



(19) 대한민국특허청(KR)
(12) 공개특허공보(A)

(11) 공개번호 10-2017-0102576
(43) 공개일자 2017년09월11일

- | | |
|--|---|
| <p>(51) 국제특허분류(Int. Cl.)
G06F 9/38 (2006.01)</p> <p>(52) CPC특허분류
G06F 9/3836 (2013.01)
G06F 9/3838 (2013.01)</p> <p>(21) 출원번호 10-2017-7024365(분할)</p> <p>(22) 출원일자(국제) 2013년06월11일
심사청구일자 없음</p> <p>(62) 원출원 특허 10-2015-7000695
원출원일자(국제) 2013년06월11일
심사청구일자 2015년03월26일</p> <p>(85) 번역문제출일자 2017년08월30일</p> <p>(86) 국제출원번호 PCT/US2013/045261</p> <p>(87) 국제공개번호 WO 2013/188460
국제공개일자 2013년12월19일</p> <p>(30) 우선권주장
61/660,548 2012년06월15일 미국(US)</p> | <p>(71) 출원인
인텔 코퍼레이션
미합중국 캘리포니아 95054 산타클라라 미션 칼리지 블러바드 2200</p> <p>(72) 발명자
압달라, 모하마드
미국 95132 캘리포니아주 산 호세 선크레스트 애비뉴 3868</p> <p>(74) 대리인
양영준, 백만기</p> |
|--|---|

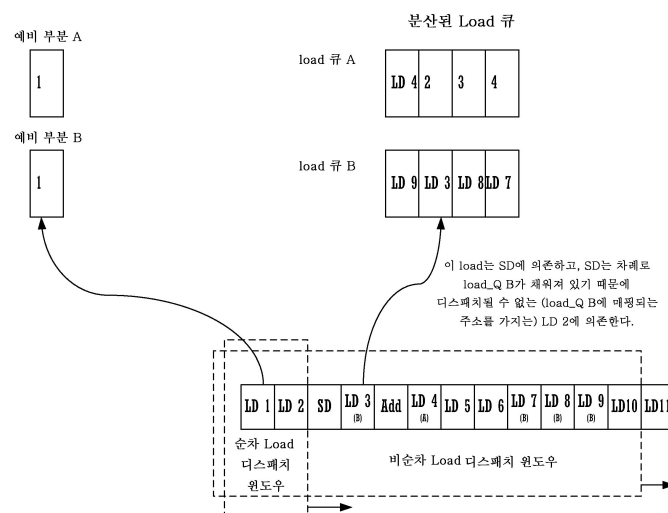
전체 청구항 수 : 총 1 항

(54) 발명의 명칭 분산된 구조를 갖는 동적 디스패치 윈도우를 가지는 가상 LOAD STORE 큐

(57) 요약

비순차 프로세서가 개시된다. 이 프로세서는 복수의 코어들 및 메모리 프래그먼트들에 걸친 load들 및 store들의 비순차 디스패치를 가능하게 하면서 단일의 프로그램 순차적 시맨틱스를 유지하는 분산된 load 큐 및 분산된 store 큐를 포함하고; 프로세서는 load/store 큐의 실제의 물리적 크기 제한을 넘어 load들 및 store들 이외의 다른 명령어들을 할당하며; 개재된 load들 또는 store들이 load store 큐에 공간들을 갖지 않더라도 다른 명령어들이 디스패치 및 실행될 수 있다.

대표도 - 도12



(52) CPC특허분류
G06F 9/3855 (2013.01)

명세서

청구범위

청구항 1

제1항에 따른 장치.

발명의 설명

기술 분야

- [0001] 본 출원은 2012년 6월 15일자로 출원된, 동시 계류 중이고 본 발명의 양수인에게 양도된 발명의 명칭이 "A VIRTUAL LOAD STORE QUEUE HAVING A DYNAMIC DISPATCH WINDOW WITH A DISTRIBUTED STRUCTURE"인 Mohammad A. Abdallah의 미국 가특허 출원 제61/660,548호(본 명세서에 그 전체가 포함됨)에 기초하여 우선권을 주장한다.
- [0002] 본 발명은 일반적으로 디지털 컴퓨터 시스템에 관한 것으로서, 보다 상세하게는, 명령어 시퀀스를 포함한 명령어들을 선택하는 시스템 및 방법에 관한 것이다.

배경 기술

- [0003] 의존적이거나 완전히 독립적인 다수의 작업들을 처리하는 데 프로세서가 필요하다. 이러한 프로세서의 내부 상태는 보통 프로그램 실행의 각각의 특징의 순간에 상이한 값들을 보유하고 있을 수 있는 레지스터들로 이루어져 있다. 프로그램 실행의 각각의 순간에, 내부 상태 이미지(internal state image)는 프로세서의 아키텍처 상태(architecture state)라고 불리운다.
- [0004] 다른 함수(예컨대, 다른 스레드, 프로세스 또는 프로그램)를 실행하기 위해 코드 실행이 전환될 때, 새로운 함수가 그의 새로운 상태를 빌드(build)하기 위해 내부 레지스터들을 이용할 수 있도록 기계/프로세서의 상태가 저장되어야만 한다. 새로운 함수가 종료되면, 그의 상태가 폐기될 수 있고 이전의 컨텍스트의 상태가 복원될 것이고 실행이 재개된다. 이러한 전환 프로세스(switch process)는 컨텍스트 전환(context switch)이라고 불리우고, 특히 많은 수(예컨대, 64개, 128개, 256개)의 레지스터들 및/또는 비순차 실행(out of order execution)을 이용하는 최근의 아키텍처들에서 보통 수십 또는 수백개의 사이클들을 포함한다.
- [0005] 스레드 인식 하드웨어 아키텍처(thread-aware hardware architecture)에서, 하드웨어가 제한된 수의 하드웨어 지원 스레드들에 대해 다수의 컨텍스트 상태들을 지원하는 것이 보통이다. 이 경우에, 하드웨어는 각각의 지원된 스레드에 대한 모든 아키텍처 상태 요소들을 복제한다. 이것은 새로운 스레드를 실행할 때 컨텍스트 전환을 필요 없게 만든다. 그렇지만, 이것은 여전히 다수의 단점들(즉, 하드웨어로 지원되는 각각의 부가적인 스레드에 대한 모든 아키텍처 상태 요소들(즉, 레지스터들)을 복제하는 것의 면적, 능력 및 복잡도)을 가진다. 그에 부가하여, 소프트웨어 스레드들의 수가 명시적으로 지원되는 하드웨어 스레드들의 수를 초과하는 경우, 컨텍스트 전환이 여전히 수행되어야만 한다.
- [0006] 이것이 흔하게 되는데, 그 이유는 미세 입도 기반의(on a fine granularity basis) 병렬 처리(parallelism)가 필요하여 많은 수의 스레드들을 필요로 하기 때문이다. 중복 컨텍스트 상태 하드웨어 저장(duplicate context-state hardware storage)을 갖는 하드웨어 스레드 인식 아키텍처들은 비-스레드 방식 소프트웨어 코드(non-threaded software code)에는 도움을 주지 못하고 스레드 방식(threaded)으로 되어 있는 소프트웨어에 대한 컨텍스트 전환들의 수를 감소시킬 뿐이다. 그렇지만, 그 스레드들은 보통 대단위 병렬 처리(coarse grain parallelism)를 위해 구성되어 있으며, 그 결과 개시 및 동기화를 위한 소프트웨어 오버헤드가 과중하게 되고, 효율적인 스레딩 개시/자동 발생 없이, 함수 호출 및 루프 병렬 실행과 같은 소단위 병렬 처리(fine grain parallelism)를 남겨 두게 된다. 이러한 기술된 오버헤드들은 비명시적으로/용이하게 병렬화되는/스레딩되는 소프트웨어 코드들에 대해 최신의 컴파일러 또는 사용자 병렬화 기법들을 사용한 이러한 코드들의 자동 병렬화의 어려움을 수반한다.

발명의 내용

- [0007] 하나의 실시예에서, 본 발명은 비순차 프로세서로서 구현된다. 이 프로세서는 복수의 코어들 및 메모리 프래그

먼트(memory fragment)들에 걸친 load들 및 store들의 비순차 디스패치를 가능하게 하면서 단일의 프로그램 순차적 시맨틱스를 유지하는 분산된 load 큐 및 분산된 store 큐를 포함하고; 프로세서는 load/store 큐의 실제의 물리적 크기 제한을 넘어 load들 및 store들 이외의 다른 명령어들을 할당하며; 개재된(intervening) load들 또는 store들이 load store 큐에 공간들을 갖지 않더라도 다른 명령어들이 디스패치 및 실행될 수 있다.

[0008] 이상은 발명의 내용이고, 따라서 필연적으로 상세의 간략화, 일반화 및 생략을 포함하고; 그 결과 통상의 기술자라면 이 발명의 내용이 예시적인 것에 불과하고 결코 제한하기 위한 것이 아님을 잘 알 것이다. 청구항들에 의해서만 한정되는 본 발명의 다른 측면들, 발명 특징들 및 장점들이 이하에 기재되는 비제한적인 상세한 설명에서 명백하게 될 것이다.

도면의 간단한 설명

[0009] 유사한 참조 번호들이 유사한 요소들을 가리키고 있는 첨부 도면의 도면들에, 본 발명이 제한이 아닌 예로서 도시되어 있다.

도 1은 본 발명의 하나의 실시예에 따른, load 큐(load queue) 및 store 큐(store queue)를 도시한 도면.

도 2는 본 발명의 하나의 실시예에 따른, load 및 store 명령어 분할(instruction splitting)의 제1 다이어그램을 도시한 도면.

도 3은 본 발명의 하나의 실시예에 따른, load 및 store 명령어 분할의 제2 다이어그램을 도시한 도면.

도 4는 본 발명의 하나의 실시예에 따른, load store 재정렬 및 최적화로부터 생기는 투기적 포워딩 예측 착오/오류로부터의 복원을 구현하는 규칙들이 나타내어져 있는 프로세스의 단계들의 플로우차트를 도시한 도면.

도 5는 본 발명의 하나의 실시예에 따른, 프로세스(300)의 규칙들이 프로세서의 load 큐 및 store 큐 자원들로 구현되는 방식을 나타낸 다이어그램을 도시한 도면.

도 6은 본 발명의 하나의 실시예에 따른, 프로세스(300)의 규칙들이 프로세서의 load 큐 및 store 큐 자원들로 구현되는 방식을 나타낸 다른 다이어그램을 도시한 도면.

도 7은 본 발명의 하나의 실시예에 따른, 프로세스(300)의 규칙들이 프로세서의 load 큐 및 store 큐 자원들로 구현되는 방식을 나타낸 다른 다이어그램을 도시한 도면.

도 8은 본 발명의 하나의 실시예에 따른, store가 load 이후에 디스패치되는 디스패치 기능의 개요의 프로세스의 플로우차트를 도시한 도면.

도 9는 본 발명의 하나의 실시예에 따른, load가 store 이후에 디스패치되는 디스패치 기능의 개요의 프로세스의 플로우차트를 도시한 도면.

도 10은 본 발명의 하나의 실시예에 따른, 통합된 load 큐(unified load queue)의 다이어그램을 도시한 도면.

도 11은 본 발명의 하나의 실시예에 따른, 슬라이딩 load 디스패치 윈도우(sliding load dispatch window)를 보여주는 통합된 load 큐를 도시한 도면.

도 12는 본 발명의 하나의 실시예에 따른, 분산된 load 큐(distributed load queue)를 도시한 도면.

도 13은 본 발명의 하나의 실시예에 따른, 순차 연속성 윈도우(in order continuity window)를 가지는 분산된 load 큐를 도시한 도면.

도 14는 본 발명의 하나의 실시예에 따른, 멀티코어 프로세서에 대한 프래그먼트화된 메모리 서브시스템(fragmented memory subsystem)의 다이어그램을 도시한 도면.

도 15는 본 발명의 실시예들에 의해 load 및 store가 어떻게 처리되는지의 다이어그램을 도시한 도면.

도 16은 본 발명의 하나의 실시예에 따른, store 필터링 알고리즘의 다이어그램을 도시한 도면.

도 17은 본 발명의 하나의 실시예에 따른, 메모리로부터 순차적으로 판독하는 load를 구성하는 메모리 일관성 모델에서의 비순차 load를 갖는 세마포어(semaphore) 구현을 도시한 도면.

도 18은 본 발명의 하나의 실시예에 따른, 로크 기반 모델(lock-based model) 및 트랜잭션 기반 모델(transaction-based model) 둘 다의 사용에 의해 메모리로부터 순차적으로 판독하는 load를 구성하는 메모리 일

관성 모델로의 비순차 load를 도기한 도면.

도 19는 본 발명의 하나의 실시예에 따른, 멀티 코어 세그먼트화된 메모리 서브시스템의 복수의 코어들을 도기한 도면.

도 20은 본 발명의 하나의 실시예에 따른, store가 store 상위 서열(store seniority)에 기초하여 어느 한 스레드로부터 이용가능할 수 있는 경우 비동기적 코어들이 통합된 store 큐에 액세스하는 다이어그램을 도기한 도면.

도 21은 본 발명의 하나의 실시예에 따른, store가 다른 스레드에서의 대응하는 store보다 높은 서열을 가지는 경우의 기능을 나타낸 다이어그램을 도기한 도면.

도 22는 본 발명의 하나의 실시예에 따른, 비명확(non-disambiguated) 비순차 load store 큐 회수 구현(out of order load store queue retirement implementation)을 도기한 도면.

도 23은 본 발명의 하나의 실시예에 따른, 비명확화된 비순차 load store 큐 재정렬 구현(non-disambiguated out of order load store queue reordering implementation)의 재정렬 구현을 도기한 도면.

도 24는 본 발명의 하나의 실시예에 따른, 명령어 시퀀스(예컨대, 트레이스) 재정렬된 투기적 실행 구현(an instruction sequence (e.g., trace) reordered speculative execution implementation)을 도기한 도면.

도 25는 본 발명의 하나의 실시예에 따른, 한 예시적인 마이크로프로세서 파이프라인의 다이어그램을 도기한 도면.

발명을 실시하기 위한 구체적인 내용

[0010] 본 발명이 하나의 실시예와 관련하여 기술되어 있지만, 본 발명이 본 명세서에 기재된 구체적인 형태들로 제한되는 것으로 보아서는 안된다. 그와 달리, 본 발명은 첨부된 청구항들에 의해 한정되는 본 발명의 범주 내에 타당하게 포함될 수 있는 이러한 대안들, 수정들 및 등가물들을 포함하는 것으로 보아야 한다.

[0011] 이하의 상세한 설명에서, 구체적인 방법 순서들, 구조들, 요소들 및 연결들과 같은 수많은 구체적인 상세들이 기재되어 있다. 그렇지만, 이들 및 기타 구체적인 상세가 본 발명의 실시예들을 실시하는 데 이용될 필요는 없다는 것을 잘 알 것이다. 다른 상황들에서, 본 설명을 불필요하게 모호하게 하는 것을 피하기 위해 공지된 구조들, 요소들, 또는 연결들이 생략되어 있거나, 특별히 상세히 기술되어 있지 않다.

[0012] 본 명세서 내에서 "하나의 실시예" 또는 "일 실시예"라고 말하는 것들은 그 실시예와 관련하여 기술된 특징의 특징, 구조 또는 특성이 본 발명의 적어도 하나의 실시예에 포함되어 있다는 것을 나타내기 위한 것이다. 본 명세서 내의 다양한 곳들에서 나오는 "하나의 실시예에서"라는 문구는 모두가 반드시 동일한 실시예를 말하는 것은 아니며 다른 실시예들과 상호 배타적인 별도의 또는 대안의 실시예들도 아니다. 더욱이, 일부 실시예들은 나타내고 있을 수 있지만 다른 실시예들은 그렇지 않을 수 있는 다양한 특징들이 기술되어 있다. 이와 유사하게, 일부 실시예들에 대해서는 요구사항들일 수 있지만 다른 실시예들에 대해서는 그렇지 않을 수 있는 다양한 요구사항들이 기술되어 있다.

[0013] 이하의 상세한 설명의 어떤 부분들은 컴퓨터 메모리 내의 데이터 비트들에 대한 동작들의 절차들, 단계들, 논리 블록들, 처리 및 다른 심볼 표현들로 제시되어 있다. 이들 설명 및 표현은 데이터 처리 분야의 통상의 기술자가 자신의 작업의 내용을 다른 통상의 기술자들에게 가장 효과적으로 전달하기 위해 사용되는 수단이다. 절차, 컴퓨터 실행 단계, 논리 블록, 프로세스 등은 여기에서 일반적으로 원하는 결과를 가져오는 자기 모순이 없는 단계들 또는 명령어들의 시퀀스인 것으로 생각된다. 단계들은 물리적 양의 물리적 조작을 필요로 하는 것이다. 보통, 꼭 그렇지는 않지만, 이들 양은 컴퓨터 판독가능 저장 매체의 전기 또는 자기 신호의 형태를 취하고, 컴퓨터 시스템에서 저장, 전송, 결합, 비교, 및 다른 방식으로 조작될 수 있다. 원칙적으로 흔히 사용되기 때문에, 이들 신호를 비트, 값, 요소, 심볼, 문자, 용어, 숫자 등으로 지칭하는 것이 때로는 편리한 것으로 밝혀졌다.

[0014] 그렇지만, 이들 및 유사한 용어 모두가 적절한 물리적 양과 연관되어 있고 이들 양에 적용되는 편리한 명칭들에 불과하다는 것을 염두에 두어야 한다. 달리 구체적으로 언급하지 않는 한, 이하의 논의로부터 명백한 바와 같이, 본 발명 전체에 걸쳐 "처리" 또는 "액세스" 또는 "기입" 또는 "저장" 또는 "복제" 등과 같은 용어를 이용한 논의가 컴퓨터 시스템의 레지스터 및 메모리 및 다른 컴퓨터 판독가능 매체 내에서 물리적(전자적) 양으로 표현된 데이터를 조작하여 컴퓨터 시스템 메모리 또는 레지스터 또는 다른 이러한 정보 저장, 전송 또는 디스플레이

디바이스 내에서 물리적 양으로 유사하게 표현되는 다른 데이터로 변환하는 컴퓨터 시스템 또는 유사한 전자 컴퓨팅 디바이스의 동작 및 프로세스를 말한다는 것을 잘 알 것이다.

[0015] 본 발명의 실시예들은 입력 명령어 시퀀스 내의 명령어들이, 프로세서 자원들이 이들을 실행하는 데 이용가능하게 되자마자, 비순차적으로 발행될 수 있는 비순차 명령어 스케줄링 프로세스(out of order instruction scheduling process)를 구현한다. 본 발명의 실시예들은 명령어들이 순차적으로 실행되는 것(예컨대, 메모리 일관성 규칙/모델)을 외부 에이전트들이 보도록 보장할 수 있다. 명령어들이 순차적으로 실행되는 것을 외부 에이전트들이 볼 수 있도록 보장하는 것은 그로써 오류 없는 프로그램 실행을 보장한다. 본 발명의 실시예들은 프로세서의 메모리 계층구조(예컨대, L1 캐시, L2 캐시, 시스템 메모리 등)가 명령어들의 일관성있는 순차 실행(consistent in order execution)을 보도록 보장한다.

[0016] 도 1은 본 발명의 하나의 실시예에 따른, load 큐 및 store 큐를 도시한 것이다. 도 1은 또한 입력 명령어 시퀀스를 도시하고 있다. 앞서 기술한 바와 같이, 프로세서의 메모리 계층구조(예컨대, L1 캐시, L2 캐시, 시스템 메모리 등)는 명령어들의 일관성있는 순차 실행을 본다. load 큐 및 store 큐(이후부터 종종 load/store 큐라고 함)는 순차 실행의 시맨틱스(semantics)를 유지하는 데 사용될 수 있다.

[0017] 그에 추가하여, load 및 store의 비순차 실행은 투기적 실행을 가져온다. 투기적 실행을 수행할 때, 기계는 투기적 오류들을 인식할 필요가 있다. 도 1의 실시예에서, load/store 큐는 load store 재정렬 및 최적화로부터 생기는 투기적 포위딩 또는 예측 착오/오류로부터의 복원을 구현하는 시스템을 제공한다. load/store 큐는 포위딩, 분기(branch) 및 폴트(fault)의 결과로서 load store 재정렬/최적화로부터 생기는 투기적 오류로부터의 복원을 가능하게 하는 하드웨어 지원을 포함한다. 기계가 투기적 오류로부터 복원할 수 있게 하기 위해, 투기적 실행의 결과가 load 큐 및 store 큐에 유지된다. load 큐 및 store 큐는, 오류가 정정될 수 있고 store 결과가 메모리로 회수될 수 있을 때까지, 투기적 실행의 결과를 보유한다. load 큐 및 store 큐의 투기적 실행 내용은 외부 에이전트들에게 보이지 않는다. 가시성(visibility)과 관련하여, store는 순차적으로 메모리로 회수될 필요가 있다.

