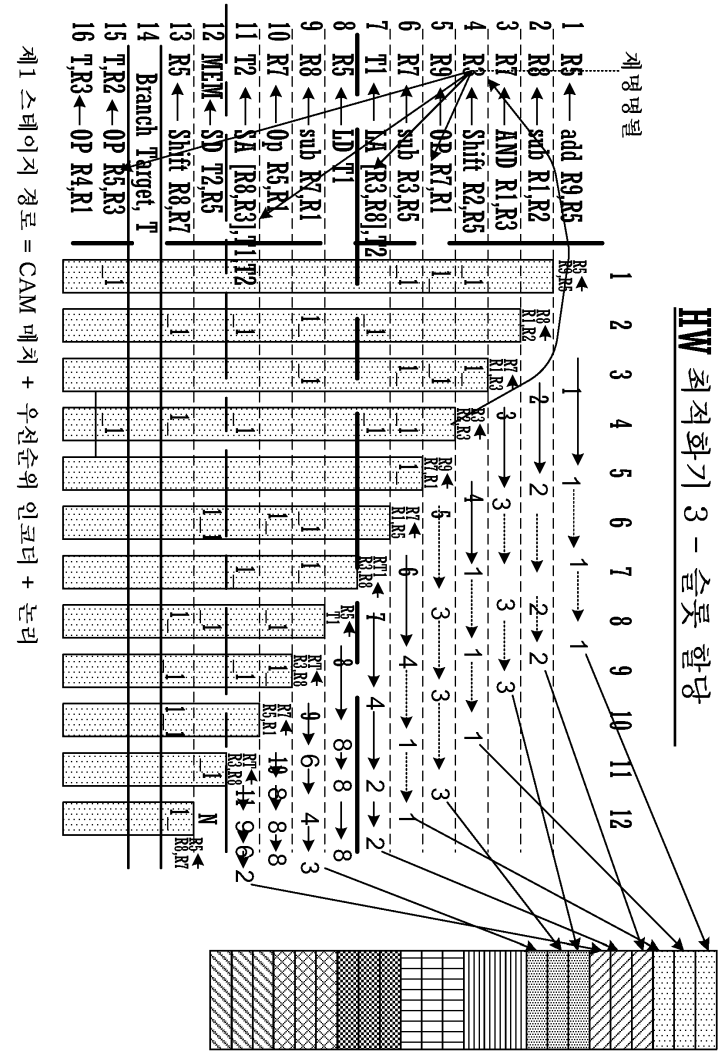
[0104] 설명 목적상, 전술한 설명은 본 발명을 제한하거나 완전하게 하는 것으로 의도되지 않는 특정 실시예를 언급한다. 전술한 교시와 일치하는 많은 수정 및 변경도 가능하다. 이 기술분야의 통상의 기술자가 본 발명 및 이들의 특정 용도에 적합할 수 있는 것으로서 다양한 수정을 갖는 그의 다양한 실시예를 잘 활용할 수 있도록 본 발명의 원리 및 그의 실시 응용을 잘 설명하도록 실시예가 선별되어 설명되었다.