[0018] 도 2는 본 발명의 하나의 실시예에 따른, load 및 store 명령어 분할의 제1 다이어그램을 도시한 것이다. 본 발명의 하나의 특징은 load가 2개의 매크로명령어(macroinstruction)로 분할되고, 제1 매크로명령어는 주소 계산 및 임시 장소(load store 큐)로의 페치를 행하고, 제2 매크로명령어는 메모리 주소 내용(데이터)의 레지스터 또는 ALU 목적지로의 load라는 사실이다. 유의할 점은, 본 발명의 실시예들이 load 명령어 및 store 명령어를 2개의 각자의 매크로명령어로 분할하고 이들을 재정렬하는 것과 관련하여 기술되어 있지만, 동일한 방법 및 시스템이 load 명령어 및 store 명령어를 2개의 각자의 마이크로명령어(microinstruction)로 분할하고 마이크로코드 컨텍스트 내에서 이들을 재정렬하는 것에 의해 구현될 수 있다는 것이다.

[0019] 이 기능은 store에 대해 동일하다. store도 또한 2개의 매크로명령어로 분할된다. 제1 명령어는 store address 및 fetch이고, 제2 명령어는 그 주소에의 데이터의 store이다. store의 분할 및 2개의 명령어는 load에 대해 이하에 기술된 것과 동일한 규칙을 따른다.

[0020] load의 2개의 명령어로의 분할은 런타임 최적화(runtime optimizer)가 주소 계산 및 페치 명령어를 주어진 명령어 시퀀스 내에서 훨씬 더 빨리 스케줄링할 수 있게 한다. 이것은 데이터를 캐시 계층구조와 별개인 임시 버퍼로 프리페치(prefetch)하는 것에 의해 메모리 미스(memory miss)로부터의 보다 용이한 복원을 가능하게 한다. 임시 버퍼는 LA/SA와 LD/SD 사이의 일대일 대응관계에 대해 프리페치된 데이터의 이용가능성을 보장하기 위해 사용된다. load address와 load data 사이의 윈도우에 있는 이전의 store와의 엘리어싱(aliasing)이 있는 경우(예컨대, 이전의 store로부터 포위딩 경우가 검출되는 경우), 또는 주소 계산에서 임의의 폴트 문제(예컨대, 페이지 폴트(page fault))가 있는 경우, 대응하는 load data 명령어가 재발행될 수 있다. 그에 추가하여, load들의 2개의 명령어들로의 분할은 또한 정보를 2개의 명령어들 내로 복제하는 것을 포함할 수 있다. 이러한 정보는 주소 정보, 소스 정보, 다른 부가의 식별자 등일 수 있다. 이 복제는 LA/SA가 없는 경우 2개의 명령어의 LD/SD의 독립적인 디스패치를 가능하게 한다.

[0021] load address 및 fetch 명령어는 load data가 다시 시작하기를 기다리는 일 없이 실제의 기계 회수 윈도우(machine retirement window)로부터 회수될 수 있고, 그로써 그 주소(예컨대, 이 단락의 시작에서 언급된 load address)에 대한 캐시 미스(cache miss)의 경우에도 기계가 전방으로 진행할 수 있게 한다. 예를 들어, 그 주소(예컨대, 주소 X)에 대한 캐시 미스 시에, 기계는 어찌면 데이터가 메모리 계층구조로부터 페치되기를 기다리면서 수백 사이클 동안 정지(stall)될 수 있다. load data가 다시 시작하기를 기다리는 일 없이 실제의 기계 회수 윈도우로부터 load address 및 fetch 명령어를 회수하는 것에 의해, 기계는 여전히 전방으로 진행할 수 있

다.

- [0022] 유의할 점은, 명령어들의 분할이 load들 및 store들의 보다 빠른 디스패치 및 실행을 가능하게 하기 위해 명령어 시퀀스에서 보다 빨리 그리고 LD/SD로부터 더 멀리 떨어지게 LA/SA 명령어들을 재정렬하는 본 발명의 실시예들의 주요 장점을 가능하게 한다는 것이다.
- [0023] 도 3은 본 발명의 하나의 실시예에 따른, load 및 store 명령어 분할의 제2 다이어그램을 도시한 것이다. 도 2의 실시예는 load 명령어의 분할을 수행하기 위해 복제 특징이 어떻게 사용되는지를 나타내고 있다. 이 실시예에서, load는 2개의 매크로명령어로 복제되고, 제1 매크로명령어는 주소 계산 및 임시 장소(load store 큐)로의 페치를 행하고, 제2 매크로명령어는 메모리 주소 내용(데이터)의 레지스터 또는 ALU 목적지로의 load이다. 유의할 점은, 본 발명의 실시예들이 load 명령어 및 store 명령어를 2개의 각자의 매크로명령어로 복제하고 이들을 재정렬하는 것과 관련하여 기술되어 있지만, 동일한 방법 및 시스템이 load 명령어 및 store 명령어를 2개의 각자의 마이크로명령어로 복제하고 마이크로코드 컨텍스트 내에서 이들을 재정렬하는 것에 의해 구현될 수 있다.
- [0024] 본 발명의 하나의 실시예에 따른 명령어 분할 기능의 보다 상세한 설명이 이제부터 기술된다. 하나의 실시예에서, 명령어 세트는 LA, SA, LD 또는 SD에 대한 직접 유사 명령어(direct analogue instruction)를 갖지 않는다. 이러한 실시예에서, 이 개념은 명령어 프리픽스(instruction prefix) LAF, SAF, LASAF와 컴패니언 서픽스 명령어(companion suffix instruction)의 조합으로 실현된다. 그리고 LA가 LAD를 갖고 SA가 SAD를 갖는 것으로의 매핑을 대략적으로 행하는 명령어 세트, 및 결합된 LADSAD가 구현될 수 있다. 이 개념은 또한 마이크로코드 내의 마이크로명령어로서 구현될 수 있다.
- [0025] a) 여기서 LAF-프리픽스+서픽스 명령어로서 정의되는 것은 'LD'로서 기술될 수 있다.
- [0026] b) 여기서 SAF-프리픽스+서픽스 명령어로서 정의되는 것은 'SD'로서 기술될 수 있다.
- [0027] c) 여기서 LAD 명령어로서 정의되는 것은 'LA'로서 기술될 수 있다.
- [0028] d) 여기서 SAD 명령어로서 정의되는 것은 'SA'로서 기술될 수 있다.
- [0029] e) 게다가 LASAF 명령어 및 LADSAD 명령어를 가진다. 이들 명령어는 세마포어(로크된 원자적(locked-atomic)) 동작을 구현하는 데 사용될 수 있는 결합된 LAF/SAF-프리픽스+서픽스 명령어를 포함한다. 메모리 피연산자를 또다시 프리페치하기 위해 결합된 LAD-SAD 명령어를 정의하는 것도 또한 가능하고, 그 결과 하드웨어가 복잡해진다.
- [0030] LAD는 'LA-defused'를 나타낸다.
- [0031] LAD 명령어는 실행 파이프라인 내로의 데이터 프리페치(data-prefetch)를 개시한다. 이는 실행 파이프라인 내로 직접 로드하여 제1 레벨 캐시보다 더 낮은 실행 지연 시간을 제공한다는 점에서 보통의 프리페치와 상이하다. 하나의 실시예에서, 이 기능은 LA-LD 쌍(LA-LD pair) 사이의 ID 링크(예컨대, QID 번호)를 사용하여 태깅될 수 있는 LA-LD 쌍에 대한 고정 기억 장치(fixed storage)를 사용하는 것에 의해 구현된다. LAD 명령어는 (예컨대, 어쩌면 복잡한 규격으로부터) 유효 메모리 주소를 계산하고, 피연산자 크기(바이트, 하프 워드(half word), 워드, 더블 워드(double word), 또는 그 이상)를 명시하며; TLB 및 캐시 계층구조를 통해 메모리 참조를 개시한다. 예외(페이지 워크 미스(page walk miss), 특권, 보호)가 LAF+서픽스 실행에서 보고되도록 기록되거나, 대안의 구현은 큐 엔트리를 취소/무효화하여, LAF+서픽스 명령어를 강제로 재실행시켜 예외를 취하게 할 수 있다.
- [0032] LAD 명령어는 일반 포맷 및 피연산자들을 가진다:
- [0033] LAD(os) QID,M[EA]
- [0034] • EA - 베이스 레지스터(base-register), 인덱싱 레지스터(indexing register), 시프팅 인자(shifting factor) 및/또는 인덱싱 오프셋(indexing offset)의 조합일 수 있는 유효 주소 규격이다. 예컨대, M[B,RI,sf,offset]
- [0035] • os - 판독될 바이트의 수의 표시이다.
- [0036] • QID - 메모리 참조 연산을 위해 사용될 load memory QID이다. 이는 또한 LAD의 연산과 후속하는 LAF-프리

픽스형 명령어(LAF-prefixed instruction)를 링크시키는 데 사용된다. QID는 1부터 N까지의 범위에 있으며, N은 구현 특정 값이다. 예상된 값은 31, 63, 127이다. QID=0은 선행 LAD를 갖지 않는 LAF 명령어의 특수한 경우를 위해 예비되어 있다. QID=0은 항상 LAF에 의해 즉각 사용되며, 그에 따라 LAD에 대해 이용가능하지 않다.

[0037] LAF는 'LA-fused'를 나타낸다.

[0038] LAF는 서픽스 명령어와 직접 결합(또는 융합(fuse))되어 있어야만 한다는 것을 나타내는 명령어 프리픽스이다. 서픽스 명령어는 독립형(stand alone)일 수 있다. 서픽스 명령어는 적어도 하나의 소스 레지스터를 가지는 임의의 명령어일 수 있다. LAF는 프리픽스로서 결합되어야만 한다. LAF-프리픽스는 서픽스 명령어의 속성을 변경한다. 그의 레지스터 피연산자들 중 하나 이상은 프리픽스에 의해 메모리 큐 식별자(memory queue identifier)(QID)로서 재정의된다. 게다가, 레지스터로부터 소싱(source)되는 것으로 연관되어 있는 데이터는 이제 메모리 큐로부터 소싱된다.

[0039] LAF-프리픽스+서픽스 명령어는 선행 LAD 명령어를 가질 수 있거나 그렇지 않을 수 있다. QID == 0인 경우, LAF는 선행 LAD를 갖지 않는다. QID != 0인 경우, LAF는 선행 LAD를 가지거나 가졌었다. load 명령어의 LA 및 LD로의 분할을 생성하는 것이 의도되어 있을 때, LAF는 QID!=0을 가질 것이고 LAD는 LAF에 선행하는 동일한 QID로 인스턴스화될 것이다(예컨대, 이것은 기본적으로 분할된 LA/LD 쌍을 생성한다).

[0040] LAF/서픽스가 실행되고 QID == 0을 가질 때, 메모리 큐의 0 엔트리는 'LA' 연산, 메모리 판독을 수행하고, 데이터를 메모리 큐 내로 스테이징(stage)하기 위해 사용되고, 이어서 데이터를 서픽스 명령어 소스 내로 로드하고 연산이 잠재적인 다른 소스와 결합되어 적용되며 결과가 서픽스 명령어 목적지 레지스터(들)에 기입되는 것에 의해 완료된다. 다른 한편으로는, QID!=0인 경우, 일치하는 QID가 있는지 메모리 큐가 조회(탐색)되고, 존재하는 경우, 데이터가 메모리 큐로부터 판독되고 서픽스 명령어의 연산이 적용되며, 결과가 목적지 레지스터에 기입된다. 일치하는 QID가 유효하지만 완성되지 않은 경우, 데이터가 이용가능할 때까지 데이터가 정지된다. QID가 유효하지 않은 경우, LAF는 연산을 재시작하기에 충분한 정보(주소 및 데이터 피연산자 크기)를 가지고 있다. 일치하는 QID가 다양한 이유로 존재하지 않을 수 있고, 그 이유들 중 일부는 다음과 같다:

[0041] a) 선행 LAD가 결코 실행되지 않음, 나쁜 코딩 또는 다른 이유.

[0042] b) LAF와 선행 LAD 사이의 실행 흐름이 예외 또는 인터럽트에 의해 중단되었음

[0043] c) 개재된 store 연산들이 LAD의 주소와 엘리머싱되고 그의 데이터 무결성을 무효화시킴.

[0044] 이 경우들 중 임의의 것에서, LAF 프리픽스+서픽스는 LAD(LA) 연산을 반복하기에 충분한 정보를 가진다. 이 능력은 그 LAD 명령어를 힌트로 만든다. LAD는 성공적으로 실행될 필요가 없었거나 그 점에 대해서는 올바른 코드가 그것을 사용하기 위해 NOP인 것을 넘어서 심지어 구현될 필요가 없었다.

[0045] 서픽스 명령어를 갖는 LAF 명령어의 일반 포맷 및 피연산자들은 다음과 같다:

[0046] LAF M[ea]

[0047] SUFFIX(os) Rt,QID,...

[0048] LAF 명령어는 서픽스 명령어의 인코딩으로부터 피연산자 크기, QID를 빌려온다(borrow). 서픽스가 SIMD인 경우, 이는 또한 서픽스로부터 연산의 SIMD 폭을 빌려온다. QID는 항상 서픽스 명령어의 소스 레지스터 지정 필드(source register specification field)들 중 하나에 인코딩된다. SMI의 특정의 구현에서, 이것은 항상 비트들 23:18이지만, 이것이 그럴 필요는 없다.

[0049] SAD는 'SA-defused'를 나타낸다.

[0050] SAD는 store에 대해서만 LAD에 대한 병렬 명령어(parallel instruction)이다. 이는 또한 데이터를 프리페치하여, 수정을 위해 데이터를 캐시로 가져온다. 게다가, 이는 메모리 저장 큐 엔트리(memory-store-queue entry)를 생성한다. SAD 프라이머리(SAD primary)는 2가지 주된 용도를 가진다:

[0051] a) 프리페치로서, 수정을 위한 데이터의 판독

[0052] b) 올바른 메모리 정렬(memory ordering)을 유지하고 store(기입) 이전에 load(판독)를 승격(promote)시킨 후에 잠재적인 판독후 기입(write-after-read) 위험을 노출시키고 처리하는 것.

[0053] SAD는 힌트 명령어(hint instruction)이다. SAD 명령어는 (어쩌면 복잡한 규격으로부터) 유효 메모리 주소를 계산하고, 피연산자 크기(바이트, 하프 워드, 워드, 더블 워드, ...)를 명시하며; TLB, 캐시/메모리 계층구조를

통해 메모리 참조를 개시한다. 재실행하여 예외를 취하기 위해 예외(페이지 워크 미스, 특권, 보호)가 SAF+서픽스 실행에서 기록된다.

[0054] SAD 명령어는 일반 포맷 및 피연산자들을 가진다:

[0055] SAD(os) M[ea],QID

[0056] • Ea - 베이스 레지스터, 인덱싱 레지스터, 시프팅 인자 및/또는 인덱싱 오프셋의 조합일 수 있는 유효 주소 규격이다. 예컨대, M[B,RI,sf,offset]

[0057] • Os - Ea에 기입될 바이트의 수의 표시이다.

[0058] • QID - 메모리 참조 연산을 위해 사용될 store memory QID이다. 이는 또한 SAD의 연산과 후속하는 SAF 프리픽스형 명령어(SAF prefixed instruction)를 링크시키는 데 사용된다. QID는 1부터 N까지의 범위에 있으며, N은 구현 특정 값이다. 예상된 값은 31, 63, 127이다. QID=0은 선행 SAD를 갖지 않는 SAF 명령어의 특수한 경우를 위해 예비되어 있다. 이 QID는 항상 SAF에 의해 즉각 사용된다. SAF는 'SA-fused'를 나타낸다.

[0059] SAF는 store에 대해서만 LAF 프리픽스에 대한 병렬 프리픽스(parallel prefix)이다. 프리픽스로서, 이는 서픽스 명령어와 직접 결합(또는 융합)되어 있어야만 한다. 서픽스 명령어는 독립형일 수 있다. 서픽스 명령어는 적어도 하나의 목표 레지스터를 가지는 임의의 명령어일 수 있다. SAF는 프리픽스로서 결합되어야만 한다. SAF는 서픽스 명령어의 속성을 변경한다: 보통 메모리 저장 큐 식별자(QID)에 대한 레지스터-선택 인덱스인 목적지 레지스터 피연산자들 중 하나 이상, 및 레지스터를 목표로 하는 것으로부터 메모리(보다 정확하게는, 메모리 큐 엔트리)를 목표로 하는 것으로의 연산. 그에 따라, 이는 레지스터 연산을 store memory 연산으로 변경한다.

[0060] SAF 명령어는 선행 SAD를 가질 수 있거나 그렇지 않을 수 있다. QID == 0인 경우, SAF는 선행 SAD를 갖지 않는다. QID != 0인 경우, SAF는 선행 SAD를 가지거나 가졌었다. SAF/서픽스가 실행될 때, QID == 0인 경우, 메모리 큐의 0 엔트리는 'SA' 연산, 메모리 기입을 수행하고, 데이터를 메모리 큐 내로 스테이징하기 위해 사용되고, 이어서 서픽스 명령어 소스에 의해 제공되는 데이터를 저장하는 것에 의해 완료된다. 다른 한편으로는, QID!=0인 경우, 일치하는 QID가 있는지 메모리 큐가 조회(탐색)되고, 존재하는 경우, 서픽스 명령어의 연산이 적용될 때 데이터가 메모리 큐 엔트리 내로 기입될 것이다. 일치하는 QID가 유효하지만 완성되지 않은 경우, 데이터가 이용가능할 때까지 데이터가 정지된다. QID가 유효하지 않은 경우, SAF는 연산을 재시작하고 메모리 기입 연산을 완성하기에 충분한 정보(주소 및 데이터 피연산자 크기)를 가진다. 일치하는 QID가 다양한 이유들로 존재하지 않을 수 있고, 그 이유들 중 일부는 다음과 같다:

[0061] a) 선행 SAD가 결코 실행되지 않음, 나쁜 코딩 또는 다른 이유.

[0062] b) SAF와 선행 SAD 사이의 실행 흐름이 예외 또는 인터럽트에 의해 중단되었음

[0063] 이 경우들 중 임의의 것에서, SAF 프리픽스+서픽스는 SAD(SA) 연산을 반복하기에 충분한 정보를 가진다. 이 능력은 우리의 SAD 명령어를 힌트로 만든다. SAD는 성공적으로 실행될 필요가 없었거나 그 점에 대해서는 올바른 코드가 그것을 사용하기 위해 NOP인 것을 넘어서 심지어 구현될 필요가 없었다.

[0064] LASAF는 명령어 프리픽스이다.

[0065] LASAF는 프리픽스로서 소스 및 목적지와 동일한 레지스터를 가지는 명령어를 수식한다(modify). LASAF는 이러한 명령어를 원자적 메모리 참조 한번 판독/기입 연산(atomic memory reference read/write once operation)으로 변경한다. load memory 큐로부터의 것 및 store memory 큐로부터의 것이 사용된다. 선행 LAD 또는 SAD 명령어가 없다.

[0066] LASAF M[ea3]

[0067] ADD QID1,QID2,R1

[0068] LASAF는 load memory 큐 및 store memory 큐 둘 다에 QID 엔트리들을 생성한다. 그 다음 QID2를 사용하여 메모리 [ea3]를 판독하고, R1을 가산하며 결과를 Store 메모리 QID1에 저장하고, M[ea3]의 원자적 판독-수정 기입을 수행한다.

[0069] 일 실시예에서 구현은 QID1=QID2=0을 필요로 할 것이다. 그렇지만, 본 발명은 이 실시예로 제한되는 것이 아니

다.

- [0070] 일 실시예에서, LASAD 명령어가 가능할 수 있지만, 그것을 구현하기 위해 메모리 큐들을 계속하여 스누핑(snoop)하고, 히트 시에 스누핑에 대한 양 엔트리들을 무효화시키고, 이어서 LASAF에 대해 load/store를 재실행할 필요가 있을 것이다.
- [0071] 예시적인 사용:
- [0072] A. 데이터의 사용보다 훨씬 이전에 load를 승격시킨 후에 레지스터 자원을 절감한다.
- [0073] 원래의 코드가 다음과 같은 것으로 가정한다.
- [0074] LDR R1,M[ea1]
- [0075] ADD32 Rt,R1,R2
- [0076] 메모리 액세스 지연 시간을 감추기 위해, R1 데이터(ADD)의 사용보다 가능한 한 빨리 실행 흐름에서 LDR를 승격시키고자 한다.
- [0077] LDR R1,M[ea1]
- [0078] ... 많은 명령어들
- [0079] ADD32 Rt,R1,R2
- [0080] 이것을 하는 것의 하나의 불리한 면은 R1 레지스터가 데이터를 기다리느라 '비지 상태(busy)'로 있고, 다른 목적을 위해 사용될 수 없다는 것이다. 메모리 큐는 데이터를 보유할 자원들의 풀을 확장시킨다. 따라서 LDR을 LAD 및 후속하는 LAD로 변환한다:
- [0081] LAD QID,M[ea1]
- [0082] ... 많은 명령어들
- [0083] LAF M[ea1]
- [0084] ADD32 Rt,QID,R2
- [0085] load 큐 엔트리 QID가 사용되기 때문에, R1이 다른 목적을 위해 자유롭게 사용된다. 또는 Rt에 Rt-QID의 차를 로드하거나, QID가 존재하지 않는 경우, M[ea1]으로부터 데이터를 재로드하고 그로부터 R2를 감산하며 결과를 Rt에 위치시킨다.
- [0086] 유의할 점은, 앞서 기술한 구현에서, 메모리 주소 계산이 2개의 load들 간에 일치할 필요가 없다. 2개의 주소 계산이 상이하고 QID가 여전히 유효하다면, 프로그래머가 좋지 않을 가능성이 많다. OS(이 예에서, 32)가 또한 LAD와 LAF 간에 일치할 필요가 없다. 필요 이상의 데이터가 LAD에 의해 판독될 수 있고, 이 경우에 판독된 데이터의 최하위 바이트가 사용될 것이다. 또는 판독된 LAD보다 더 많은 데이터가 LAF+서픽스에 의해 필요하게 될 수 있고, 이 경우에 LAD에 의해 판독된 최하위 바이트가 사용될 것이고, 뒤이어서 서픽스 연산이 만족될 때까지 0이 뒤따를 것이다. 게다가, 주소 계산 피연산자들이 LAD와 LAF 간에 일치할 필요가 없지만, 양호한 코딩을 위해, 이들이 동일한 결과적인 유효 주소를 가져야만 한다.
- [0087] B. 통상적인 ALU 레지스터-레지스터 연산을 ALU 레지스터 메모리 연산으로 모핑(morphing)하는 것에 의해 실행 사이클들을 절감한다.
- [0088] 여기서 통상적인 레지스터-레지스터 ADD 명령어를 메모리 참조로 단지 변경하기 위해 특수 QID=0(%0)을 사용하고 있다. LAF가 프리픽스이고 ADD에 직접 결합되어야만 하기 때문에, 이들 사이에 명령어가 없다. QID=0은 사용하도록 항상 이용가능한데, 그 이유는 그것이 즉각 사용되기 때문이다.
- [0089] LAF M[ea1]
- [0090] ADD32 Rt,%q0,R2
- [0091] LAF는 이상의 명령어를
- [0092] ADD32 Rt,M[ea1],R2

- [0093] 로 효과적으로 변경한다.
- [0094] 또한, ALU 레지스터-레지스터 연산을 ALU 연산의 결과를 메모리에 저장하는 연산으로 모핑하기 위해 SAF를 사용할 수 있다.
- [0095] SAF M[ea1]
- [0096] ADD %q0,R2,R3
- [0097] 는 R2와 R3를 가산한 결과를 주소 ea1에 있는 메모리에 저장할 것이다.
- [0098] C. store보다 load를 승격시킬 때 순서 시맨틱을 유지한다.
- [0099] 다른 문제는 load(LDR)를 load의 주소와 엘리어싱할 수 있거나 그렇지 않을 수 있는 store(STR)보다 승격시키고자 한다는 것이다. 엘리어스: ea1에 의한 데이터 주소의 일부 또는 전부가 ea2와 동일하다.
- [0100] STR M[ea2]
- [0101] LDR R1,M[ea1]
- [0102] ADD32 Rt,R1,R2
- [0103] 가
- [0104] LDR R1,M[ea1]
- [0105] 0 내지 다수개의 명령어들
- [0106] STR M[ea2],R3
- [0107] 0 내지 다수개의 명령어들
- [0108] ADD32 Rt,R1,R2
- [0109] 로 된다.
- [0110] 이것을 안전하게 행하기 위해, 실행 순서를 유지하고 올바른 코드를 발생시킬 수 있도록 LAD 및 SAD 명령어들과 그 각자의 LAF 및 SAF 프리픽스+서픽스 명령어들을 사용할 수 있다.
- [0111] LOOP:
- [0112] SAD M[ea2],R3
- [0113] 0 내지 다수개의 명령어들
- [0114] a)
- [0115] LAD R1,M[ea1]
- [0116] 0 내지 다수개의 명령어들
- [0117] b)
- [0118] SAF M[ea1],R3
- [0119] 0 내지 다수개의 명령어들
- [0120] c)
- [0121] saf-suffix
- [0122] LAF M[ea1]
- [0123] d)
- [0124] BRN LOOP
- [0125] 이상의 코드에서, LAD 및 SAD 둘 다 승격되고 동일한 순서를 유지할 필요가 있다. 무슨 일이 일어날 수 있는가? 각각의 시점 a, b, c, d에서, 대안이 표시되어 있다.

- [0126] a1) 인터럽트하고, SAD를 무효화시키고, 후속하는 SAF가 재실행되어야만 할 것이다
- [0127] a2) LAD가 SAD와 엘리어싱하고, LAD를 무효화시키거나 오히려 메모리 큐에 삽입되지 않을 것이다
- [0128] b1) 인터럽트하고, SAD 및 LAD를 무효화시킨다
- [0129] b2) SAF는 LAD와 엘리어싱하고, LAD를 무효화시킨다
- [0130] b3) SAF는 여전히 유효한 SAD를 사용하거나 재실행된다.
- [0131] c1) 인터럽트하고, LAD를 무효화시키고,
- [0132] c2) 여전히 유효한 LAF가 LAD의 데이터를 사용하는 경우, 다른 방식으로 재실행한다.
- [0133] c3) 루프를 돌고, 하드웨어의 매직(magic of hardware)으로 해서, IP 및 실행 시퀀스 ID로 태깅하는 것과 QID의 조합, LAD/SAD/LAF/SAF가 적절히 관리된다.
- [0134] LA/SA 및 LD/SD의 앞서 기술한 설명에서, 포워딩을 위한 순서를 시행하기 위해 LA 및 SA 상대 프로그램 순서 위치들이 사용된다. 다른 실시예에서, (예컨대, 이하에서 기술하는 바와 같이) 포워딩을 위한 순서를 시행하기 위해 LD/SD 상대 프로그램 순서 위치들이 사용될 수 있다.
- [0135] 도 4는 본 발명의 하나의 실시예에 따른, load store 재정렬 및 최적화로부터 생기는 투기적 포워딩 예측 착오/오류로부터의 복원을 구현하는 규칙들이 나타내어져 있는 프로세스(400)의 단계들의 플로우차트를 도시한 것이다.
- [0136] 단계(401)로 나타낸 바와 같이, 본 발명의 실시예의 목적은 그 store와 그 load 간의 주소 일치 시에 load로 포워딩할 store를 찾아내는 것이다. 단계(402)에서, (예컨대, 기계 순서에서) 가장 가까운 이전의 store가 load로 포워딩된다.
- [0137] 단계(403)에서, LD/SD가 기계 순서로 할당될 때 LA/SA에 대해 실제 에이지(actual age)가 업데이트된다. LA/SA 실제 에이지는 LD/SD 에이지와 동일한 값을 할당받는다. LD/SD는 실제 에이지를 유지하고 원래의 프로그램 순서 시맨틱스를 시행한다.
- [0138] 단계들(404 내지 407)은 투기적 실행을 지원하면서 프로그램 순차적 시맨틱스(program sequential semantics)를 유지하는 규칙들을 나타내고 있다. 단계들(404 내지 407)은 이 규칙들을 구현하는 메커니즘들이 동시에 기능한다는 것을 나타내기 위해 서로 수평으로 배열되어 도시되어 있다. 단계(404)에 도시된 바와 같이, store가 실제 에이지를 갖지만 load는 아직 실제 에이지를 획득하지 않은 경우, store가 load보다 더 빠르다. 단계(405)에 도시된 바와 같이, load가 실제 에이지를 갖지만 store는 아직 실제 에이지를 획득하지 않은 경우, load가 store보다 더 빠르다. 단계(406)에 도시된 바와 같이, load 또는 store 중 어느 것도 실제 에이지를 획득하지 않은 경우, 어느 것이 더 빠른지를 알아내기 위해 가상 식별자(virtual identifier)(VID)가 사용될 것이다(예컨대, 일부 실시예들에서, load/store 명령어들과 연관되어 있는 QID는 VID를 나타낸다). 단계(407)에 도시된 바와 같이, load 및 store 둘 다 실제 에이지를 획득한 경우, 어느 것이 더 빠른지를 알아내기 위해 실제 에이지가 사용된다.
- [0139] 유의할 점은, load와 store 간의 상대 에이지(relative age)를 결정하는 데 사용되는 도 4의 실시예에 의해 기술된 알고리즘이 또한 복수의 store들 간의 상대 에이지를 결정하는 데 사용될 수 있다는 것이다. 이것은 도 4 및 후속 도면들에서 이하에 기술되는 바와 같이 store 에이지 스탬프(store age stamp)를 업데이트하는 데 유용하다.
- [0140] 도 5는 본 발명의 하나의 실시예에 따른, 프로세스(400)의 규칙들이 프로세서의 load 큐 및 store 큐 자원들로 구현되는 방식을 나타낸 다이어그램을 도시한 것이다. 도 5의 실시예는 명령어들의 루프가 2개의 동일한 명령어 시퀀스(401 및 402)로 펼쳐져 있는 한 예를 도시한 것이다. 유의할 점은, SA 및 LA는 자유롭게 재정렬될 수 있지만, SD 및 LD는 그의 상대 프로그램 순서를 유지해야만 한다는 것이다. 보다 빠른 store는 보다 늦은 load로 포워딩될 수 있다. 보다 빠르다는 것은 보다 작은 VID(예컨대, 가상 ID 테이블에 유지됨) 또는 보다 작은 에이지를 의미한다. SA가 VID를 갖지만 에이지를 갖지 않는 경우, 그 SA는 에이지를 가지는 load보다 더 늦다. LA/SA의 실제 에이지는 LD/SD의 할당 시에 업데이트되고 LD/SD의 동일한 에이지를 할당받는다. store 또는 load가 실제 에이지를 가지는 경우, 그것이 실제 에이지와 비교되고, 그렇지 않은 경우, VID 에이지가 사용된다.

- [0141] 유의할 점은, VID 테이블이 LA/SA 대응 기계 ID 및 각각의 VID 고유 식별자에 대응하는 기계 자원들을 저장함으로써 LA/SA와 LD/SD 명령어들 간의 연관 관계를 추적하는 것에 의해 기능한다는 것이다. 또한 유의할 점은, 도 2a 및 도 2b의 논의에서 기술한 바와 같이, "VID"라는 용어가 "QID"라는 용어와 동의어라는 것이다.
- [0142] 도 4의 실시예의 동작의 한 예가 이제부터 기술된다. 처음에, 할당 포인터(410)가 초기에 3이었던 경우를 생각해 보자. V3 LA는 디스패치되고 load 큐 엔트리 #4에 할당되었다. V1 SA 및 V2 SA 둘 다 디스패치되었다. 이들이 V3 LA와 비교되고, V2 SA가 V3 LA보다 더 작고 V1 SA보다 그에 더 가깝기 때문에, 그는 어쩌면 V3 LA로 포워딩되고, 따라서 V3 LA load 큐 엔트리에 대한 store 초기 에이지(store initial age)를 업데이트한다.
- [0143] 할당 포인터는 이제 6으로 이동한다. V2 SA(#5)의 store 실제 에이지(store actual age)는 이제 V3 LA load 큐 엔트리를 업데이트한다(그 이유는 V2 SA가 이 load로 포워딩하도록 스탬핑되어 있는 기록의 저장이기 때문이다). V4 SA가 이제 디스패치되고 load 초기 에이지(load initial age)와 비교되며, V4가 V3 LA보다 더 크기 때문에, 그는 포워딩되지 않는다. 할당 포인터는 이제 11로 이동한다. V3 LD의 할당 시에, 그는 load 큐 엔트리 #4를 V3 LD(#7)의 실제 에이지로 업데이트한다. V1 SA #11이 이제 디스패치된다. V3 LA #1은 이제 실제 에이지를 갖지만 V1 SA #11은 그렇지 않기 때문에, load가 store보다 더 빠르며, 따라서 포워딩이 가능하지 않다.
- [0144] 예측 테이블은 기본 가정이 틀렸던 경우들을 검출하기 위한 것이다. 기본 가정은 store가 load로 포워딩되지 않는다는 것이다. load store 쌍에 대해 포워딩이 검출되면, 그 store 주소가 디스패치되기를 load가 항상 기다리도록 load store 쌍의 프로그램 카운터가 기록되고 그 load 주소가 그 store 주소와 일치하는지를 알아내기 위해 주소가 계산되고 그에 따라서 그로부터 포워딩될 필요가 있다.
- [0145] 유의할 점은, 하나의 실시예에서, 본 명세서에 기술된 특징 - LA/SA가 없는 경우에 LD/SD가 디스패치될 수 있음 - 이 주어진 명령어들의 시퀀스에서 분기에 앞서 또는 분기 범위 내에서 LA/SA의 재정렬을 용이하게 한다는 것이다. 분기의 결과로서 LA 및 SA를 건너뛰었거나, 메모리 예외를 야기한 결과로서 LA 및 SA가 무시된 경우, LD 및 SD는 여전히 올바르게 기능할 수 있는데, 그 이유는 이들이 두번(처음에는 LA/SA로서 그리고 두번째는 LD/SD로서) 디스패치하는 데 필요한 정보를 포함하고 있기 때문이다. 이러한 경우에, LD/SD의 제1 디스패치는 주소 계산(예컨대, 로드 주소)을 수행하는 것이다. 이어서, load 또는 store의 소비 부분(예컨대, 로드 데이터)을 완수하기 위해 동일한 LD/SD가 또다시 디스패치할 수 있다. 이 메커니즘은 load 및 store 명령어들의 "듀얼 디스패치(dual dispatch)"라고 할 수 있다.
- [0146] 유의할 점은, 하나의 실시예에서, 대응하는 해체된(defused) LA/SA가 존재하지 않을 때(예컨대, 융합된(fused) LD/SD에서와 같이), 또는 분기의 결과로서 LA/SA를 건너뛰었거나 메모리 예외를 야기한 결과로서 LA/SA가 무시된 경우, 또는 이와 유사한 경우, LD/SD의 듀얼 디스패치가 일어난다는 것이다.
- [0147] 앞서 기술한 듀얼 디스패치 기능은 LD/SD가 상실된, 무시된 또는 건너뛴 LA/SA와 관계없이 올바르게 실행되도록 보장한다. 앞서 기술한 특징에 의해 제공된 이점은, 분기, 잠재적 폴트, 예외 등의 존재에도, LA/SA를 더 빨리 스케줄링하는 것에 의해, load/store에 의해 지정된 데이터의 프리페치가 프로그램 순서에서 더 빨리 시작할 수 있다(예컨대, 지연 시간을 감소시킴)는 것이다.
- [0148] 도 6은 본 발명의 하나의 실시예에 따른, 프로세스(400)의 규칙들이 프로세서의 load 큐 및 store 큐 자원들로 구현되는 방식을 나타낸 다른 다이어그램을 도시한 것이다. 도 6의 실시예에서, 할당 포인터가 초기에 3이었던 경우를 생각해 보자. V3 LA는 디스패치되고 load Q 엔트리 #4에 할당되었다. 할당 포인터는 이제 6으로 이동한다. V1 및 V2(#4, #5)의 store 실제 에이지가 이제 대응하는 SA의 것을 기계 ID 2 및 3으로 업데이트한다. V4 SA가 이제 디스패치되고 load 초기 에이지와 비교되며, V4 SA가 V3 LA보다 더 크기 때문에, 그는 포워딩되지 않는다. 할당 포인터는 이제 11로 이동한다. V3 LD의 할당 시에, 그는 load Q 엔트리 #4를 V3 LD(#7)의 실제 에이지로 업데이트한다. 이제, ID 10의 V1 LA가 디스패치된다.
- [0149] 기계 ID 2의 V1 SA 및 기계 ID 3의 V2 SA 둘 다 이제 디스패치된다. 이들은 ID 10의 V1 LA와 비교되고, ID 10의 V1 LA가 기계 에이지(machine age)를 갖지 않는(그의 대응하는 LD가 아직 할당되지 않은) 반면 기계 ID 2의 V1 SA 및 기계 ID 3의 V2 SA 둘 다 실제 에이지를 갖기 때문에, V1 및 V2 store 둘 다 V1보다 더 빠르다/더 오래되었다는 것을 알게 된다. 이어서, 이 2개의 store 중 최근의 것(V2)이 ID 10의 V1으로 포워딩될 수 있다. SA(V2) #11이 이제 디스패치된다. V1 LA 및 V2 SA가 실제 에이지를 갖지 않기 때문에, 그들의 VID가 비교를 위해 사용되고, 어떤 포워딩도 검출되지 않는다. 할당 포인터는 이제 16으로 이동한다. ID 16의 V4 SA가 이제 디스패치되고, 이는 ID 10의 V1 LA와 비교되며, V1 LA가 실제 에이지를 갖지만 V4 SA는 그렇지 않기 때문

에, V4 SA가 V1 LA보다 더 늦다. 따라서, 이 store로부터 이 보다 빠른 load로의 어떤 포워딩도 가능하지 않다.