도면

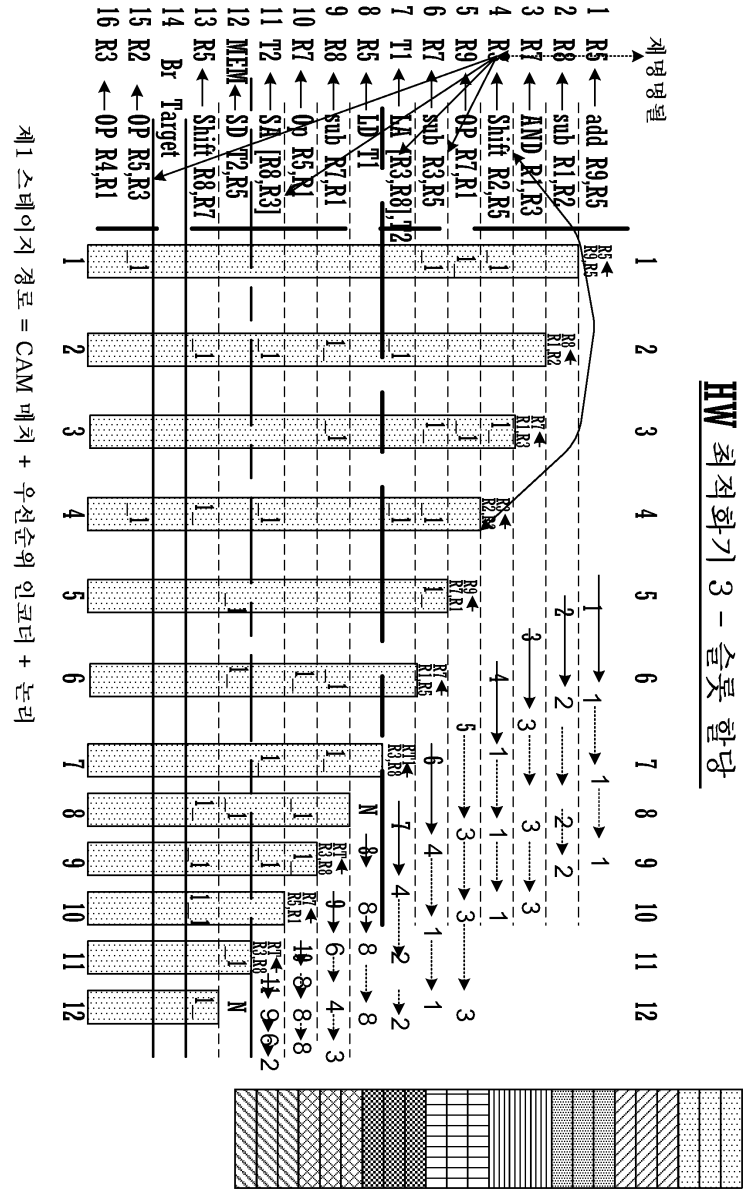
도면1



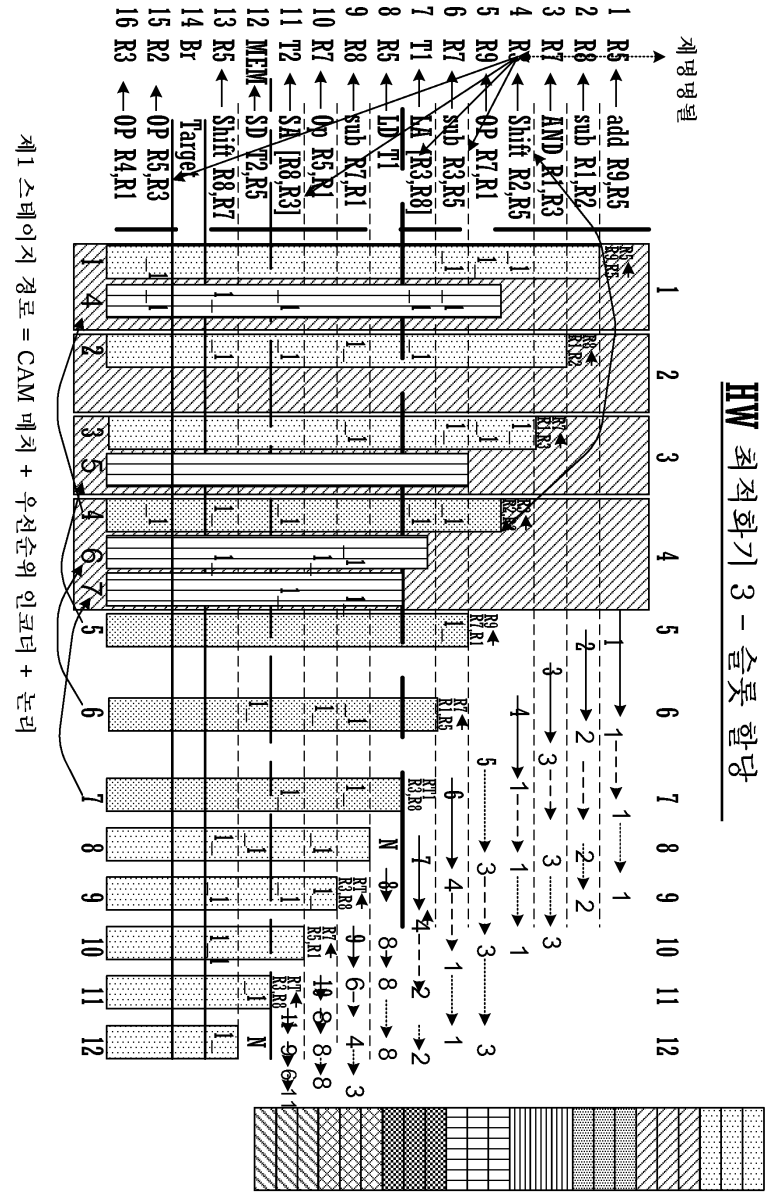
도면2



도면3



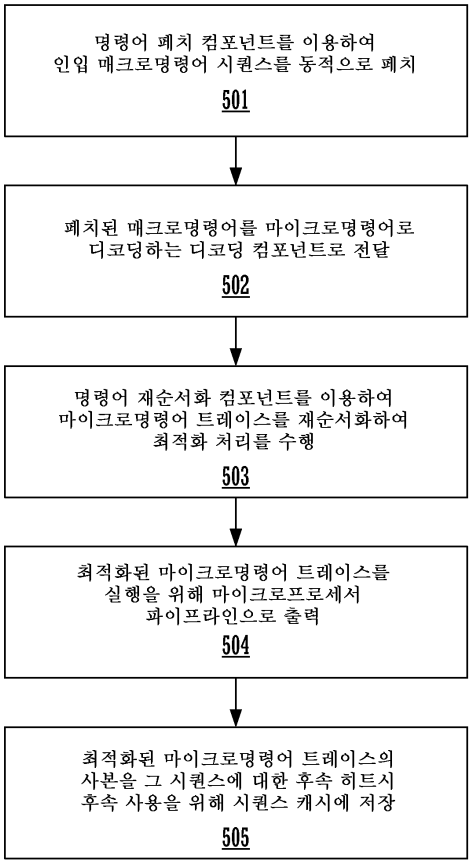
도면4



도면5

500

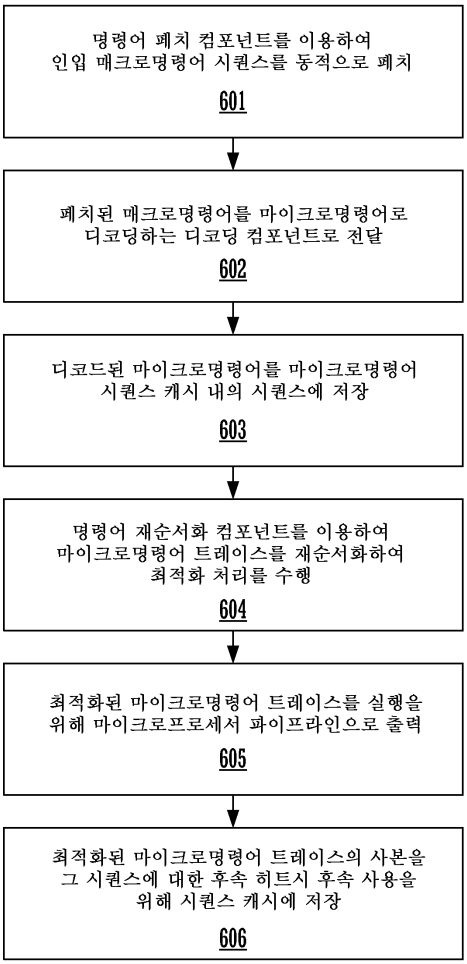
하드웨어 최적화 흐름도



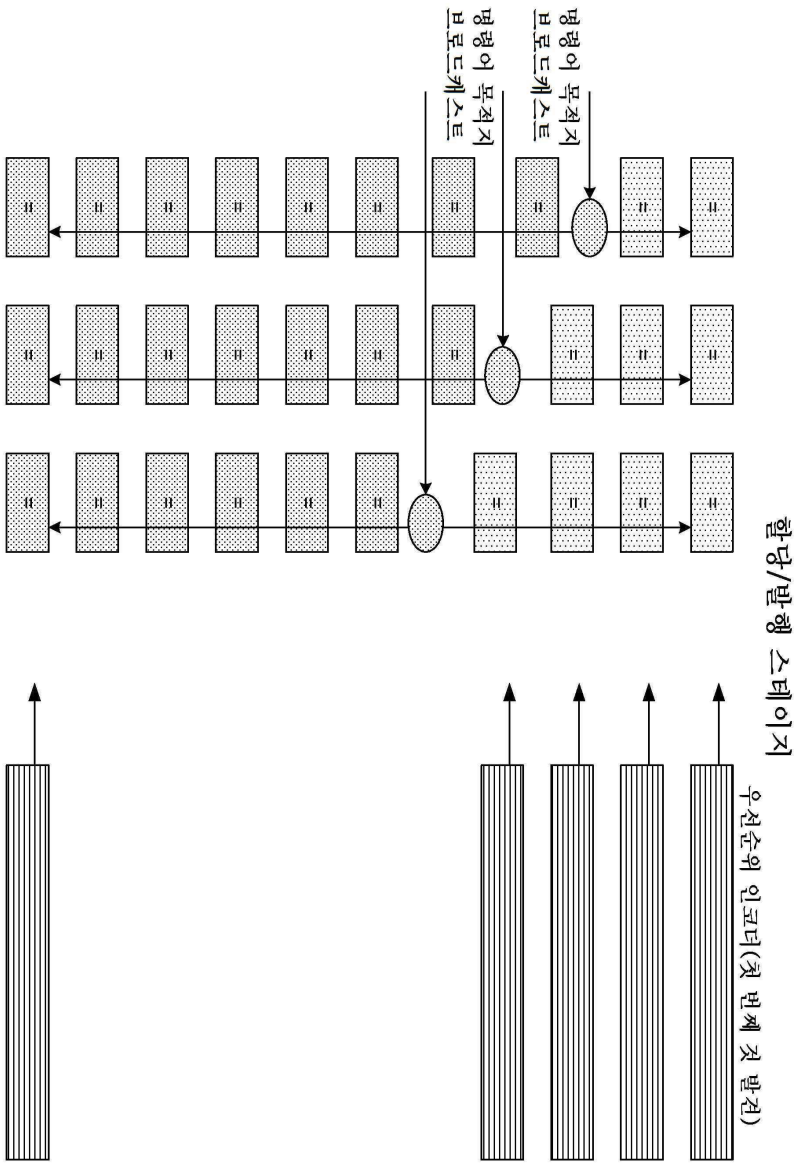
도면6

600

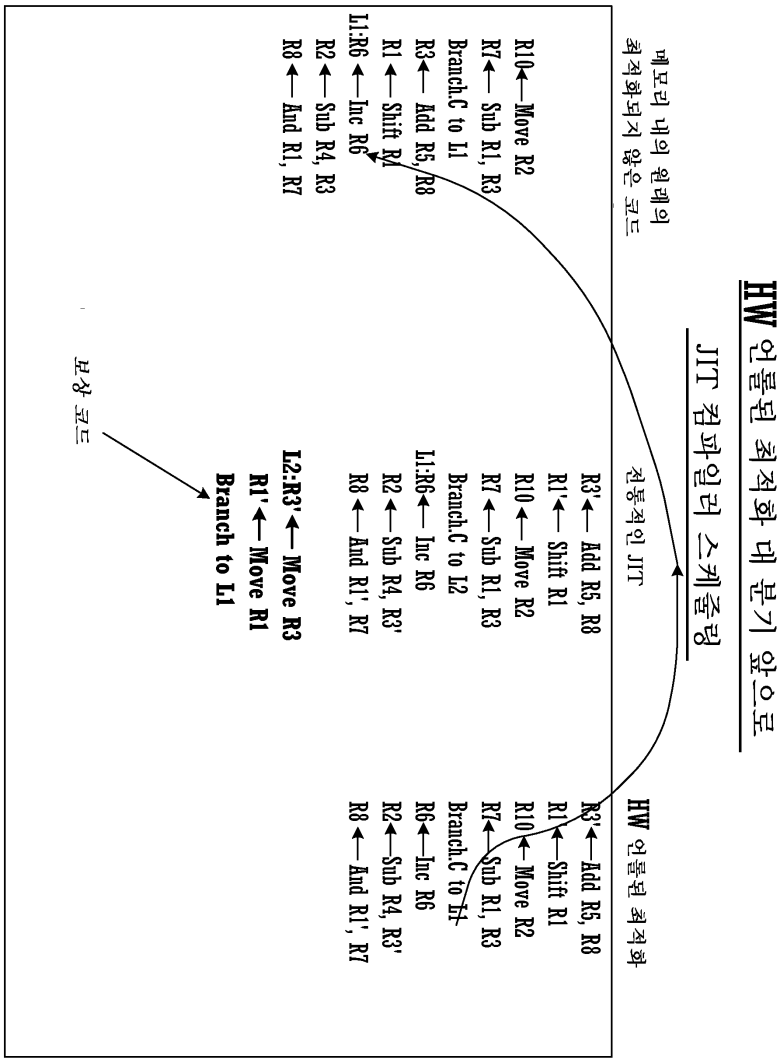
하드웨어 최적화 흐름도



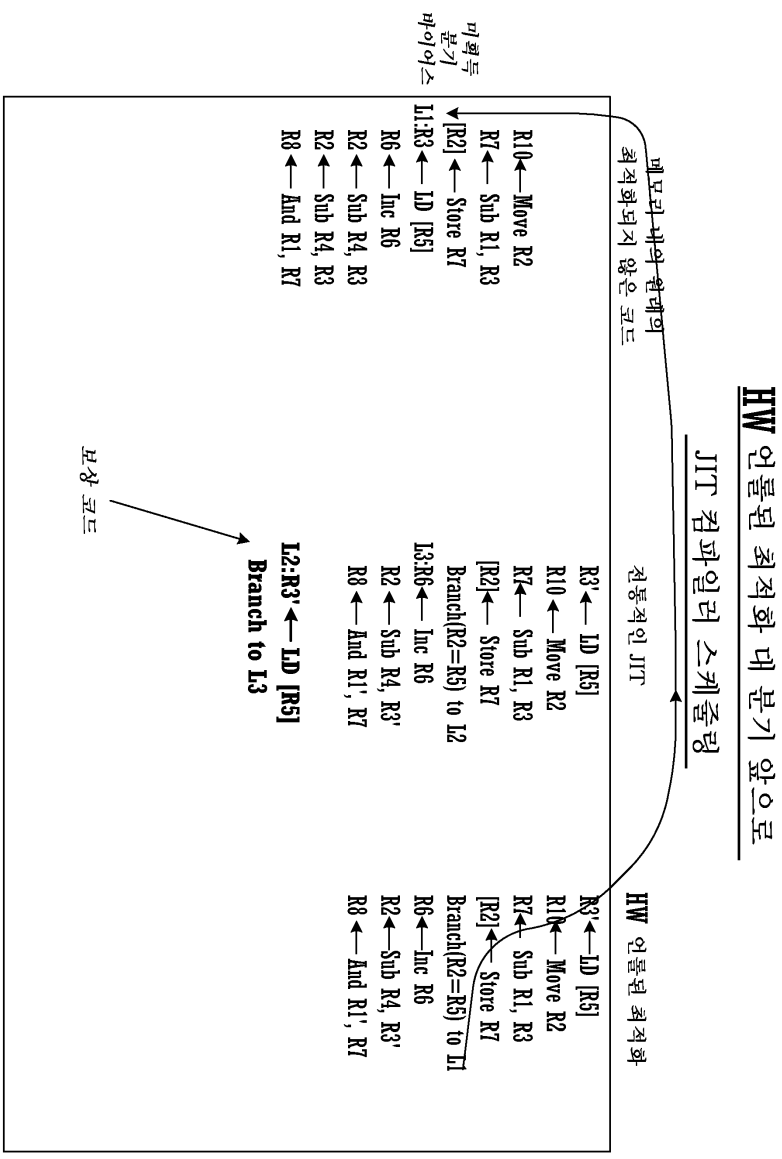
도면7



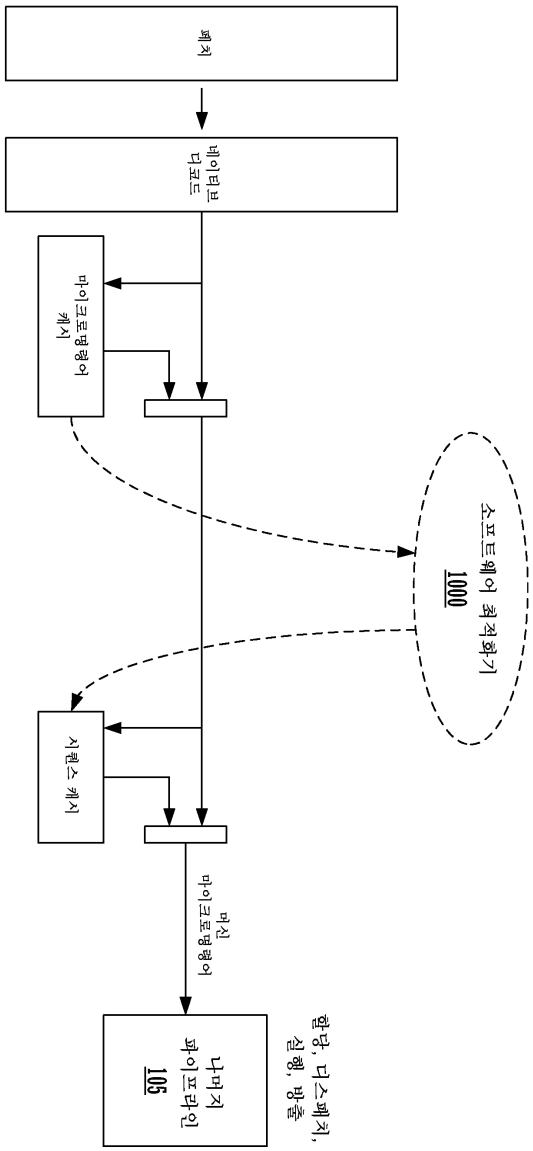
도면8


HW 언롤된 최적화 대 분기 앞으로
JIT 컴파일러 스케줄링
메모리 내의 원래의
최적화되지 않은 코드
전통적인 JIT
HW 언롤된 최적화
미확득
분기
바이어스
보상 코드
R10←Move R2
R7←Sub R1, R3
Branch.C to L1
R3←Add R5,R8
R1←Shift R1
L1:R6←Inc R6
R2←Sub R4, R3
R8←And R1, R7
R3'←Add R5, R8
R1'←Shift R1
R10←Move R2
R7←Sub R1, R3
Branch.C to L2
L1:R6←Inc R6
R2←Sub R4, R3'
R8←And R1', R7
L2:R3'←Move R3
R1'←Move R1
Branch to L1
R3'←Add R5, R8
R1'←Shift R1
R10←Move R2
R7←Sub R1, R3
Branch.C to L1
R6←Inc R6
R2←Sub R4, R3'
R8←And R1', R7