- [0150] 도 7은 본 발명의 하나의 실시예에 따른, 프로세스(400)의 규칙들이 프로세서의 load 큐 및 store 큐 자원들로 구현되는 방식을 나타낸 다른 다이어그램을 도시한 것이다. 도 7의 실시예에서, 할당 포인터가 초기에 3이었던 경우를 생각해보자. V1 SA 및 V2 SA가 디스패치되고 store Q 엔트리 #4 및 #5에 할당되었다. 할당 포인터는 이제 6으로 이동하고, V4 SA가 디스패치된다. V1 SA 및 V2 SA 둘 다는 실제 에이지가 4 및 5이다.
- [0151] 할당 포인터는 이제 11로 이동한다. V3 LA는 실제 에이지가 7이다. V1 SA #10 V2 SA #11이 디스패치된다. V3 LA가 디스패치되고, 이는 그의 주소를 store Q 엔트리들과 비교하여 V1 SA, V2 SA 및 V4 SA 및 V2 SA #11에 걸쳐 일치를 찾아낸다. V3 LA가 실제 에이지가 7이기 때문에, 이는 그의 실제 에이지를 V2 SA에 속하는 그에 가장 가까운 store 에이지(에이지 5임)와 비교하며, 따라서 그 load는 이 store로부터 포워딩할 것이고 load Q 에 그와 같이 표시될 것이다.
- [0152] 도 8은 본 발명의 하나의 실시예에 따른, store가 load 이후에 디스패치되는 디스패치 기능의 개요의 프로세스 (800)의 플로우차트를 도시한 것이다.
- [0153] 프로세스(800)는 store 명령어가 SA와 SD로 분할되는 단계(801)에서 시작한다. 앞서 기술한 바와 같이, 분할된 SA와 방금 할당된 SD 간에 VID 테이블에서의 일치가 없는 경우, SA 명령어는 듀얼 디스패치를 가능하게 하기 위해 SD 명령어와 함께 시퀀틱스를 유지한다. 단계(802)에서, SA가 보다 빠른 기계 가시 프로그램 순서(machine visible program order)로 재정렬되고, 그 SA가 원래의 SD 프로그램 순서를 유지하도록 VID 테이블을 사용하여 추적된다. 단계(803)에서, SA의 디스패치 시에, SA에 대한 주소 일치가 있는지 load 큐 내의 모든 load들에 대해 검사가 행해진다.
- [0154] 단계(804)에서, 주소 일치 시에, load들 및 SA의 VID 번호들을 사용하여 또는 load들 및 store들의 실제 에이지 들을 사용하여, 일치하는 load들의 프로그램 순서가 SA의 프로그램 순서와 비교된다. 이것은 도 3의 논의에서 앞서 도표로 나타내었던 프로세스이다. store는 실제 에이지를 갖지만 load는 그렇지 않은 경우, store가 load 보다 더 빠르다. load는 실제 에이지를 갖지만 store는 그렇지 않은 경우, load가 store보다 더 빠르다. load 또는 store 중 어느 하나가 실제 에이지를 가지는 경우, 어느 것이 더 빠른지를 알아내기 위해 가상 식별자 (VID)가 사용될 수 있다. load 및 store 둘 다가 실제 에이지들을 가지는 경우, 어느 것이 더 빠른지를 알아내 기 위해 실제 에이지가 사용된다. 앞서 기술한 바와 같이, VID 번호는 원래의 프로그램 순서 및 재정렬된 SA 및 LA의 추적을 가능하게 한다. VID 테이블 내의 엔트리들은 대응하는 SD 및 LD가, 기계 자원들이 할당될 때, SA 및 LA에 할당되었던 기계 자원들과 연관될 수 있게 한다.
- [0155] 단계(805)에서, 프로그램 순서에서 보다 늦은 load들에 대해, store는 load들이 다른 store들에 의해 포워딩되 었는지를 알아보기 위해 검사를 할 것이다. 단계(806)에서, 그러한 경우, store는 이 load로 이전에 포워딩되 었던 store가 프로그램 순서에서 그 자신보다 더 빠른지를 알아보기 위해 그 store의 스탬프를 검사한다. 단계 (807)에서, 그러한 경우, store는 이 load로 이전에 포워딩되었던 store가 프로그램 순서에서 그 자신보다 더 빠른지를 알아보기 위해 그 store의 스탬프를 검사한다. 단계(808)에서, 그렇지 않은 경우, store는 이 load로 포워딩되지 않는다.
- [0156] 도 9는 본 발명의 하나의 실시예에 따른, load가 store 이후에 디스패치되는 디스패치 기능의 개요의 프로세스 (900)의 플로우차트를 도시한 것이다.
- [0157] 단계(901)에서, load 명령어가 앞서 기술한 방식으로 LA 및 LD로 분할된다. 단계(902)에서, LA는 앞서 기술한 바와 같이 보다 빠른 기계 가시 프로그램 순서로 재정렬되고 VID 테이블을 사용하여 추적된다. 단계(903)에서, LA가 load에 대한 주소 일치가 있는지 store 큐 내의 모든 store들에 대해 검사된다.
- [0158] 단계(904)에서, 주소 일치 시에, load 및 store의 VID 번호들을 사용하여 또는 load 및 store의 실제 에이지들 을 사용하여, 일치하는 load의 프로그램 순서를 store의 프로그램 순서와 비교한다. 이것은 도 3의 논의에서 앞서 도표로 나타내었던 프로세스이다. store는 실제 에이지를 갖지만 load는 그렇지 않은 경우, store가 load 보다 더 빠르다. load는 실제 에이지를 갖지만 store는 그렇지 않은 경우, load가 store보다 더 빠르다. load 또는 store 중 어느 하나가 실제 에이지를 가지는 경우, 어느 것이 더 빠른지를 알아내기 위해 가상 식별자 (VID)가 사용될 수 있다. load 및 store 둘 다가 실제 에이지들을 가지는 경우, 어느 것이 더 빠른지를 알아내 기 위해 실제 에이지가 사용된다. 앞서 기술한 바와 같이, VID 번호는 원래의 프로그램 순서 및 재정렬된 SA 및 LA의 추적을 가능하게 한다. 이어서, 단계(905)에서, load는 프로그램 순서에서 그 자신의 프로그램 순서에

가장 가까운 store로부터의 데이터를 소비(consume)한다.

- [0159] 도 10은 본 발명의 하나의 실시예에 따른, 통합된 load 큐의 다이어그램을 도시한 것이다. 가상 load/store 큐의 목적은 프로세서가 그의 load/store 큐의 실제의 물리적 크기를 사용하여 수용될 수 있는 것보다 더 많은 load들/store들을 기계에 할당할 수 있게 하는 것이다. 그 보답으로, 이것은 프로세서가 프로세서의 그의 load/store 큐의 물리적 크기 제한을 넘어 load/store 이외의 다른 명령어들을 할당할 수 있게 한다. load들/store들의 일부가 여전히 load/store 큐에 공간을 갖지 않더라도 이 다른 명령어들은 여전히 디스패치되고 실행될 수 있다.
- [0160] load들이 load 큐로부터 회수될 때, load 디스패치 윈도우는 시퀀스에서의 후속하는 명령어들로 이동하고, load 큐로부터 회수된 load들의 수와 동등한 디스패치를 위해 고려될 더 많은 할당된 load들을 포함할 것이다. 이 다이어그램에서, load 디스패치 윈도우는 좌에서 우로 이동할 것이다.
- [0161] 하나의 실시예에서, load 디스패치 윈도우는 load 큐에 있는 엔트리들의 수와 같은 수의 load들을 항상 포함할 것이다. load 디스패치 윈도우 밖에서 load들이 언제라도 디스패치될 수 없다. load들 이외의 스케줄러 윈도우 내의 다른 명령어들(예컨대, Sub, Add 등)이 디스패치될 수 있다. load 디스패치 윈도우 내의 모든 load들은 준비되어 있을 때 언제라도 디스패치될 수 있다.
- [0162] 도 11은 본 발명의 하나의 실시예에 따른, 슬라이딩 load 디스패치 윈도우를 보여주는 통합된 load 큐를 도시한 것이다. 도 11은 도 10과 비교하여 시간상 후속하는 인스턴스를 나타낸 것이다. load들이 load 큐로부터 회수될 때, load 디스패치 윈도우는 시퀀스에서의 후속하는 명령어들로 이동하고, load 큐로부터 회수된 load들의 수와 동등한 디스패치를 위해 고려될 더 많은 할당된 load들을 포함할 것이다. load 디스패치 윈도우는 load 큐에 있는 엔트리들의 수와 같은 수의 load들을 항상 포함할 것이다. load 디스패치 윈도우 밖에서 load들이 언제라도 디스패치될 수 없다. load들 이외의 스케줄러 윈도우 내의 다른 명령어들(예컨대, Sub, Add 등)이 디스패치될 수 있다. load 디스패치 윈도우 내의 모든 load들은 준비되어 있을 때 언제라도 디스패치될 수 있다. 이와 같이, 이 방식에 의해 획득되는 하나의 이점은 load 또는 store 큐 용량이 초과되는 경우 스케줄러에 할당하는 일이 정지되지 않고, 그 대신에 load 또는 store 큐 용량이 초과되더라도 load 및 store를 비롯한 명령어들을 스케줄러에 계속 할당하고, load 및 store 동적 윈도우는 load 또는 store 큐의 용량 밖의 load 또는 store가 디스패치되지 않도록 보장할 것이라는 것이다.
- [0163] 도 12는 본 발명의 하나의 실시예에 따른, 분산된 load 큐를 도시한 것이다. 도 12의 실시예의 목적은 단일의 프로그램/스레드 순차적 시맨틱스를 유지하지만 여전히 다수의 코어들/메모리 프래그먼트(memory fragment)들에 걸친 load들 및 store들의 비순차 디스패치를 가능하게 하는 분산된 load 큐 및 분산된 store 큐를 구현하는 것이다.
- [0164] 도 12의 다이어그램은 교착 상태(deadlocks)를 피하기 위한 load 큐 확장 해결책을 나타낸 것이다. load/store 큐의 확장이 생성되고 교착 상태를 야기한 load/store의 지점으로부터(그 지점으로부터 전방으로) load/store 큐가 이용가능한 자유로운 엔트리들을 가질 때까지 교착 상태의 load들/store들을 프로그램 순서로 그 확장 큐에 할당하기 위해 사용된다. 도 12의 시나리오에서, LD 3 load는 SD에 의존하고, SD는 차례로 load_Q B가 채워져 있기 때문에 디스패치될 수 없는 (load_Q B에 매핑되는 주소를 가지는) LD 2에 의존한다. 이 교착 상태 시나리오에서, 교착 상태의 검출시에, LD 1 및 LD 2는 디스패치되고 순차적으로 연달아 예비 부분 B로 회수되도록 허용된다. 분산된 load/store 큐에 대한 보수적 정책(conservative policy)은 각각의 load/store에 대해 각각의 load/store 분산된 큐에 엔트리를 예비하는 것이다. 이 도면에서, 각각의 할당된 load는 load_Q A에 하나의 엔트리를 예비하고 load_Q B에 다른 엔트리를 예비할 필요가 있다.
- [0165] 유의할 점은, 분산된 load/store 큐에서, 할당 시에 그들의 주소가 알려져 있지 않다는 점에서 할당된 load/store와 관련한 문제가 있다는 것이다. 이 때문에, 주어진 load 또는 store가 분산된 큐들 중 어느 것을 차지할 것인지가 비순차 디스패치 시에만 알려진다.
- [0166] 본 발명의 실시예들은 비순차 디스패치에서의 교착 상태를 피하기 위해 분산된 load/store 큐에 대한 3개의 상이한 해결책을 이용할 수 있다:
- [0167] 1. 교착 상태에 빠진(load/store 버퍼로 디스패치할 공간을 갖지 않는) 가장 빠른 load/store에서 예측 착오 및 플러시를 야기하고 어떤 기간 동안 순차적으로 또는 보수적인 할당에 의해 load/store를 디스패치하기 시작하며, 여기서 각각의 load/store는 모든 분산된 큐들에서의 공간을 할당한다. 그 load/store의 주소가 (디스패치 시에) 알려지고, 따라서 그 load/store를 수신하게 될 특정의 load 큐가 알려지면, 이는 다른 큐들에 있

는 예약된 공간들을 할당 해제할 수 있다.

- [0168] 2. load/store 큐의 확장이 생성되고 교착 상태를 야기한 load/store의 지점으로부터 교착 상태의 load들/store들을 프로그램 순서로 그 확장 큐에 할당하기 위해 사용된다(도 9).
- [0169] 3. 동적 디스패치 윈도우 크기 조정, 여기서 연속성 윈도우 밖의 비디스패치된(un-dispatched) load들의 합이 그 특정 load 큐 내의 자유로운 비예약된 공간들의 수보다 작거나 같아야만 한다(예를 들어, 도 11).
- [0170] 도 13은 본 발명의 하나의 실시예에 따른, 순차 연속성 윈도우를 가지는 분산된 load 큐를 도시한 것이다. 동적 load 디스패치 윈도우 크기 조정은 연속성 윈도우 밖의 비디스패치된 load들의 합이 그 특정 load 큐 내의 자유로운 비예약된 공간들의 수보다 작거나 같아야만 하도록 결정된다. 각각의 load 큐는 여기서 나타난 바와 같이 그 각자의 디스패치 윈도우를 사용하여 그의 엔트리들을 추적할 것이다. 임의의 때에 각각의 load 큐에 대한 동적 윈도우 크기 = 그 큐의 물리적 크기 + 가상 엔트리들이고(이 경우에, $6 + 4 = 10$), 따라서 이 경우에 윈도우 크기는 단지 10개의 load들만을 커버해야 한다. 유의할 점은, 다른 큐들에 대한 load들이 카운트되지 않는다는 것이다(예컨대, LD 4).
- [0171] 예비의 예약률(booking ratio)은 3이다. 예약률은 예비된 공간들 각각을 위해 경쟁하는 순차 load들의 수이다. 이 예에서, (순차 연속성 윈도우를 좌에서 우로 스캔할 때) 처음 2개의 순차 비디스패치된 load들만이 예비 부분으로 디스패치될 수 있다(큐의 2개의 엔트리가 예비에 할당된 것으로 가정함). 따라서, 가상 엔트리들의 수 = (예약률 - 1) * 예비 엔트리들의 수 = $(3-1)*2=4$ 이다.
- [0172] 순차 연속성 윈도우 크기 조정과 관련하여, load 큐 내의 엔트리(포착된 공간)로 디스패치되지 않은 임의의 때의 load들의 수(가장 오래된 것부터 가장 신생의 것으로 카운트함) + 예비 공간으로 디스패치된 load들의 수가 (예약률 * 예비 엔트리들의 수)보다 작거나 같아야만 한다. 이 경우에, load들의 수가 3보다 작거나 같아야만 한다. 예약률은 예비된 공간의 인정된(점유 대 예약) 비율이 얼마인지를 결정하는 설계 구성가능 성능 메트릭이다. 이것은 가장 빠른 비디스패치된 load들이 예비된 엔트리들 밖에서 디스패치할 큐 공간을 찾을 수 없는 경우에 행해진다. 이러한 경우에, 가장 빠른(가장 오래된) load로부터 시작하여 그 load들은 예비된 공간을 위해 경쟁할 것이고, 예약률은 각각의 예비된 엔트리를 차지하기 위해 몇개의 load들이 기다릴지를 결정하며, 예비된 엔트리들은 항상 가장 오래된 비디스패치된 load에 먼저 할당되고, 그 load가 회수되면, 그 다음 가장 오래된 load가 엔트리를 차지할 수 있다(예약률은 가장 오래된 디스패치된 것으로부터 시작하여 연달아 예비된 엔트리들을 차지하는 그 load들의 수를 결정한다).
- [0173] 유의할 점은, 하나의 실시예에서, 각각의 큐의 순차 연속성 윈도우로부터의 load들은, (순서상 가장 오래된 load로부터 시작하는) 그 큐의 비예약된 부분에 남아 있는 공간이 없을 때, 그 큐의 예비된 공간으로 디스패치될 수 있다는 것이다. 또한 유의할 점은, 하나의 실시예에서, 어느 한 큐의 순차 연속성 윈도우 밖의 그리고 그 큐의 동적 디스패치 윈도우 내의 load들은 그 큐의 예비된 부분으로 디스패치될 수 없다는 것이다.
- [0174] 또한 유의할 점은, 큐의 비예약된 부분에 공간이 있는 한, 그 큐의 동적 디스패치 윈도우 전체 내의 임의의 load는 분산된 큐들 중 임의의 것의 비예약된 부분의 임의의 엔트리로 비순차적으로 디스패치될 수 있다는 것이다. 어느 한 큐의 순차 연속성 윈도우 및 동적 디스패치 윈도우 둘 다의 크기가 각각의 load 디스패치 또는 회수 후에 앞서 제공된 식들에 나타난 그들의 크기 제한들을 반영하기 위해 각각의 사이클마다 조절된다.
- [0175] 도 14는 본 발명의 하나의 실시예에 따른, 멀티코어 프로세서에 대한 프래그먼트화된 메모리 서브시스템의 다이어그램을 도시한 것이다. 도 13은 스레드들 간의 및/또는 load들 및 store들 간의 동기화 방식의 포괄적인 방식 및 구현을 개괄적으로 나타낸 것이다. 이 방식은 load/store 아키텍처들에 걸쳐 및/또는 메모리 참조들 및/또는 스레드들의 메모리 액세스들에 걸쳐 메모리 참조들의 동기화 및 명확화(disambiguation)를 위한 바람직한 방법을 기술하고 있다. 도 14에서, 레지스터 파일들의 다수의 세그먼트들(주소 및/또는 데이터 레지스터들)이 실행 유닛들, 주소 계산 유닛들, 및 레벨 1 캐시들 및/또는 load store 버퍼들 및 레벨 2 캐시들의 프래그먼트들 그리고 주소 레지스터 상호연결부들(1200) 및 주소 계산 유닛 상호연결부들(1201)과 함께 도시되어 있다. 이 프래그먼트화된 요소들은 그의 집중된 자원들을 몇개의 엔진들로 프래그먼트화 및 분산시키는 것에 의해 하나의 코어/프로세서 내에 구성될 수 있거나, 멀티 코어/멀티 프로세서 구성들에서 상이한 코어들/프로세서들의 요소들로부터 구성될 수 있다. 그 중 하나의 프래그먼트(1211)가 도면에서 프래그먼트 번호 1로 도시되어 있고; 프래그먼트들은 큰 수로(일반적으로 도면에 도시된 바와 같이, N개의 프래그먼트들로) 스케일링될 수 있다.
- [0176] 이 메커니즘은 또한 그 엔진들/코어들/프로세서들 간에 메모리 아키텍처에 대한 일관성 방식으로 역할한다.

이 방식은 하나의 프래그먼트/코어/프로세서에서의 주소 계산 유닛들 중의 하나로부터의 주소 요청에 의해 시작된다. 예를 들어, 주소가 프래그먼트 1(예컨대, 1211)에 의해 요청되는 것으로 가정하자. 그는 그 자신의 프래그먼트에 속하는 주소 레지스터들을 사용하여 및/또는 주소 상호연결 버스(1200)를 사용하여 다른 프래그먼트들에 걸쳐 있는 레지스터들로부터 그의 주소를 획득하고 계산할 수 있다. 주소를 계산한 후에, 그는 캐시 및 메모리에 액세스하는 데 사용되는 32 비트 주소 또는 64 비트 주소의 참조 주소를 생성한다. 이 주소는 보통 태그 필드(tag field)와 세트 및 라인 필드(set and line field)로 프래그먼트화된다. 이 특징의 프래그먼트/엔진/코어는 주소를 그의 load store 버퍼 및/또는 L1 및/또는 L2 주소 어레이들(1202)에 저장할 것이고, 이와 동시에, 압축 기법을 사용함으로써 태그의 압축된 버전(주소의 원래의 태그 필드보다 더 적은 수의 비트들을 가짐)을 생성할 것이다.

[0177] 더욱이, 상이한 프래그먼트들/엔진들/코어들/프로세서들이 세트 필드 또는 세트 필드의 서브셋을 주소가 어느 프래그먼트/코어/프로세서에 유지되는지를 식별해주는 인덱스로서 사용할 것이다. 주소 세트 필드 비트에 의한 프래그먼트의 이러한 인덱싱은 특징의 프래그먼트/코어/엔진에서의 주소의 소유권의 배타성을 보장해주지만, 그 주소에 대응하는 메모리 데이터가 다른 또는 다수의 다른 프래그먼트들/엔진들/코어들/프로세서들에 살아 있을 수 있다. 주소 CAM/태그 어레이(1202/1206)가 각각의 프래그먼트에서 데이터 어레이(1207)에 결합되어 있는 것으로 도시되어 있지만, 이들이 단지 배치 및 레이아웃의 물리적 근접성으로 또는 심지어 둘 다가 특징의 엔진/코어/프로세서에 속해 있다는 사실에 의해 결합될 수 있지만, 하나의 프래그먼트 내부에서 주소 어레이에 유지되는 주소들과 데이터 어레이 내의 데이터 간에 어떤 관계도 없다.

[0178] 도 15는 본 발명의 실시예들에 의해 load 및 store가 어떻게 처리되는지의 다이어그램을 도시한 것이다. 도 15에 도시된 바와 같이, 각각의 프래그먼트는 그의 load store 버퍼 및 store 회수 버퍼(store retirement buffer)와 연관되어 있다. 임의의 주어진 프래그먼트에 대해, 그 프래그먼트 또는 다른 프래그먼트와 연관된 주소 범위를 지정하는 load 및 store는 처리를 위해 그 프래그먼트의 load store 버퍼로 송신된다. 유의할 점은, 코어가 명령어를 비순차적으로 실행하기 때문에 그들이 비순차적으로 도착할 수 있다는 것이다. 각각의 코어 내에서, 코어는 그 자신의 레지스터 파일 뿐만 아니라 다른 코어들의 레지스터 파일들 각각에도 액세스한다.

[0179] 본 발명의 실시예들은 분산된 load store 정렬 시스템(distributed load store ordering system)을 구현한다. 이 시스템은 다수의 프래그먼트들에 걸쳐 분산되어 있다. 하나의 프래그먼트 내에서, 로컬 데이터 의존성 검사(local data dependency checking)가 그 프래그먼트에 의해 수행된다. 이러한 이유는 프래그먼트가 그 특징의 프래그먼트의 store 회수 버퍼 내에만 로드 및 저장하기 때문이다. 이것은 데이터 일관성을 유지하기 위해 다른 프래그먼트들을 고려해야만 하는 필요성을 제한한다. 이러한 방식으로, 하나의 프래그먼트 내에서의 데이터 의존성이 로컬적으로 시행된다.