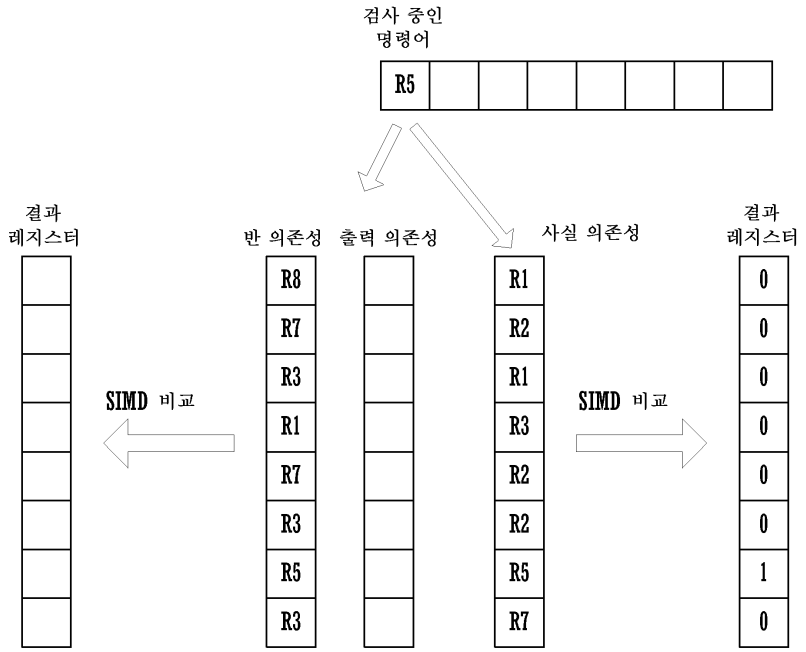
도면9


HW 언롤된 최적화 대 분기 앞으로
JIT 컴파일러 스케줄링
메모리 내의 원래의
최적화되지 않은 코드
전통적인 JIT
HW 언롤된 최적화
미혹등
분기
바이어스
R10←Move R2
R7←Sub R1, R3
[R2]←Store R7
L1:R3←LD [R5]
R6←Inc R6
R2←Sub R4, R3
R2←Sub R4, R3
R8←And R1, R7
R3'←LD [R5]
R10←Move R2
R7←Sub R1, R3
[R2]←Store R7
Branch(R2=R5) to L2
L3:R6←Inc R6
R2←Sub R4, R3'
R8←And R1', R7
L2:R3'←LD [R5]
Branch to L3
R3'←LD [R5]
R10←Move R2
R7←Sub R1, R3
[R2]←Store R7
Branch(R2=R5) to L1
R6←Inc R6
R2←Sub R4, R3'
R8←And R1', R7
보상 코드

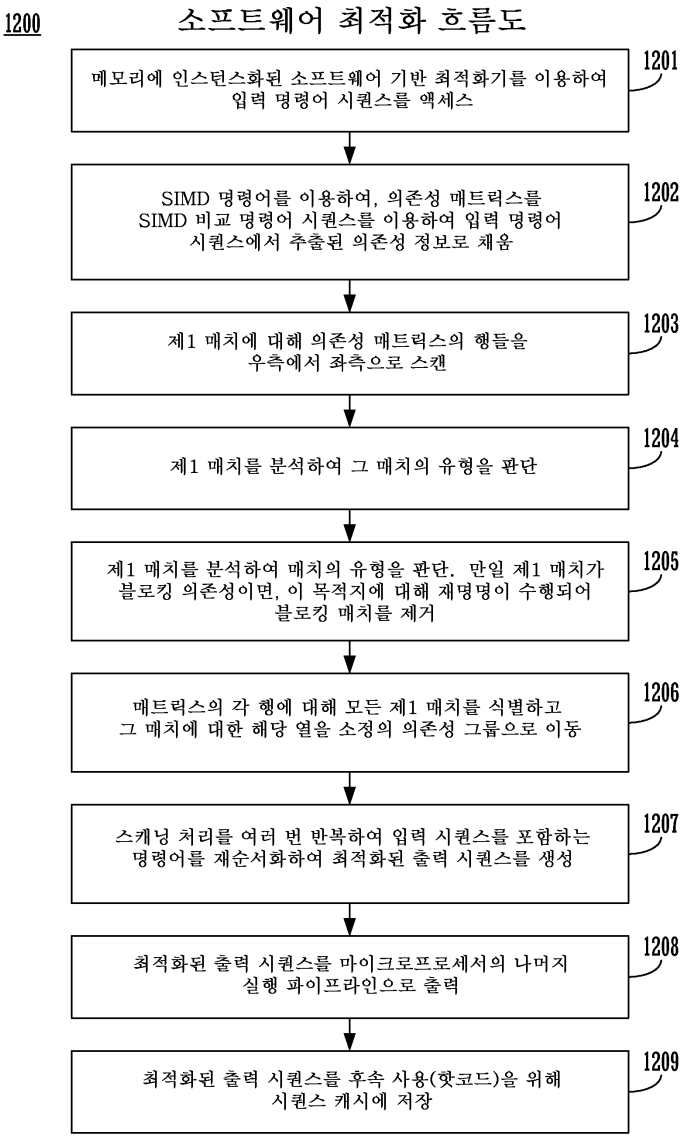
도면10


패치
네이티브 디코드
마이크로명령어 캐시
소프트웨어 최적화기
1000
시퀀스 캐시
머신 마이크로명령어
나머지 파이프라인
105
할당, 디스패치, 실행, 방출

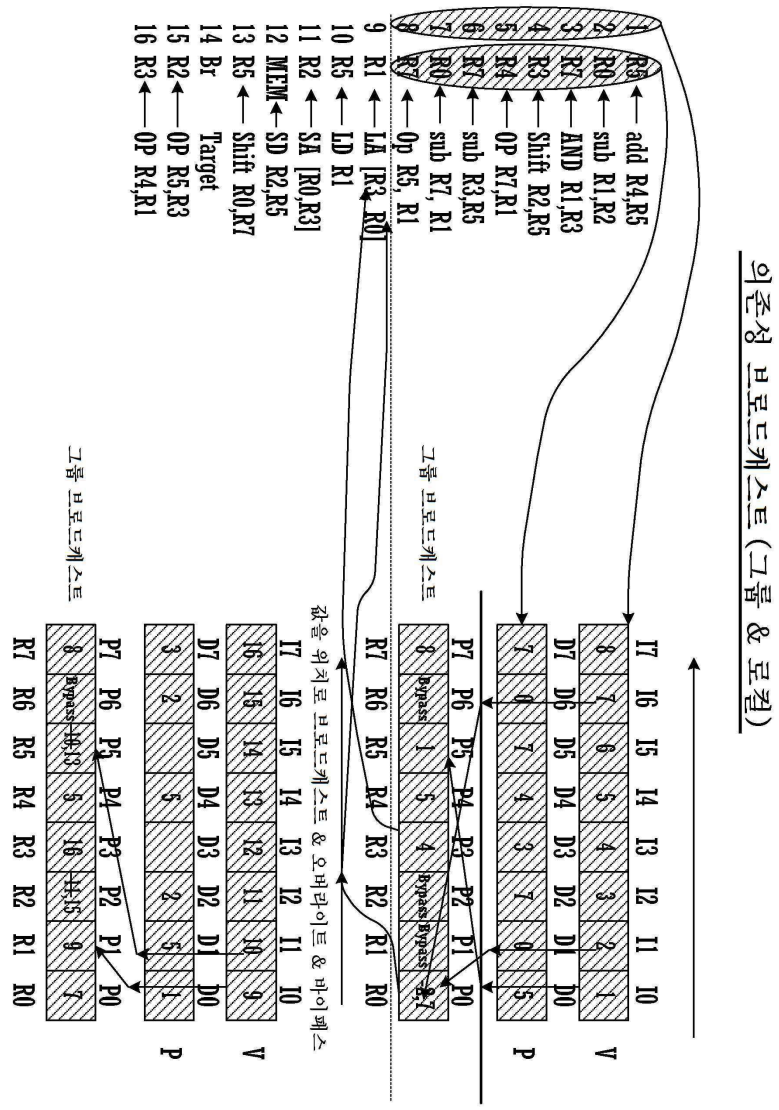
도면11


검사 중인
명령어
R5
결과
레지스터
반 의존성
R8
R7
R3
R1
R7
R3
R5
R3
출력 의존성
사실 의존성
R1
R2
R1
R3
R2
R2
R5
R7
결과
레지스터
0
0
0
0
0
0
1
0
SIMD 비교
SIMD 비교

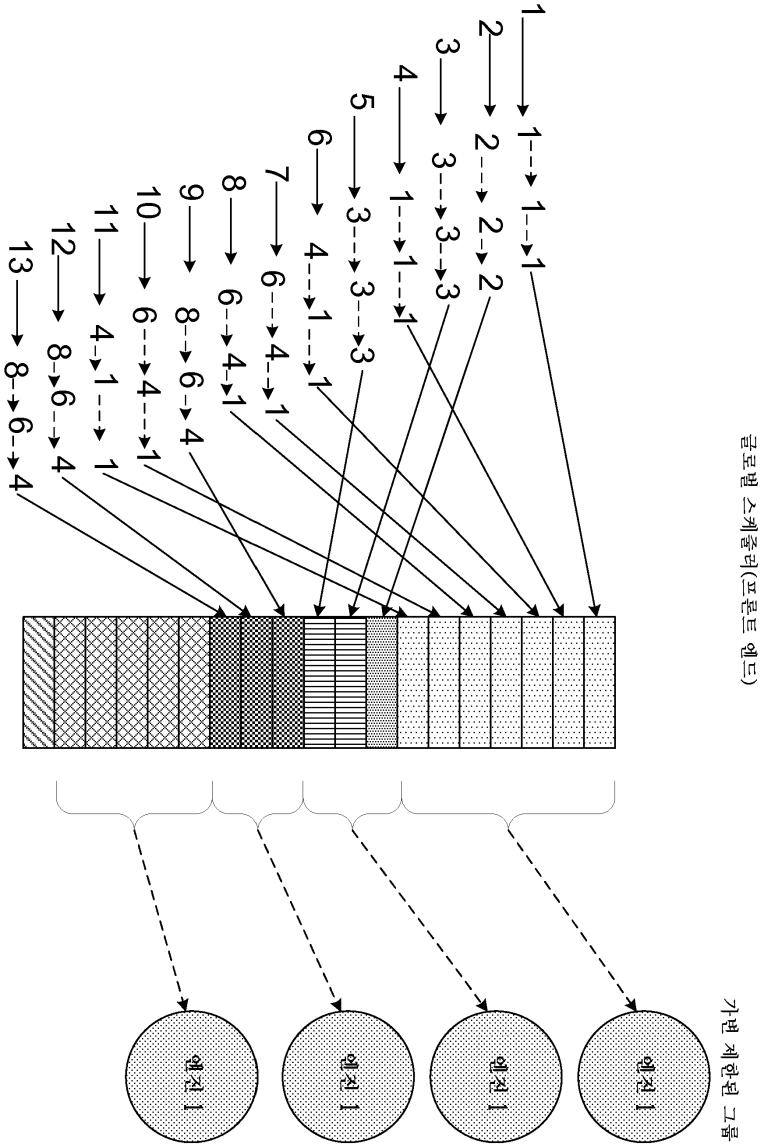
도면12


1200
소프트웨어 최적화 흐름도
메모리에 인스턴스화된 소프트웨어 기반 최적화기를 이용하여 입력 명령어 시퀀스를 액세스
1201
SIMD 명령어를 이용하여, 의존성 매트릭스를 SIMD 비교 명령어 시퀀스를 이용하여 입력 명령어 시퀀스에서 추출된 의존성 정보로 채움
1202
제1 매치에 대해 의존성 매트릭스의 행들을 우측에서 좌측으로 스캔
1203
제1 매치를 분석하여 그 매치의 유형을 판단
1204
제1 매치를 분석하여 매치의 유형을 판단. 만일 제1 매치가 블로킹 의존성이면, 이 목적지에 대해 재명명이 수행되어 블로킹 매치를 제거
1205
매트릭스의 각 행에 대해 모든 제1 매치를 식별하고 그 매치에 대한 해당 열을 소정의 의존성 그룹으로 이동
1206
스캐닝 처리를 여러 번 반복하여 입력 시퀀스를 포함하는 명령어를 재순서화하여 최적화된 출력 시퀀스를 생성
1207
최적화된 출력 시퀀스를 마이크로프로세서의 나머지 실행 파이프라인으로 출력
1208
최적화된 출력 시퀀스를 후속 사용(핫코드)을 위해 시퀀스 캐시에 저장
1209

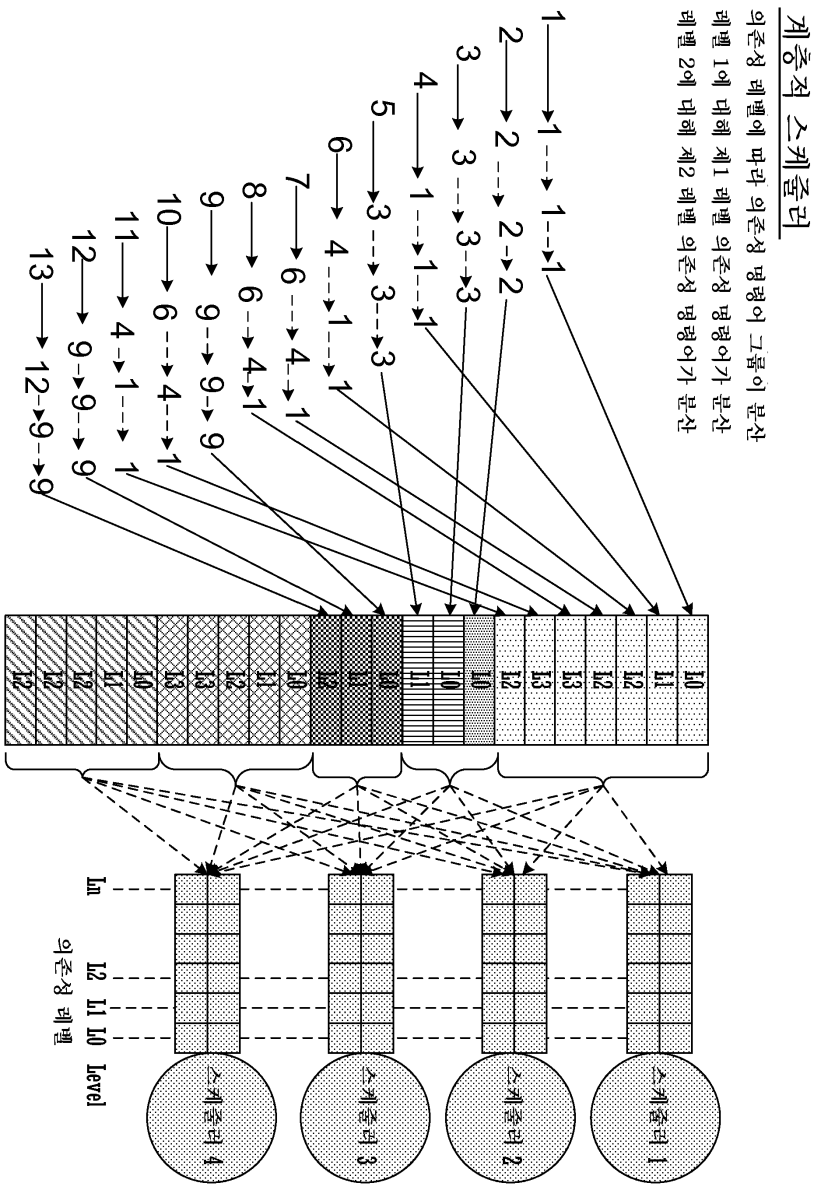
도면13


의존성 브로드캐스트 (그룹 & 로컬)
그룹 브로드캐스트
그룹 브로드캐스트
값을 위치로 브로드캐스트 & 오버라이트 & 바이패스
add R4,R5
sub R1,R2
AND R1,R3
Shift R2,R5
OP R7,R1
sub R3,R5
sub R7, R1
Op R5, R1
LA [R3,R0]
LD R1
SA [R0,R3]
SD R2,R5
Shift R0,R7
Target
OP R5,R3
OP R4,R1
Bypass
V
P