[0180] 데이터 일관성(data consistency)과 관련하여, store 디스패치 게이트는 엄격한 프로그램 순서 메모리 일관성 규칙(in-program order memory consistency rule)에 따라 store 회수를 시행한다. store는 load store 버퍼에 비순차적으로 도착한다. load도 역시 load store 버퍼에 비순차적으로 도착한다. 이와 동시에, 비순차 load 및 store는 처리를 위해 store 회수 버퍼로 포워딩된다. 유의할 점은, store가 주어진 프래그먼트 내에서 순차적으로 회수되지만, 이들이 store 디스패치 게이트로 갈 때, 이들이 다수의 프래그먼트들로부터 비순차적일 수 있다는 것이다. store 디스패치 게이트는 store가 store 회수 버퍼들에 걸쳐 비순차적으로 존재할 수 있더라도 그리고 버퍼들이 store를 store 디스패치 게이트로 다른 버퍼의 store에 대해 비순차적으로 포워딩할 수 있더라도, 디스패치 게이트가 이들이 프래그먼트 메모리로 엄격히 순차적으로 포워딩될 것을 보장하도록 보장하는 정책을 시행한다. 이러한 이유는 store 디스패치 게이트가 store 회수에 대한 전역적 뷰를 가지고 있고 단지 store가 모든 프래그먼트들에 걸쳐(예컨대, 전역적으로) 순차적으로 메모리의 전역적 가시 측면으로 갈 수 있게 하기 때문이다. 이러한 방식으로, store 디스패치 게이트는 store가 궁극적으로 모든 프래그먼트들에 걸쳐 순차적으로 메모리로 복귀하도록 보장하는 전역적 관찰자로서 기능한다.

[0181] 도 16은 본 발명의 하나의 실시예에 따른, store 필터링 알고리즘의 다이어그램을 도시한 것이다. 도 16의 실시예의 목적은 모든 store들이 load 큐 내의 모든 엔트리들에 대해 검사할 필요가 있는 것을 방지하기 위해 store를 필터링하는 것이다.

[0182] store는 일관성을 유지하기 위해 주소 일치여부 있는지 캐시를 스누핑한다. 스레드/코어 X load가 캐시 라인으로부터 판독하는 경우, 그 load는 자신이 데이터를 로드했던 캐시 라인의 부분에 표시를 한다. 다른 스레드/코어 Y store가 캐시를 스누핑할 시에, 임의의 이러한 store가 그 캐시 라인 부분과 중복하는 경우, 스레드/코어 X의 그 load에 대해 예측 착오가 야기된다.

- [0183] 이 스누핑들을 필터링하기 위한 하나의 해결책은 load 큐 엔트리들의 참조들을 추적하는 것이다. 이 경우에, store는 load 큐를 스누핑할 필요가 없다. store가 액세스 마스크와의 일치성을 가지는 경우, 참조 추적기(reference tracker)로부터 획득된 것과 같은 그 load 큐 엔트리는 그 load 엔트리로 하여금 예측 착오를 하게 할 것이다.
- [0184] (참조 추적기가 없는) 다른 해결책에서, store가 액세스 마스크와의 일치성을 가지는 경우, 그 store 주소는 load 큐 엔트리들을 스누핑할 것이고 일치된 load 엔트리로 하여금 예측 착오를 하게 할 것이다.
- [0185] 양 해결책에서, load가 캐시 라인으로부터 판독하고 있다면, 그는 각자의 액세스 마스크 비트를 세트시킨다. 그 load가 회수될 때, 그는 그 비트를 리셋시킨다.
- [0186] 도 17은 본 발명의 하나의 실시예에 따른, 메모리로부터 순차적으로 판독하는 load를 구성하는 메모리 일관성 모델에서의 비순차 load를 갖는 세마포어 구현을 도시한 것이다. 본 명세서에서 사용되는 바와 같이, 세마포어라는 용어는 다수의 스레드/코어의 공통의 자원들에 대한 액세스 제어를 제공하는 데이터 구성체(data construct)를 말한다.
- [0187] 도 17의 실시예에서, 액세스 마스크는 다수의 스레드들/코어들에 의한 메모리 자원들에의 액세스를 제어하는 데 사용된다. 액세스 마스크는 캐시 라인의 어느 워드들이 보유 중인 load를 가지는지를 추적하는 것에 의해 기능한다. 비순차 load는 캐시 라인의 워드에 액세스할 때 마스크 비트를 세트시키고, 그 load가 회수될 때 마스크 비트를 클리어시킨다. 마스크 비트가 세트되어 있는 동안 다른 스레드/코어로부터의 store가 그 워드에 기입하는 경우, 이는 (예컨대, 추적기를 통해) 그 load에 대응하는 load 큐 엔트리가 예측 착오/플러시되거나 그의 의존적 명령어로 재시도될 것으로 신호할 것이다. 액세스 마스크는 또한 스레드/코어를 추적한다.
- [0188] 이러한 방식으로, 액세스 마스크는 메모리 일관성 규칙이 정확하게 구현되도록 보장한다. 메모리 일관성 규칙은 이 세마포어가 2개의 코어/스레드에 걸쳐 동작하도록 store가 메모리를 순차적으로 업데이트하고 load가 메모리로부터 순차적으로 판독하는 것을 좌우한다(dictate). 이와 같이, 코어 1 및 코어 2에 의해 실행되는 코드는, 이들 둘 다 메모리 장소 "플래그" 및 "데이터"에 액세스하는 경우, 정확하게 실행될 것이다.
- [0189] 도 18은 본 발명의 하나의 실시예에 따른, 로크 기반 모델 및 트랜잭션 기반 모델 둘 다의 사용에 의해 메모리로부터 순차적으로 판독하는 load를 구성하는 메모리 일관성 모델의 비순차 load를 도시한 것이다.
- [0190] 앞서 기술한 바와 같이, 메모리 일관성 규칙은, 2개의 코어/스레드가 적절히 통신하도록, store가 메모리를 순차적으로 업데이트하고 load가 메모리를 순차적으로 참조하는 것을 좌우한다. 도 18의 하부 우측에, 2개의 코어(코어 1 및 코어 2)가 도시되어 있다. 2개의 메모리 자원(플래그 및 데이터)이 사용되고, 통신을 구현하며, 코어 1과 코어 2 사이에서 정확하게 데이터를 공유한다. 예를 들어, 코어 1 내의 코드로 나타낸 바와 같이, 코어 1이 데이터를 코어 2로 전달하고자 할 때, 코어 1은 데이터를 저장하고 이어서 플래그를 세트시킬 것이다. 코어 2 내의 코드로 나타낸 바와 같이, 코어 2는 플래그를 로드하고 플래그가 1인지를 검사할 것이다. 플래그가 1이 아닌 경우, 코어 2는 뒤로 점프하고 플래그가 1일 때까지 플래그를 계속 검사할 것이다. 그 시점에서, 코어 2는 데이터를 로드할 것이다.
- [0191] load 및 store가 비순차적으로 실행되는 비순차 아키텍처에서, 2개의 엔티티(예컨대, 코어 1 및 코어 2)가 순차 메모리 일관성 시맨틱스를 유지하도록 보장하기 위해 로크 기반 메모리 일관성 모델이 사용될 수 있다. 이것은 액세스 마스크, 스레드 ID 레지스터, 및 추적기 레지스터의 사용을 통해 나타내어져 있다. 코드의 중요 섹션 내에서의 임의의 load의 대응하는 액세스 마스크 비트를 세트시키는 것에 의해 로크가 세트된다. 다른 스레드/코어로부터 그 캐시 라인 워드로의 임의의 액세스가 일어나는 경우, 로크는 그 액세스를 방지할 것이다. 하나의 실시예에서, 이것은 액세스를 미스(miss)로 취급하는 것에 의해 구현될 수 있다. 로크가 클리어될 때, 그 워드에의 액세스가 허용된다.
- [0192] 다른 대안으로서, 순차 메모리 일관성 시맨틱스를 유지하기 위해 트랜잭션 기반 방법이 사용될 수 있다. 트랜잭션 기반 방법에서, 트랜잭션 내의 임의의 load의 대응하는 액세스 마스크 비트를 세트시키는 것에 의해 원자성(atomicity)이 설정된다. 마스크 비트가 세트되어 있는 동안 다른 스레드/코어 또는 병렬 트랜잭션으로부터 그 캐시 라인 워드에의 임의의 액세스가 일어나는 경우, 이는 (예컨대, 추적기를 통해) 그 load에 대응하는 load 큐 엔트리가 예측 착오/플러시되거나 그의 의존적 명령어로 재시도될 것으로 신호할 것이다. 액세스 마스크는 또한 스레드/코어를 추적한다. 트랜잭션이 종료될 때 마스크 비트가 클리어될 것이다. 어느 스레드가 통합된 store 큐 엔트리의 어느 워드에 액세스하는지를 추적하기 위해 스레드 ID 레지스터가 사용된다.

- [0193] 도 19는 본 발명의 하나의 실시예에 따른, 멀티 코어 세그먼트화된 메모리 서브시스템의 복수의 코어들을 도시한 것이다. 이 실시예는 멀티 코어 세그먼트화된 메모리 서브시스템 내로부터의 load가 진행 중인 트랜잭션의 일부로서 표시되어 있는 워드에 액세스하는 것이 어떻게 방지될 것인지를 나타내고 있다(예컨대, 로크된 경우와 유사함).
- [0194] 유의할 점은, 이 멀티 코어 세그먼트화된 서브시스템이 공유 메모리 서브시스템을 갖는 외부 프로세서/코어/클러스터가 있는 보다 큰 클러스터의 일부라는 것이다. 이 경우에, 다른 외부 프로세서/코어/클러스터에 속하는 load들이 진행할 것이고 그 메모리 장소가 트랜잭션 액세스의 일부인 경우 관심을 갖지 않는 임의의 메모리 장소로부터 로드하는 것이 방지되지 않을 것이다. 그렇지만, 모든 load들은 트랜잭션의 일부인 장래의 store들을 통지하기 위해 액세스 마스크에 표시할 것이다.
- [0195] 다른 프로세서들로부터 오는 store를 스누핑하는 것은 그의 주소들을 마스크와 비교한다. store가 자신이 저장하려고 시도하고 있는 주소가 다른 스레드 load(트랜잭션의 일부인 load)로부터의 액세스 마스크에 표시되어 있는 경우, store는 그 load로 하여금 예측 착오되게 할 것이다. 그렇지 않은 경우, 그 load가 회수될 때 마크가 클리어될 것이다(예컨대, 그로써 트랜잭션을 완료함).
- [0196] 도 20은 본 발명의 하나의 실시예에 따른, store가 store 상위 서열에 기초하여 어느 한 스레드에서 데이터를 load로 포워딩할 수 있는 경우 비동기적 코어들이 통합된 store 큐에 액세스하는 다이어그램을 도시한 것이다.
- [0197] 앞서 기술한 바와 같이, 메모리 일관성 규칙은, 코어/스레드가 적절히 통신하도록, store가 메모리를 순차적으로 업데이트하고 load가 메모리로부터 순차적으로 관독하는 것을 좌우한다. 도 20의 하부 우측에, 2개의 코어(코어 1 및 코어 2)가 도시되어 있다. 2개의 코어는 비동기적이고 플래그 및 데이터 메모리 자원에 액세스하기 위해 각각의 코어 내에 나타내어진 코드를 실행한다.
- [0198] 도 20의 실시예에서, 통합된 store 큐는 그에 액세스할 수 있는 복수의 스레드들 중 임의의 것에 무관하다. 이 구현에서, 상이한 스레드들로부터의 store들은 한 세트의 알고리즘 규칙들을 따르는 것에 의해 순차 메모리 일관성 시맨틱스를 여전히 유지하면서 상이한 스레드들의 load들로 포워딩될 수 있다. 스레드들은 store 상위 서열에 기초하여 서로로부터 포워딩될 수 있다.
- [0199] store는 동일한 스레드에서 그 이전에 있는 모든 load들 및 store들이 실행되었을 때 상위(senior)이다. 다른 스레드로부터의 포워딩을 수신하는 스레드는 load/store를 독립적으로 회수할 수 없다. 스레드들이 포워딩을 수신하는 다른 스레드들이 예측 착오된 경우 그 스레드들은 조건부로 예측 착오되어야만 한다. 특정의 load는, 동일한 스레드 포워딩 store(same thread forwarding store)로부터 또는 동일한 스레드 내에서 그에게 포워딩하는 store가 없는 경우 상이한 스레드 상위 store(different thread senior store)로부터 포워딩할 수 있다.
- [0200] 도 20의 방법에서, 임의의 액세스의 대응하는 액세스 마스크 비트를 통합된 store 큐 엔트리에서의 워드 내의 바이트들로 설정하는 것에 의해 원자성이 설정된다. 마스크 비트가 세트되어 있는 동안 다른 스레드/코어 또는 병렬 트랜잭션으로부터 그 store 큐 엔트리 워드에의 임의의 액세스가 일어나는 경우, 이는 (예컨대, 추적기를 통해) 그 load에 대응하는 load 큐 엔트리가 예측 착오/플리시되거나 그의 의존적 명령어로 재시도될 것으로 신호할 것이다. 액세스 마스크는 또한 스레드/코어를 추적한다. 트랜잭션이 종료될 때 마스크 비트가 클리어될 것이다.
- [0201] 도 21은 본 발명의 하나의 실시예에 따른, store가 서열을 가지는 기능을 나타낸 다이어그램을 도시한 것이다. 도 21에 도시된 바와 같이, 특정의 load는 동일한 스레드 포워딩 store로부터 포워딩할 것이다. 스레드 내로부터의 포워딩이 없는 경우, 이는 상이한 스레드 상위 store로부터 포워딩될 수 있다. 다수의 코어/스레드가 공유 메모리에 액세스하고 있는 경우에 이 원칙이 기능한다. 이러한 경우에, store는 어느 한 스레드로부터 store 상위 서열에 기초하여, 그렇지만, 스레드 내로부터 특정의 load로의 포워딩이 없는 경우에만, 어느 한 스레드로부터의 load로 포워딩할 수 있다. store는 동일한 스레드에서 그 이전에 있는 모든 load들 및 store들이 실행되었을 때 상위이다.
- [0202] 그에 부가하여, 유의할 점은, 스레드가 load/store를 독립적으로 회수할 수 없다는 것이다. 스레드가 포워딩을 수신한 다른 스레드가 저장 예측 착오(store miss predict)되거나 플리시될 때 그 스레드는 로드 예측 착오(load miss predict)되어야만 한다.
- [0203] 도 21은 2개의 비동기적 코어들/스레드들(예컨대, 코어/스레드 1 및 코어/스레드 2) 간의 실행의 한 예시적인 스트림을 시각적으로 나타낸 것이다. 라인들(2101 내지 2105)은 store가 그의 서열에 기초하여 상이한 load로

포워딩하는 방식을 나타내고 있다. 서열이 store로부터 store로 어떻게 진행되는지를 나타내는 데 도움을 주기 위해, 0부터 14까지 진행함에 따라 상이한 실행 스테이지들을 나타내기 위해 숫자들이 각각의 명령어 옆에 열거되어 있다. 상세하게는, 유의할 점은, 라인(2103)에 의해 나타내어진 store가 동일한 스레드 내의 load로 포워딩하는 방식이 앞서 기술한 규칙들을 따른다는 것이다. 이와 같이, 앞서 기술한 바와 같이, 그 자신의 스레드 내로부터 포워딩하는 load는 임의의 인접한 스레드로부터 포워딩할 수 없다. 이것은 포워딩 라인들에 걸쳐 색상 십자 표시로 도시되어 있다.

[0204] 도 22는 비투기적인 (예컨대, 낮은 전력, 낮은 다이 면적 및 보다 적은 타이밍 임계성(timing criticality)을 나타내는) 본 발명의 하나의 실시예에 따른 비명확화된 비순차 load store 큐 회수 구현을 도시한 것이다.

[0205] store 회수/재정렬 버퍼(store retirement/reorder buffer)(SRB)는 2개의 구현(회수 구현 및 재정렬 구현)에서 동작할 수 있다.

[0206] 회수 구현에서, store는 원래의 프로그램 순서에서 더 빠른 store가 SRB의 상부에 있도록 store의 회수 시에 원래의 프로그램 순서로 store 큐로부터 SRB 내에 로드된다. 후속하는 load는 이어서 (예컨대, 주소 CAM을 사용하여) 주소 일치를 탐색하고, SRB/store 캐시 내의 일치하는 엔트리로부터 포워딩할 수 있다. 2개 이상의 주소 일치가 있는 경우에, 우선순위 인코더는 첫번째 것이 있는지 스캔하는 것에 의해 정확한 포워딩 엔트리를 위치 확인할 수 있다. 이것은 메모리로의 이동(trip)을 절감하고 기계가 전방으로 진행할 수 있게 한다. load가 디스패치되고 그것으로 포워딩되는 store가 SRB/store 캐시로 이미 회수된 경우, 그 load는 SRB/store 캐시로부터 포워딩되고 페어링 관계(pairing relationship)를 예측 테이블에 기록한다. load로 포워딩되는 store가 SRB/store 캐시로 회수되기 전에 그 load가 디스패치되는 경우를 검출하기 위해, load는 그가 그 자신의 주소에 표시하는 주소 마스크를 생성해야만 한다. 이것은 상이한 방식으로 구현될 수 있다(예컨대, 도 17의 실시예).

[0207] 앞서 논의한 바와 같이, 도 17은 캐시 라인의 어느 워드들이 보류 중인 load를 가지는지를 추적하는 것에 의해 기능하는 액세스 마스크를 기술하고 있다. 비순차 load는 캐시 라인의 워드에 액세스할 때 마스크를 세트시키고, 그 load가 회수될 때 마스크 비트를 클리어시킨다. 마스크 비트가 세트되어 있는 동안 동일한 스레드/코어로부터의 store가 그의 회수 시에 그 워드에 기입하는 것을 검출하는 경우, 이는 (추적기를 통해) 그 load에 대응하는 load 큐 엔트리가 예측 착오/플러시되거나 그의 의존적 명령어로 재시도될 것으로 신호할 것이다. 액세스 마스크는 또한 스레드/코어를 추적한다.

[0208] 도 22는 비순차 load 및 store를 명확하게 하기 위해 대응하는 하드웨어를 포함하지 않는다는 사실에서 비명확화 load store 큐이다. load 및 store는 기계 자원들이 허용할 때 비순차적으로 디스패치된다. 종래에는, 앞서 기술한 바와 같이(예컨대, 도 5 및 도 6), 정확한 store 큐 엔트리들이 요청하는 load 큐 엔트리들로 포워딩되도록 보장하기 위해 주소 일치 및 대응하는 명확화 하드웨어가 load 큐 및 store 큐 둘 다에서 사용된다. load 큐 및 store 큐의 내용은 외부 코어/스레드에 보이지 않는다.

[0209] 도 22에서, 디스패치된 load 및 store 주소는 store 큐 또는 load 큐에서의 엔트리들에 대해 명확화되지 않는다. load/store 큐는 이제 감소된 다이 면적, 전력 소비, 및 타이밍 요구사항을 갖는 간소화된 버퍼 구현이다. SRB는 명확화 기능을 수행할 것이다. SRB에서 주소 일치가 검출될 때, 그 일치는 명령어 시퀀스의 실행이 전방으로 나아갈 때 포워딩을 시행하기 위해 store-load 포워딩 예측 테이블(store to load forwarding prediction table)에 엔트리들을 채우는 데 사용된다.

[0210] load들이 디스패치될 때, 이들은 이들이 대응하는 store와 짝을 이루고 있는지를 알아보기 위해 예측 테이블을 검사한다. load가 짝을 이루고 있고 그 특정의 store가 이미 디스패치된 경우, load가 예측 테이블에 기록된 바와 같이 그 store 큐 엔트리 번호로부터 포워딩될 것이다. store가 아직 디스패치되지 않은 경우, load는 그의 load 큐 엔트리 번호를 예측 테이블에 등록할 것이고, store data가 포워딩되기를 기다리기 위해 load 큐에서 그 자체에 표시할 것이다. store가 나중에 디스패치될 때, 그는 load 큐 엔트리 번호를 획득하고 그 load로 포워딩하기 위해 예측 테이블을 검사한다.

[0211] load store 쌍에 대해 포워딩이 검출되면, 주소 일치가 검증되도록 PC 및 load store 쌍의 주소들이 기록된다. 주소가 일치하면, load는 store data가 디스패치될 때까지 디스패치되지 않을 것이고, load가 그로부터 포워딩되도록 표시될 것이다. load store 쌍들 간의 포워딩 관계에서의 신뢰 수준을 설정하기 위해 예측 임계치가 사용된다.

[0212] 도 23은 본 발명의 하나의 실시예에 따른, 비명확 비순차 load store 큐 재정렬 구현의 재정렬 구현을 도시한 것이다. 도 23은 또한 비투기적인 낮은 전력, 낮은 다이 면적, 및 보다 적은 타이밍 임계성을 나타낸다.

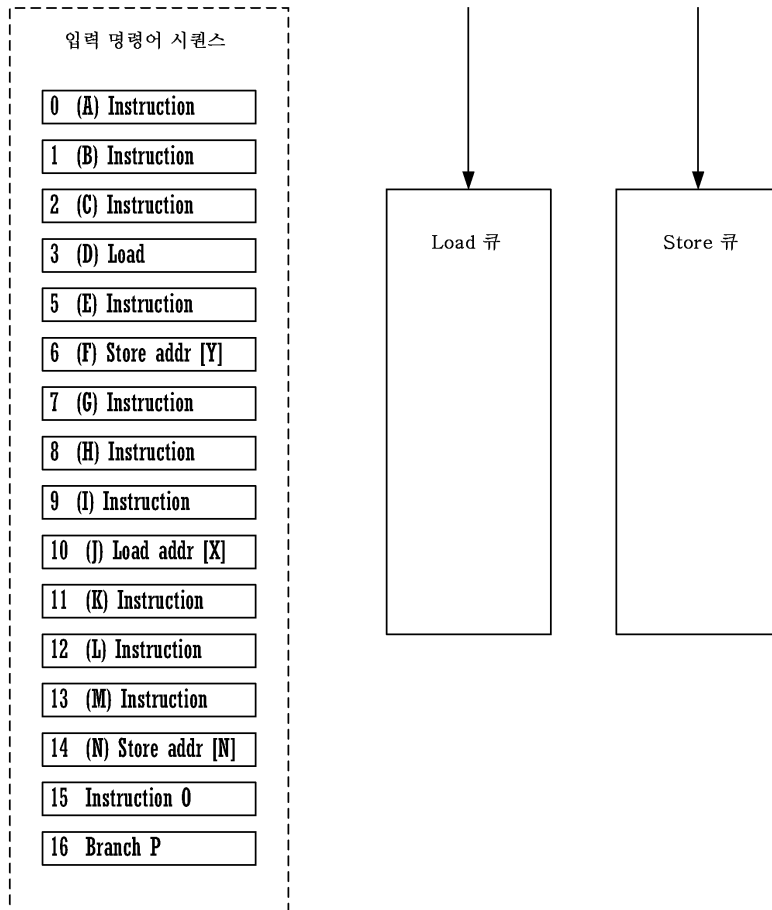
- [0213] store 회수/재정렬 버퍼(SRB)는 2개의 구현(회수 구현 및 재정렬 구현)에서 동작할 수 있다.
- [0214] 도 23의 재정렬 구현에서, (예컨대, 자원들이 허용할 때) store 주소가 store 큐로부터 비순차적으로 SRB 내로 로드된다. 각각의 store가 할당될 때, 그는 시퀀스 번호를 수신한다. SRB는 이어서 store가 원래의 프로그램 순서로 SRB에 존재하도록 그의 시퀀스 번호에 따라 store를 재정렬하는 것에 의해 기능한다. 프로그램 순서에서 보다 빠른 store는 SRB의 상부에 있다. 후속하는 load는 이어서 주소 일치 및 할당 에이지를 탐색한다(프로그램 순서의 시퀀스 번호는 load 및 store의 할당 시에 주어짐). load가 디스패치될 때, load는 SRB를 살펴보고, load가 아직 디스패치되지 않은 (그 자신의 시퀀스 번호와 비교하여) 보다 빠른 store를 보는 경우(주소 계산이 아직 없음), 2가지 해결책들 중 하나가 구현될 수 있다.
- [0215] 1. load가 디스패치되지 않고, load는 그 자체가 디스패치되기 전에 모든 보다 빠른 store들이 디스패치될 때까지 기다린다.
- [0216] 2. load가 디스패치되고 캐시의 액세스 마스크에서 그의 주소를 표시한다(도 17에 도시됨). 후속하는 store는 액세스 마스크를 검사하고 도 17에 도시된 것과 동일한 방법을 따른다.
- [0217] 유의할 점은, 우선순위 인코더는 정확한 포워딩 엔트리를 찾아내기 위해 앞서 기술한 바와 같이 기능한다는 것이다.
- [0218] 도 24는 본 발명의 하나의 실시예에 따른, 명령어 시퀀스(예컨대, 트레이스) 재정렬된 투기적 실행 구현을 도시한 것이다. 투기적 모드에서, store는 원래의 프로그램 순서에서 더 빠른 store가 SRB의 상부에 있도록 store의 회수 시에 원래의 프로그램 순서로 store 큐로부터 SRB 내로 이동된다. 후속하는 load는 이어서 (예컨대, 주소 CAM을 사용하여) 주소 일치를 탐색하고, SRB/store 캐시 내의 일치하는 엔트리로부터 포워딩할 수 있다. 2개 이상의 주소 일치가 있는 경우에, 우선순위 인코더는 첫번째 것을 스캔하는 것에 의해 정확한 포워딩 엔트리를 위치 확인할 수 있다. 이것은 기계가 전방으로 진행할 수 있게 한다. load가 디스패치되고(load가 SRB를 처음으로 검사할 때) 그것으로 포워딩되는 store가 SRB/store 캐시로 회수되는 경우, 그 load는 SRB/store 캐시로부터 포워딩되고 그것의 페어링 관계를 예측 테이블에 기록한다. load로 포워딩되는 store가 SRB/store 캐시로 회수되기 전에 그 load가 디스패치되는 경우를 검출하기 위해, load는 회수 시에 store 큐를 한번 더 검사할 것이다. load가 포워딩 store 일치를 찾아내는 경우, 이는 그 load에 대응하는 load 큐 엔트리가 예측 착오/플러시되거나 그의 의존적 명령어로 재시도되는 것으로 신호할 것이다. 포워딩 예측자는 이 포워딩 착오(miss-forwarding)로부터 학습할 것이다.
- [0219] 유의할 점은, 언급한 load를 포함하는 트레이스 내의 모든 명령어들이 트레이스 커밋 상태(trace commit state)에 도달할 때까지(예컨대, 모두가 비투기적으로 되고 트레이스가 전체로서 커밋할 준비가 될 때까지) SRB 내의 모든 store들이 외부 캐시/store 캐시의 구조적으로 보이는 상태(external cache/store cache architecturally visible state)로 커밋되지 않을 것이기 때문에(SRB 스토리지를 가시 메모리로 둘 것이기 때문에) load가 이전의 store에 대해 일치하는 주소가 있는지 SRB를 검사할 수 있을 것이라는 것이다.
- [0220] store 회수/재정렬 버퍼는 기능적으로 투기적 실행을 가능하게 한다. 투기적 결과가 알려질 때까지 투기적 실행의 결과가 store 회수/재정렬 버퍼에 저장될 수 있다. 투기적 결과가 구조적으로 보이지 않는다. 투기적 상태가 커밋되면, store는 store 캐시로 커밋될 수 있다. 상태를 커밋하기 전에, 재시도될 필요가 있는 임의의 예외 또는 load 및 store는 상태 커밋을 방지할 예외 또는 예측 착오를 신호할 것이다. store와 대응하는 load 간의 포워딩 예측 착오가 (예컨대, 로드 포워딩 착오 시점 등에서 기계를 플러시하는 예측 착오를 야기하는 것에 의해) 교정될 수 있다.
- [0221] SRB 기능에 대한 부가의 설명은 2012년 1월 27일자로 출원된, 발명의 명칭이 "HARDWARE ACCELERATION COMPONENTS FOR TRANSLATING GUEST INSTRUCTIONS TO NATIVE INSTRUCTIONS"인 Mohammad Abdallah의 미국 특허 출원 제13/360,024호(대리인 문서 번호 SMII-033)에서 찾아볼 수 있다.
- [0222] 도 25는 본 발명의 하나의 실시예에 따른, 한 예시적인 마이크로프로세서 파이프라인(2500)의 다이어그램을 도시한 것이다. 마이크로프로세서 파이프라인(2500)은, 앞서 기술한 바와 같이, 실행을 포함하는 명령어들을 식별하고 추출하는 프로세스의 기능을 구현하는 페치 모듈(2501)을 포함한다. 도 25의 실시예에서, 페치 모듈 다음에 디코드 모듈(2502), 할당 모듈(2503), 디스패치 모듈(2504), 실행 모듈(2505), 및 회수 모듈(2506)이 온다. 유의할 점은, 마이크로프로세서 파이프라인(2500)이 앞서 기술한 본 발명의 실시예들의 기능을 구현하는 파이프라인의 하나의 예에 불과하다는 것이다. 통상의 기술자라면 앞서 기술한 디코드 모듈의 기능을 포함하는 다른 마이크로프로세서 파이프라인들이 구현될 수 있다는 것을 잘 알 것이다.

[0223]

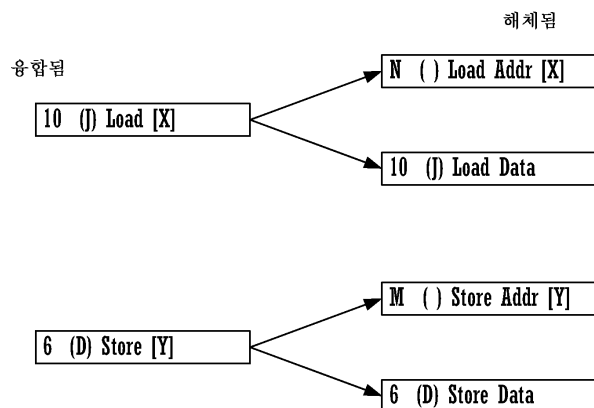
설명의 목적상, 이상의 설명은 전수적이거나 본 발명을 제한하려는 것이 아닌 구체적인 실시예들을 언급하고 있다. 이상의 기재 내용에 따라 많은 수정들 및 변형들이 가능하다. 다른 통상의 기술자가 그의 특정의 용도에 적합할 수 있는 본 발명 및 다양한 수정들을 갖는 본 발명의 다양한 실시예들을 가장 잘 이용할 수 있도록 본 발명의 원리 및 그의 실제 응용을 가장 잘 설명하기 위해 실시예들이 선택되고 기술되어 있다.

도면

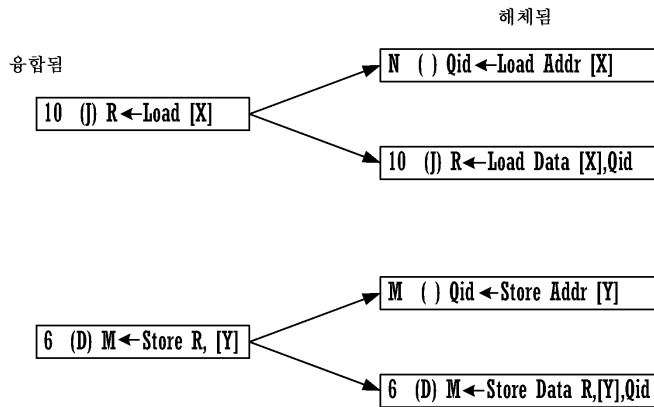
도면1



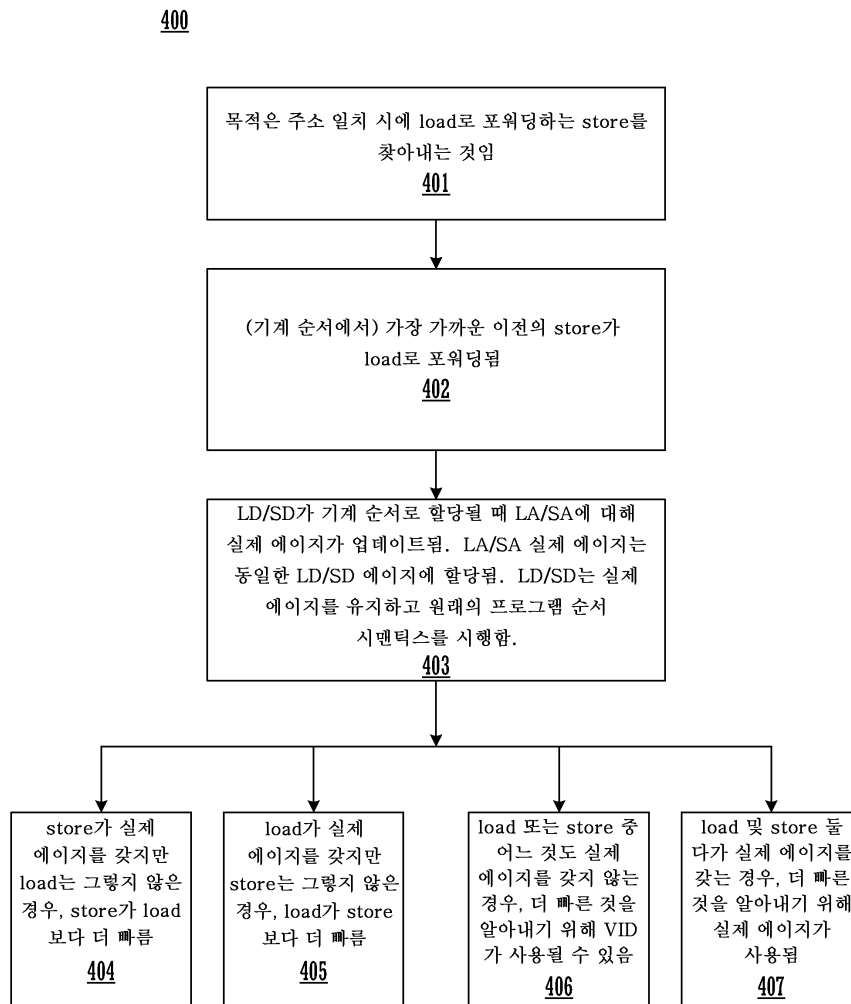
도면2



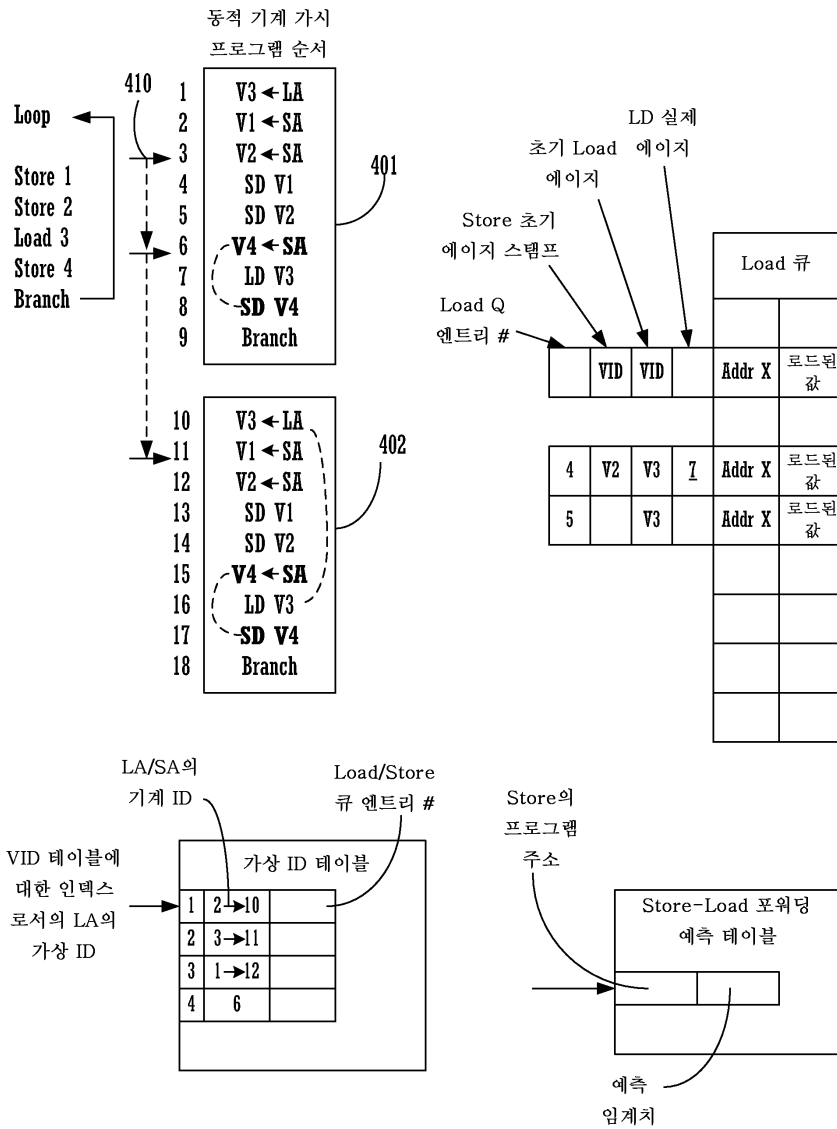
도면3



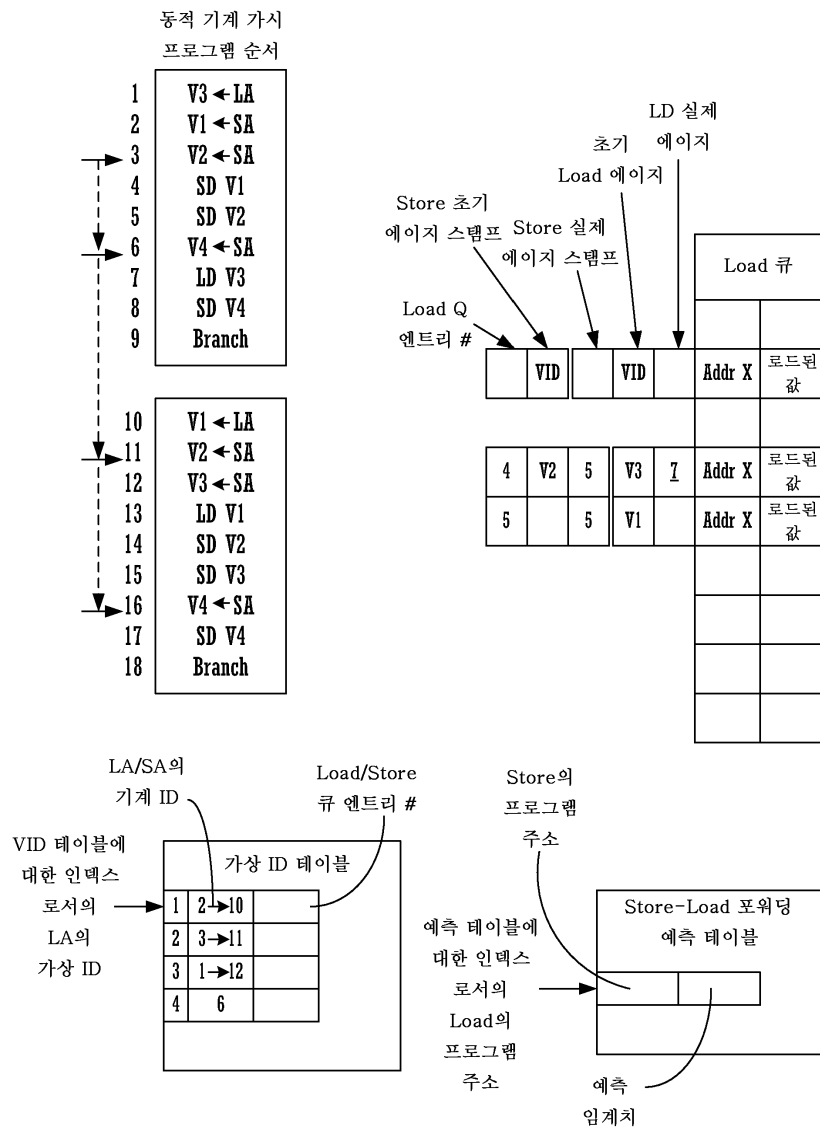
도면4



도면5



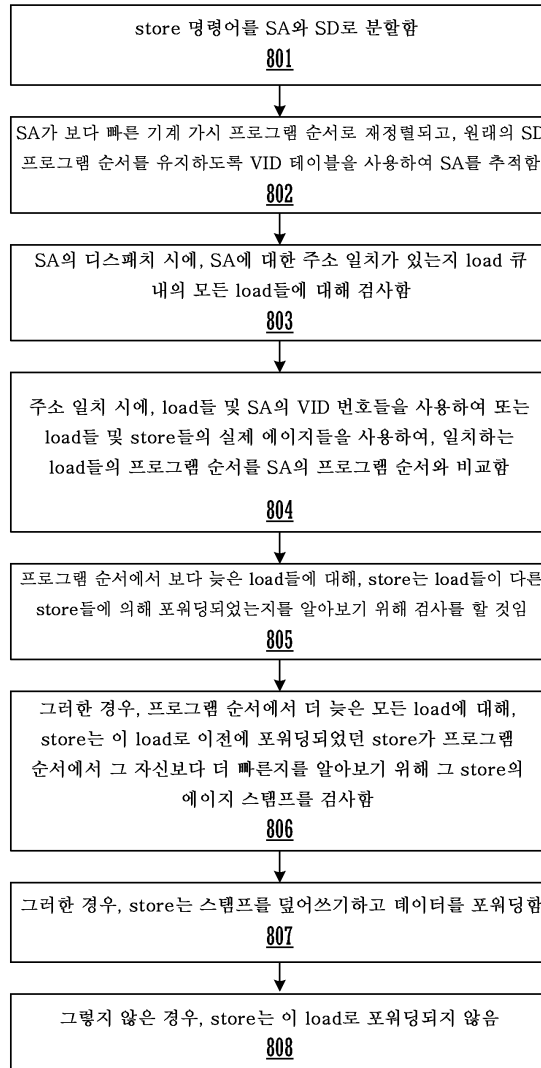
도면6



도면8

800

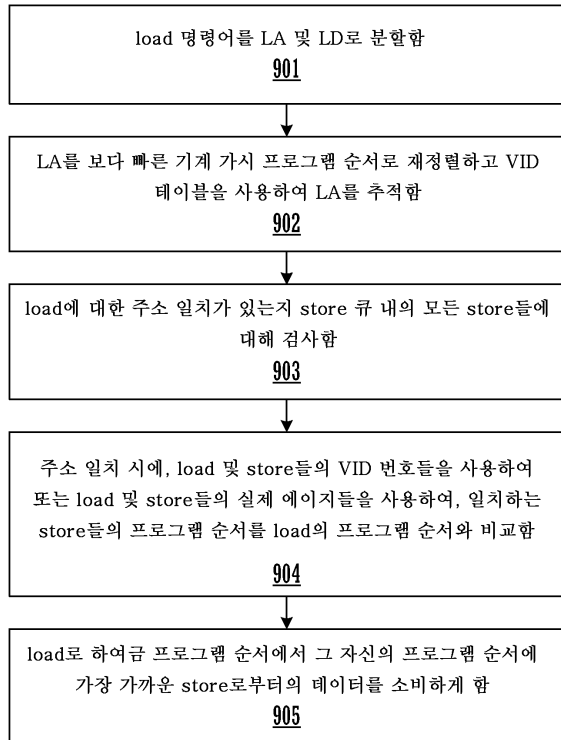
load 이후에 디스패치된 store(SD가 아직 할당되지 않음)