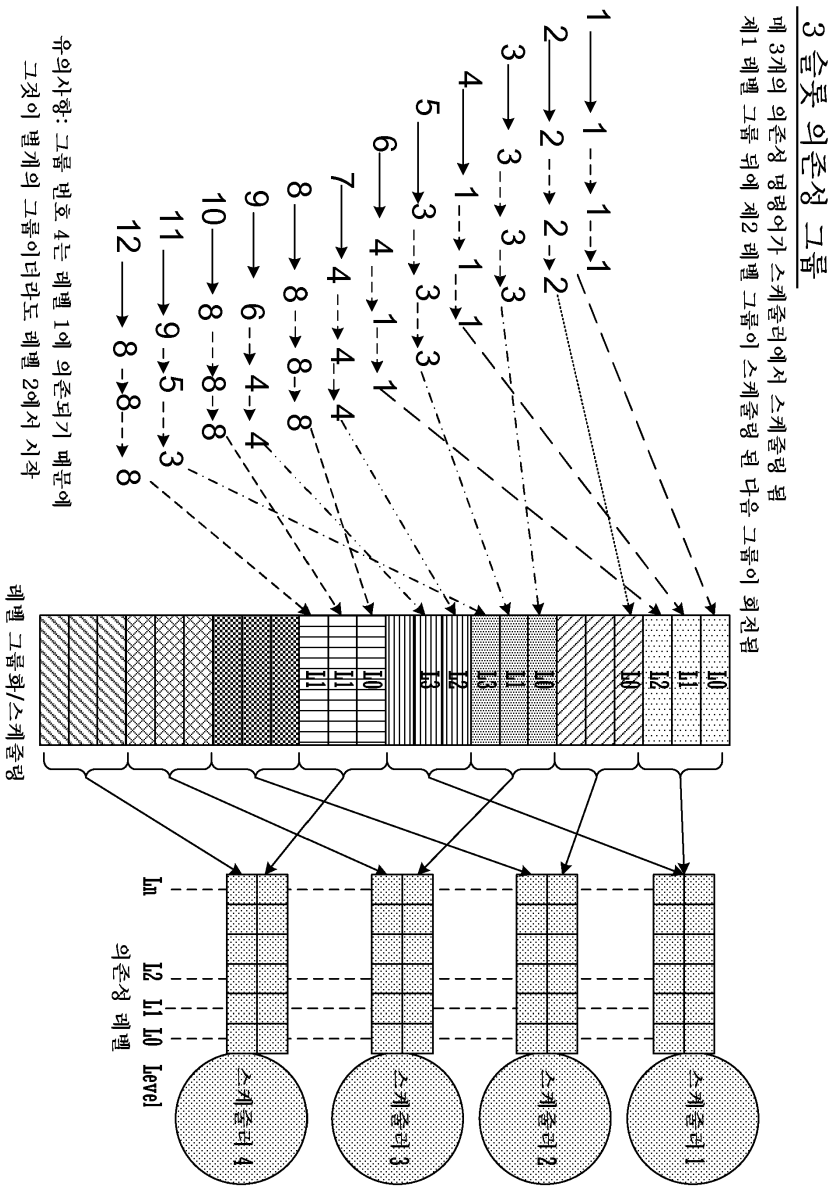
도면14



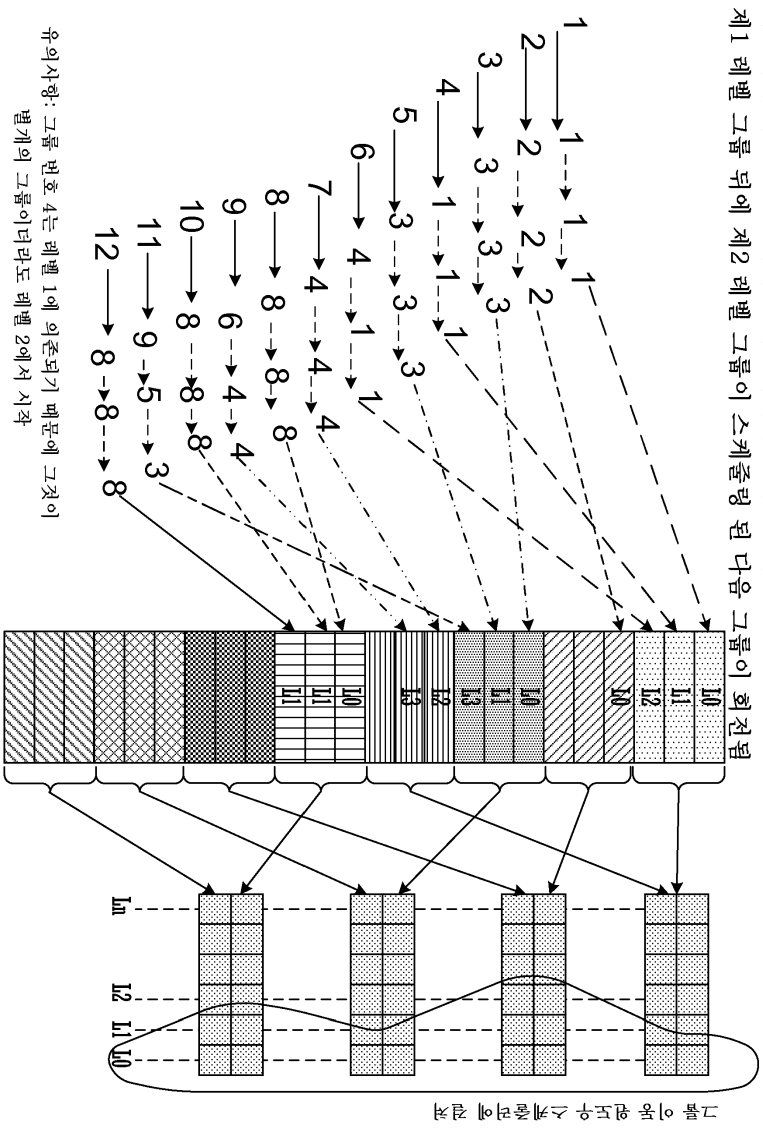
도면15


계층적 스케줄러
의존성 레벨에 따라 의존성 명령어 그룹이 분산
레벨 1에 대해 제1 레벨 의존성 명령어가 분산
레벨 2에 대해 제2 레벨 의존성 명령어가 분산
스케줄러 1
스케줄러 2
스케줄러 3
스케줄러 4
의존성 레벨
Level
L0
L1
L2
Ln

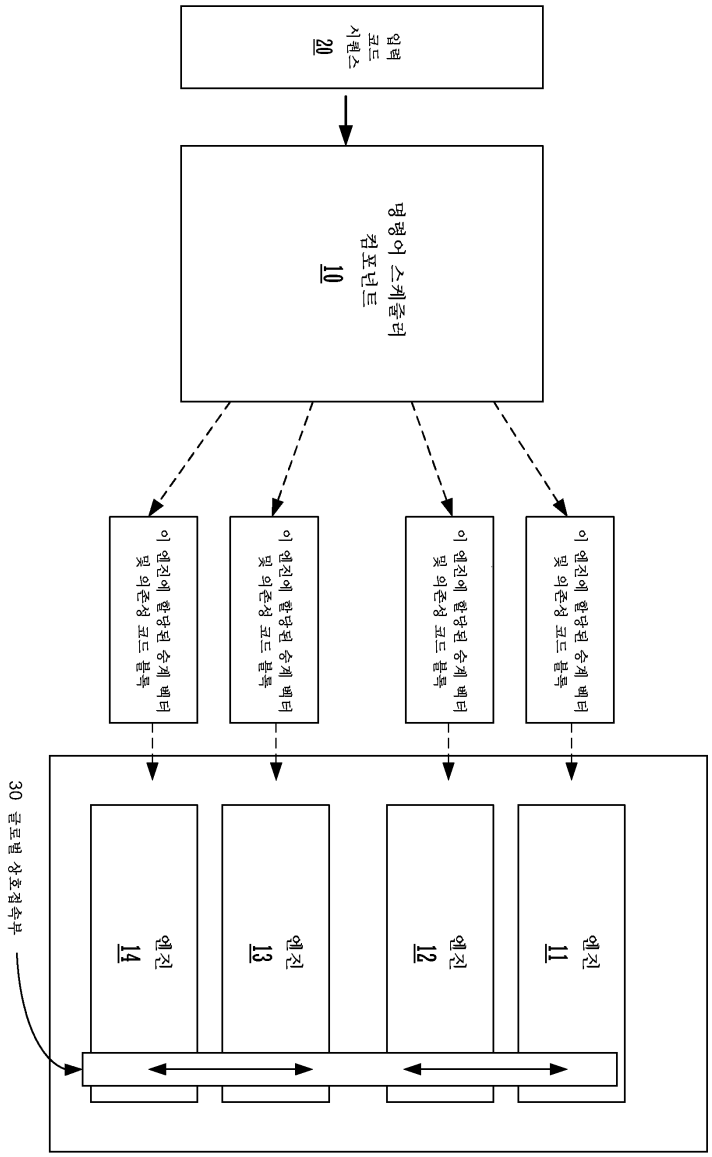
도면16


3 슬롯 의존성 그룹
매 3개의 의존성 명령어가 스케줄러에서 스케줄링 됨
제1 레벨 그룹 뒤에 제2 레벨 그룹이 스케줄링 된 다음 그룹이 회전 됨
1 2 3 4 5 6 7 8 9 10 11 12
1 2 3 1 3 4 4 8 6 8 9 8
레벨 그룹화/스케줄링
스케줄러 1
스케줄러 2
스케줄러 3
스케줄러 4
의존성 레벨
Level
I0 I1 I2 In
유의사항: 그룹 번호 4는 레벨 1에 의존되기 때문에
그것이 별개의 그룹이더라도 레벨 2에서 시작