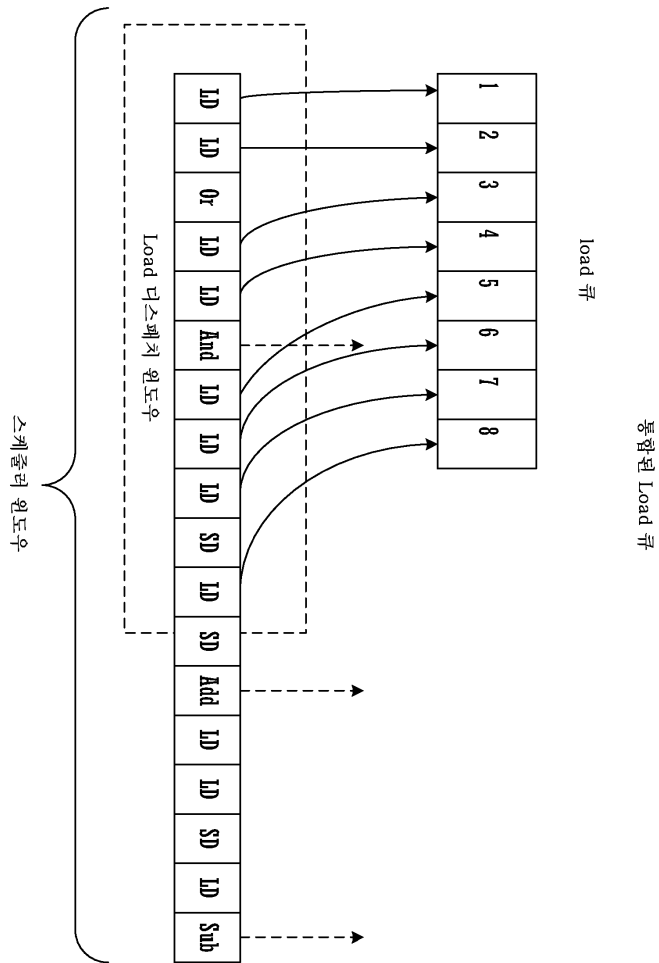
도면9

900

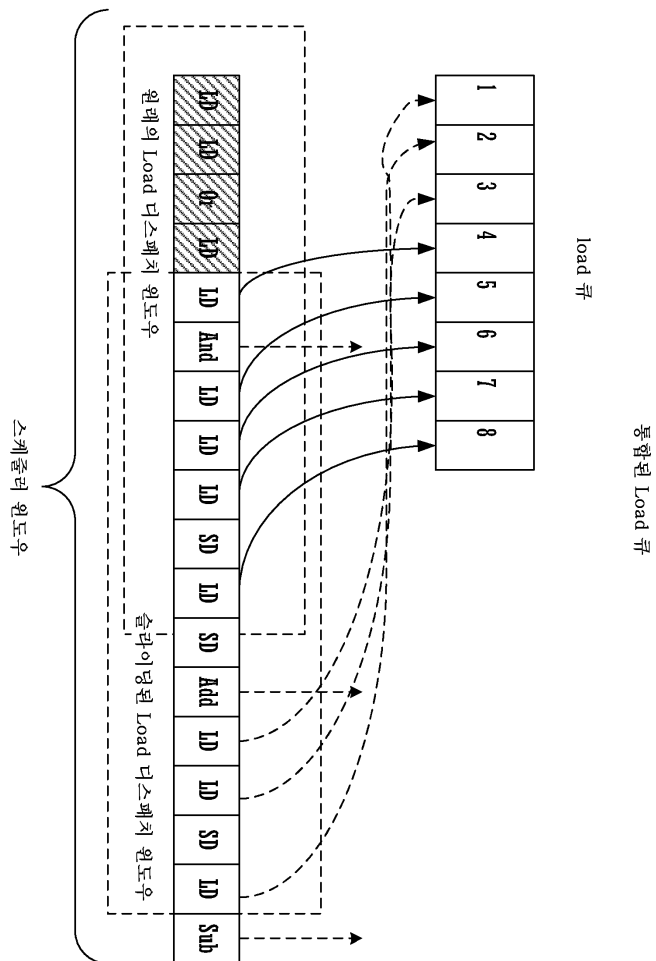
store 이후에 디스패치된 load



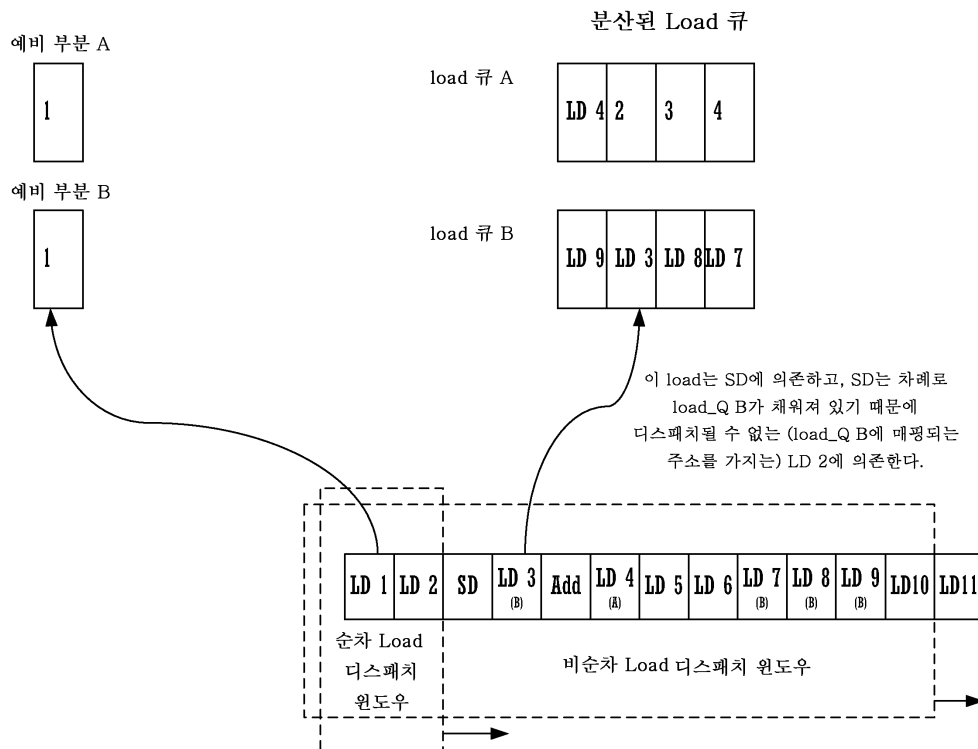
도면10



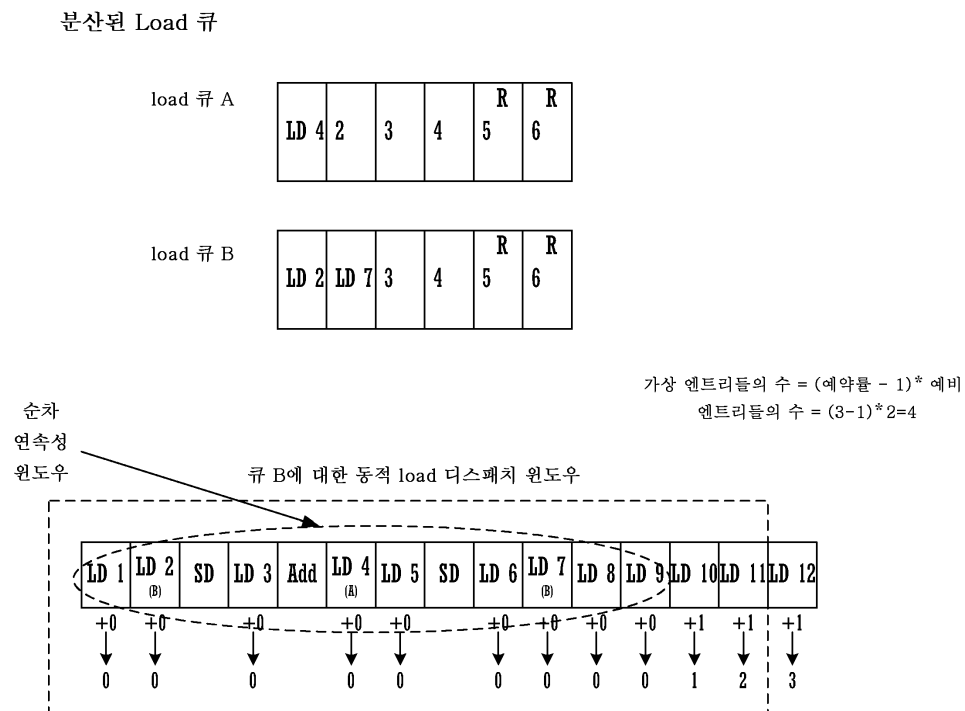
도면11



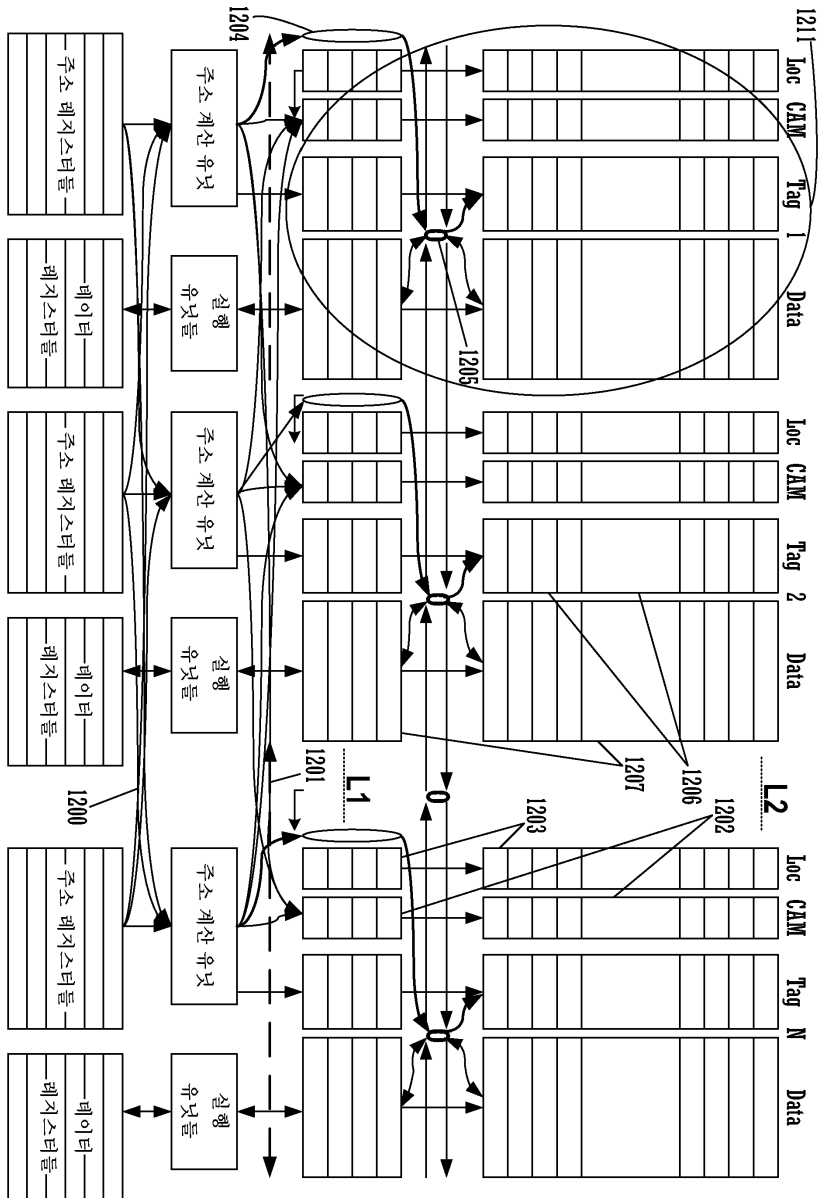
도면12



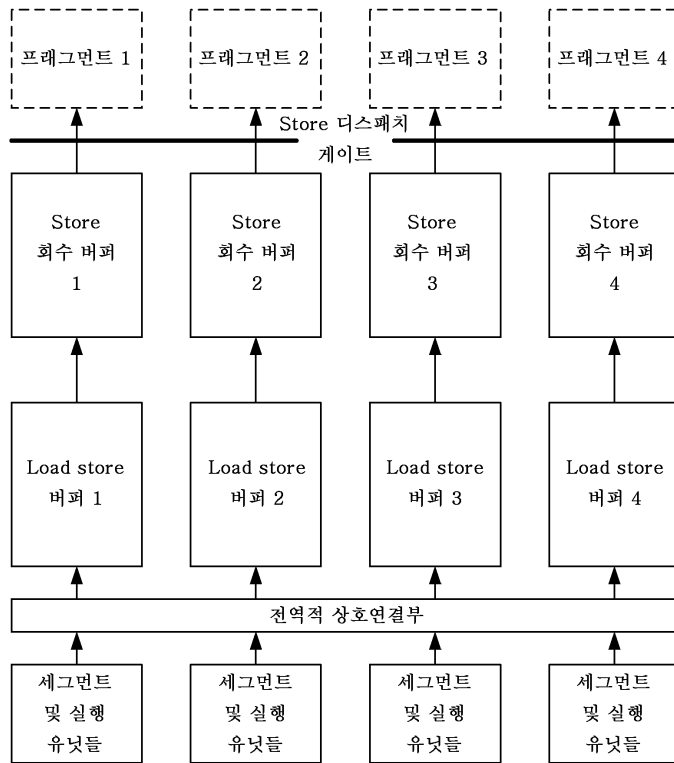
도면13



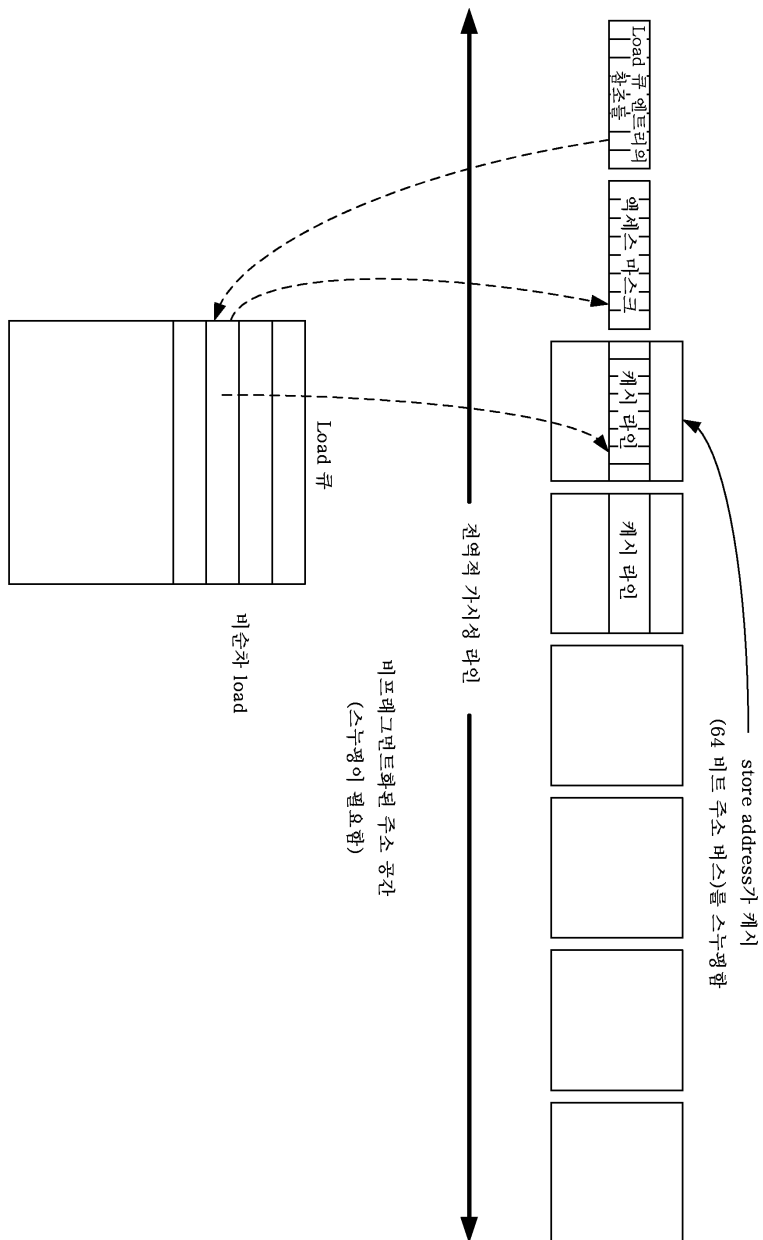
도면14



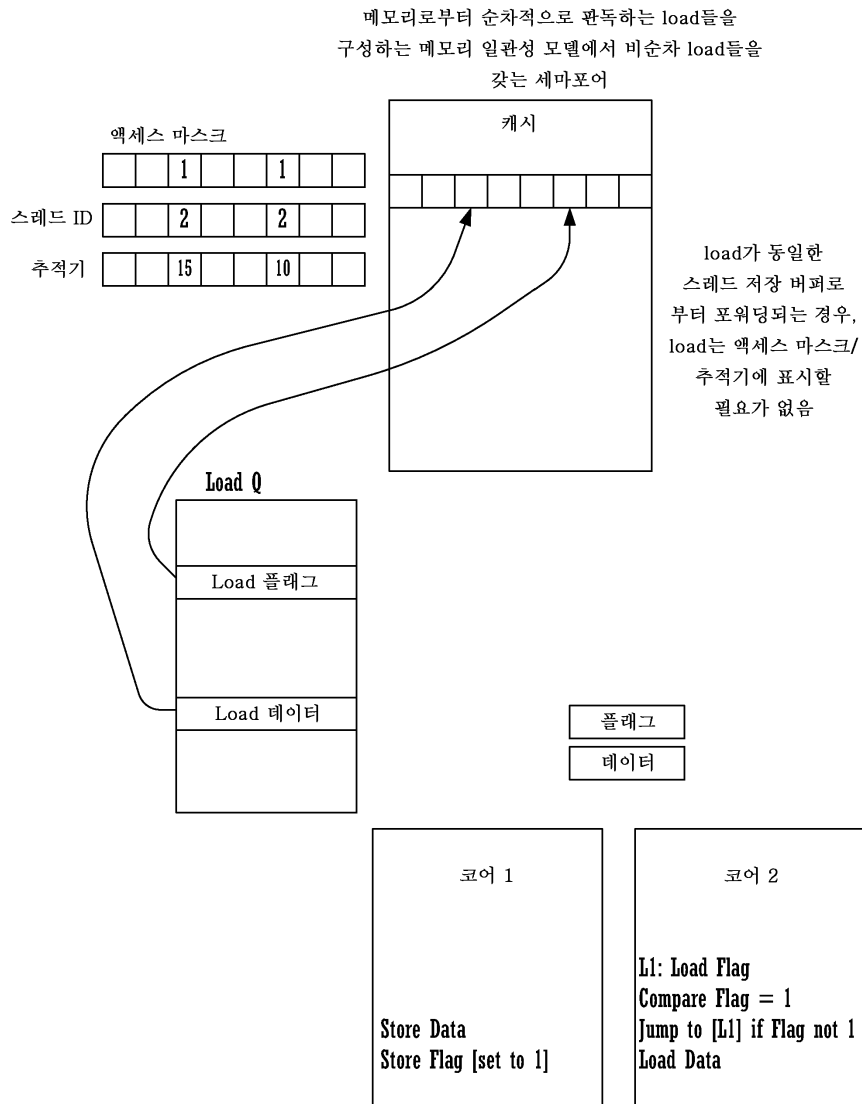
도면15



도면16

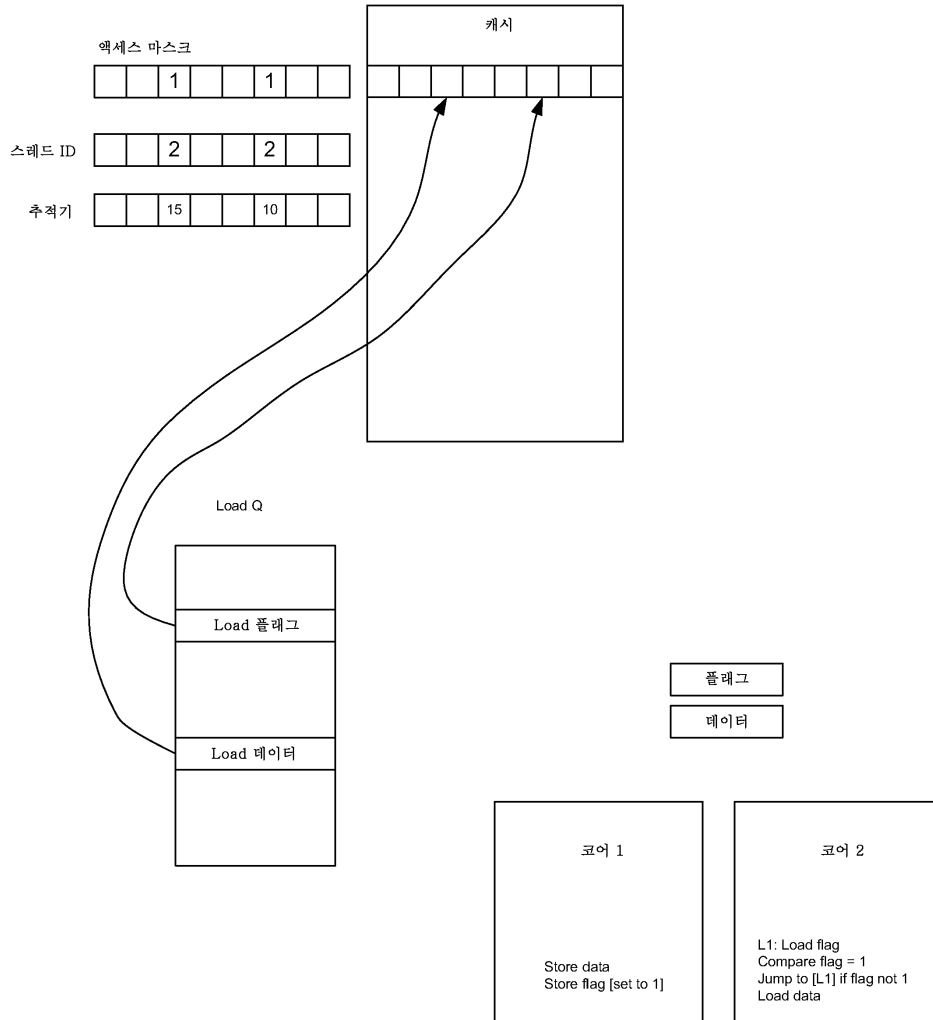


도면17

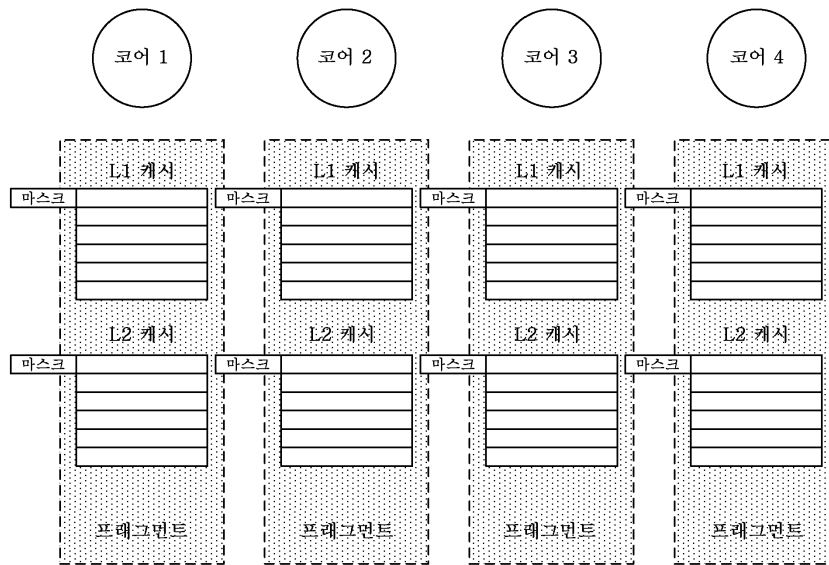


도면18

로크된 또는 트랜잭션 기반 일관성 모델
(개별적인 캐시들을 갖는 비세그먼트화된 외부 프로세서들/코어들)

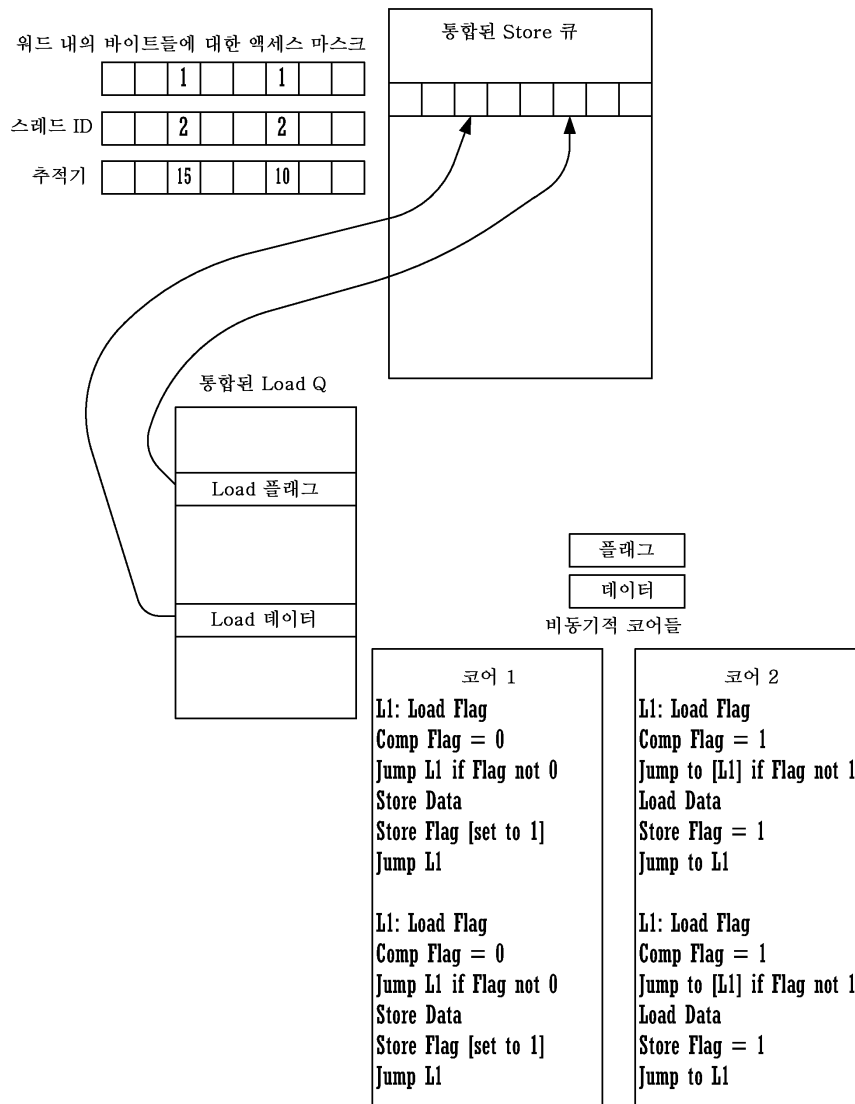


도면19



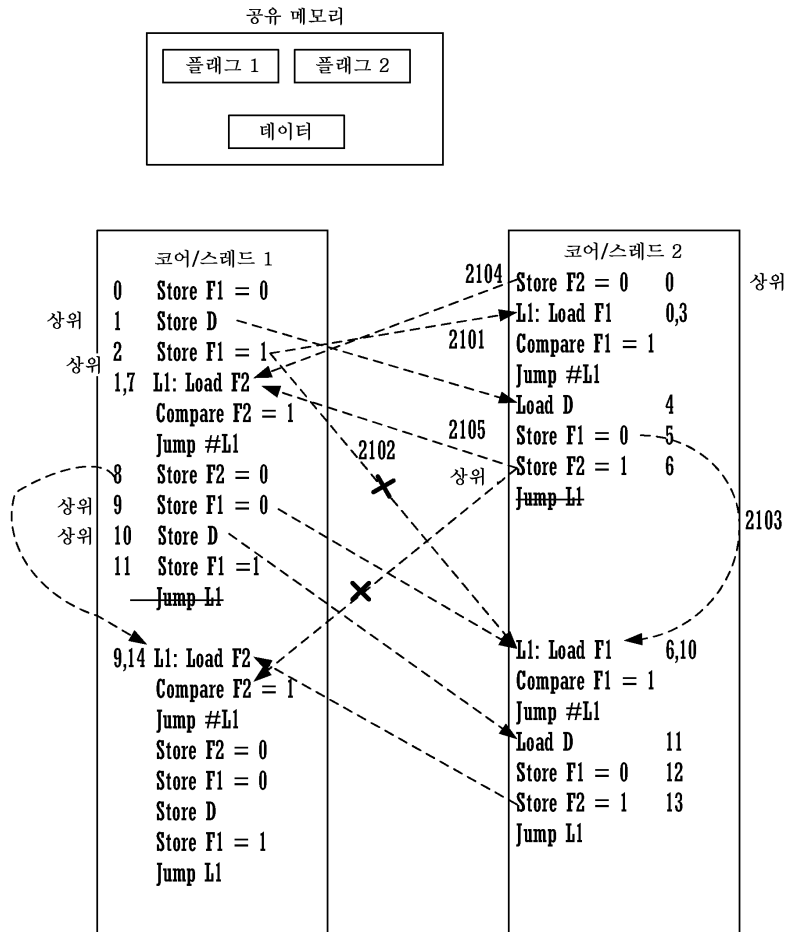
도면20

Load Store는 스택에 무관하고, store 상위 서열에 기초하여
어느 한 스택으로부터 포워딩됨

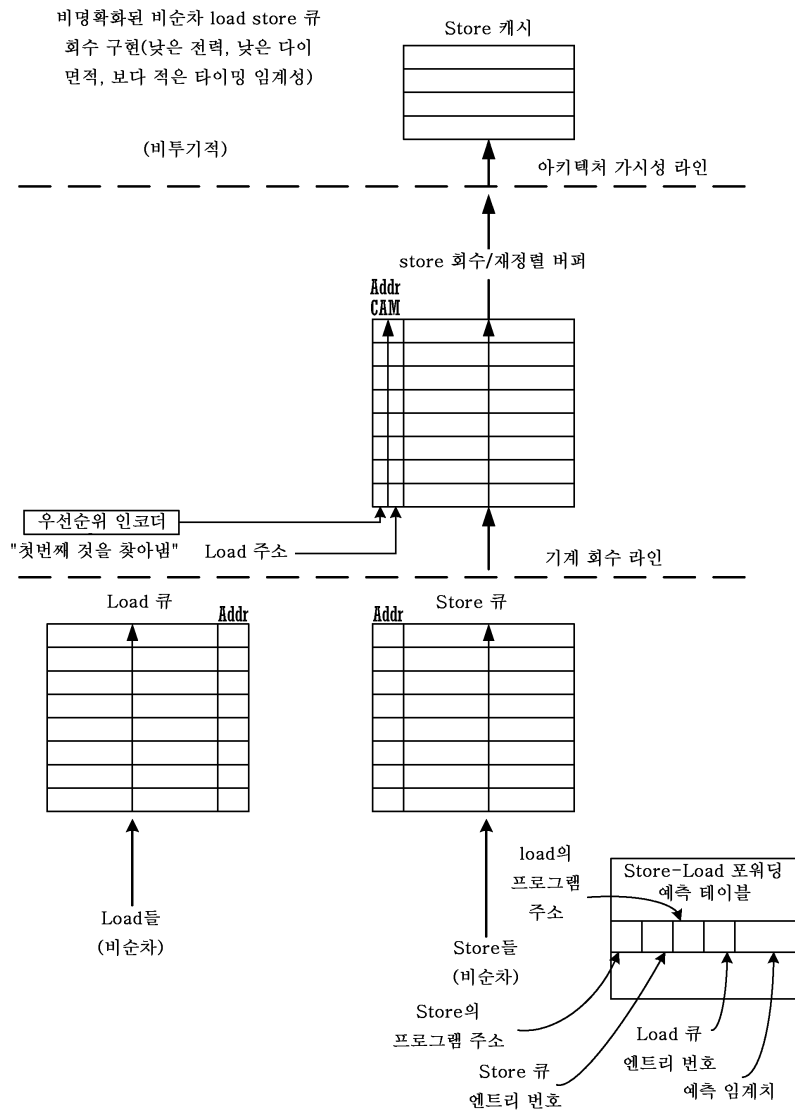


도면21

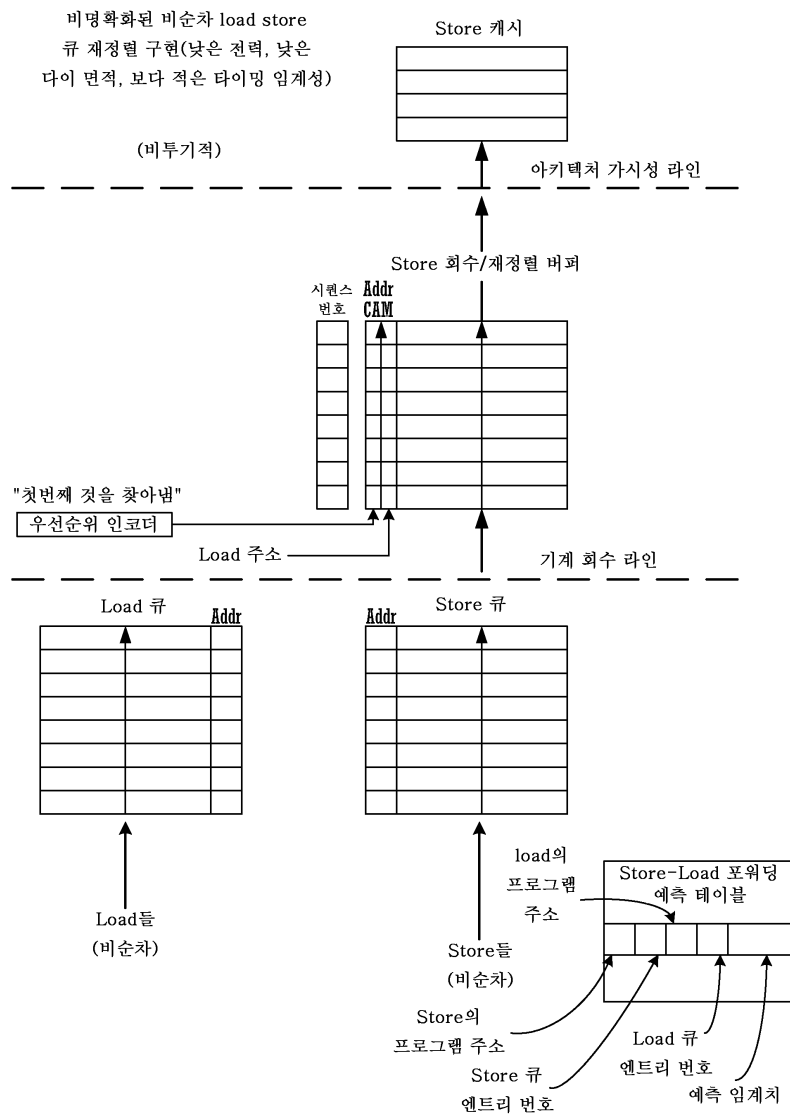
load/store 버퍼가 스레드에 무관한 기능



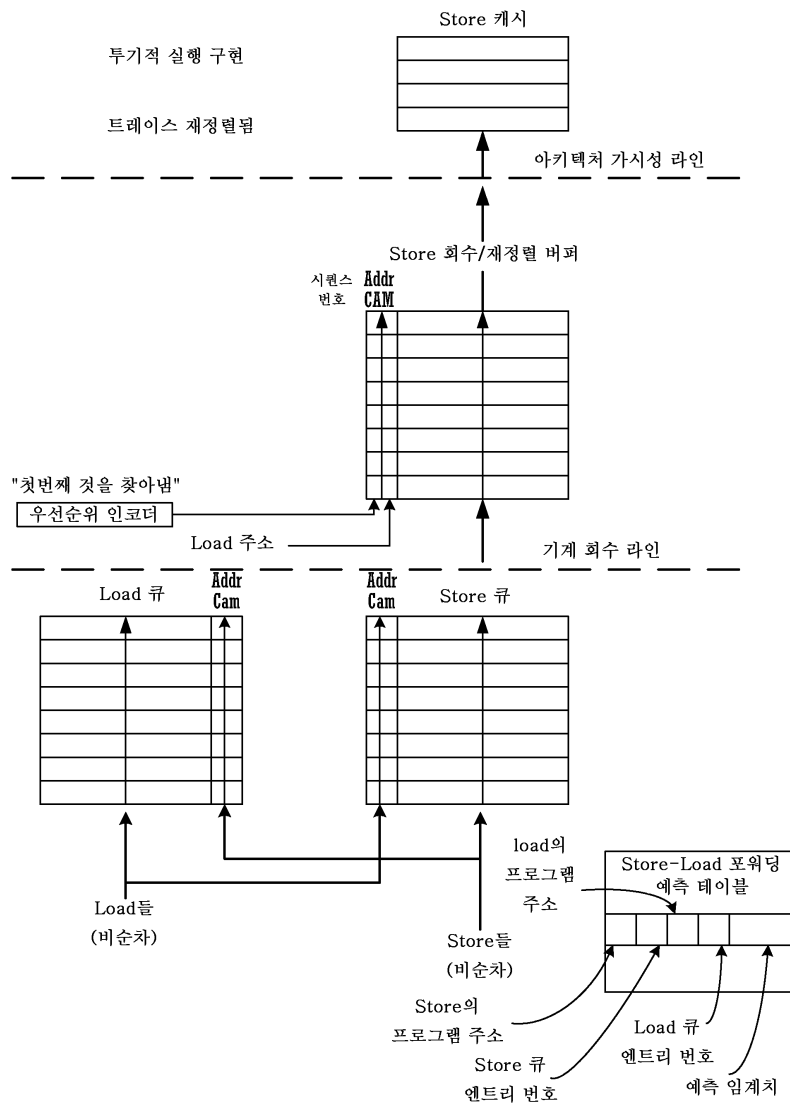
도면22



도면23



도면24



도면25