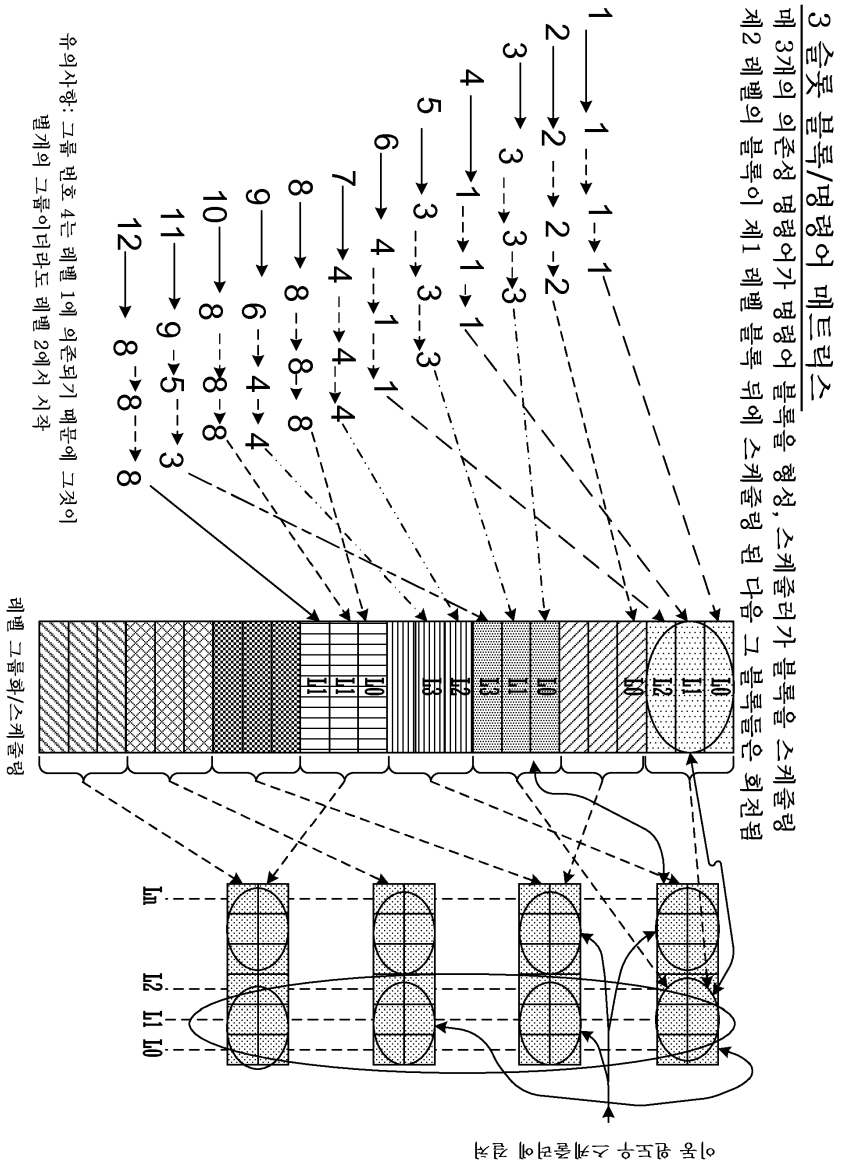
도면17


3 슬롯 의존성 그룹
매 3개의 의존성 명령어가 스케줄러에서 스케줄링 됨
제1 레벨 그룹 뒤에 제2 레벨 그룹이 스케줄링 된 다음 그룹이 회전됨
그룹 이동 윈도우 스케줄러에 걸쳐
유의사항: 그룹 번호 4는 레벨 1에 의존되기 때문에 그것이
별개의 그룹이더라도 레벨 2에서 시작

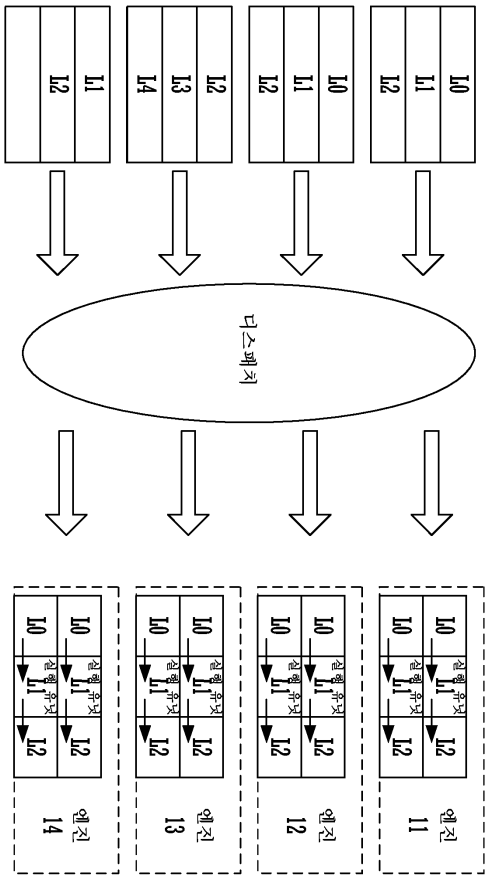
도면18


입력 코드 시퀀스 20
명령어 스케줄러 컴포넌트 10
이 엔진에 할당된 슈퍼 블록 및 의존성 코드 블록
이 엔진에 할당된 슈퍼 블록 및 의존성 코드 블록
이 엔진에 할당된 슈퍼 블록 및 의존성 코드 블록
이 엔진에 할당된 슈퍼 블록 및 의존성 코드 블록
엔진 14
엔진 13
엔진 12
엔진 11
30 글로벌 상호접속부

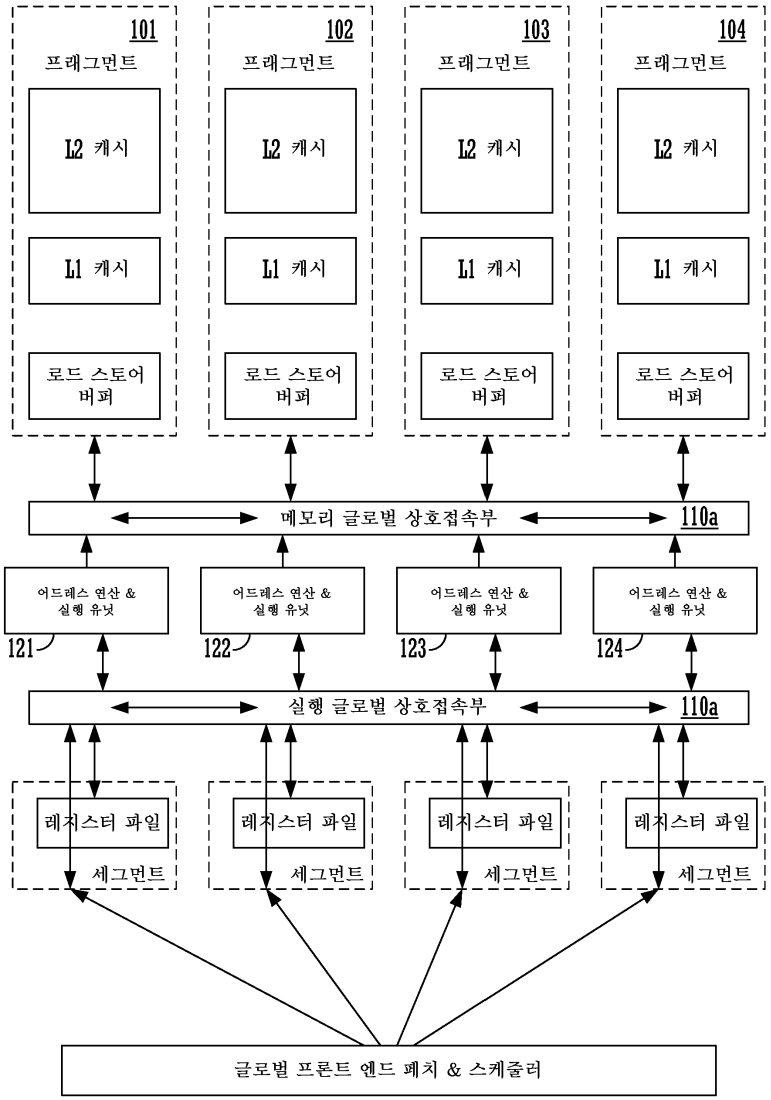
도면19


3 슬롯 블록/명령어 매트릭스
매 3개의 의존성 명령어가 명령어 블록을 형성, 스케줄러가 블록을 스케줄링
제2 레벨의 블록이 제1 레벨 블록 뒤에 스케줄링 된 다음 그 블록들은 회전됨
유의사항: 그룹 번호 4는 레벨 1에 의존되기 때문에 그것이
별개의 그룹이더라도 레벨 2에서 시작
레벨 그룹화/스케줄링
이동 윈도우 스케줄러에 접촉
L0
L1
L2
L3
Ln

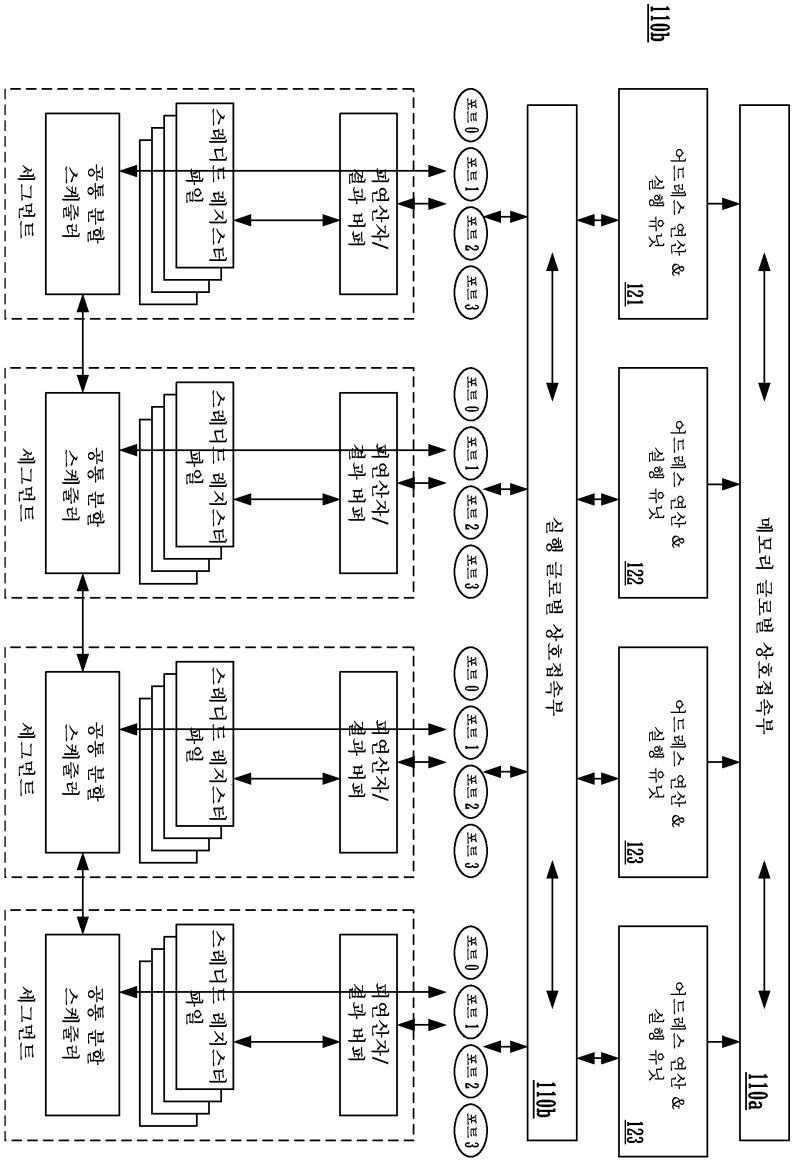
도면20



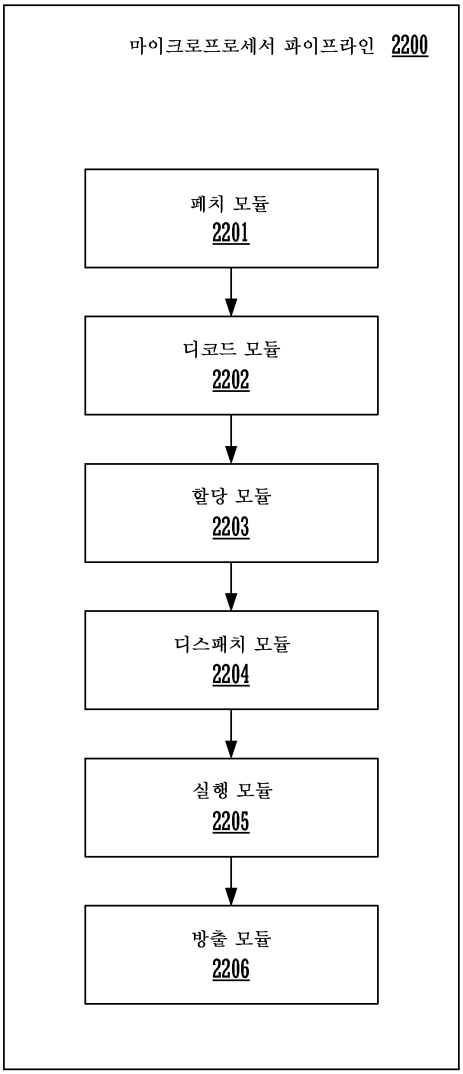
도면21


101
프래그먼트
L2 캐시
L1 캐시
로드 스토어 버퍼
102
프래그먼트
L2 캐시
L1 캐시
로드 스토어 버퍼
103
프래그먼트
L2 캐시
L1 캐시
로드 스토어 버퍼
104
프래그먼트
L2 캐시
L1 캐시
로드 스토어 버퍼
메모리 글로벌 상호접속부
110a
어드레스 연산 & 실행 유닛
121
어드레스 연산 & 실행 유닛
122
어드레스 연산 & 실행 유닛
123
어드레스 연산 & 실행 유닛
124
실행 글로벌 상호접속부
110a
레지스터 파일
세그먼트
레지스터 파일
세그먼트
레지스터 파일
세그먼트
레지스터 파일
세그먼트
글로벌 프론트 엔드 페치 & 스케줄러

도면22


110b
메모리 글로벌 상호접속부
110a
어드레스 연산 & 실행 유닛
121
어드레스 연산 & 실행 유닛
122
어드레스 연산 & 실행 유닛
123
어드레스 연산 & 실행 유닛
123
실행 글로벌 상호접속부
110b
포트 0
포트 1
포트 2
포트 3
피연산자/결과 버퍼
스레디드 레지스터 파일
공통 분할 스케줄러
세그먼트

도면23


마이크로프로세서 파이프라인 2200
페치 모듈
2201
디코드 모듈
2202
할당 모듈
2203
디스패치 모듈
2204
실행 모듈
2205
방출 모듈
2206