(19) 대한민국특허청(KR)
(12) 등록특허공보(B1)

(45) 공고일자 2017년11월23일
(11) 등록번호 10-1800948
(24) 등록일자 2017년11월17일

(51) 국제특허분류(Int. Cl.)
G06F 9/30 (2017.01) *G06F 9/06* (2006.01)
G06F 9/50 (2006.01)
(52) CPC특허분류
G06F 9/30098 (2013.01)
G06F 9/06 (2013.01)
(21) 출원번호 10-2015-7029262
(22) 출원일자(국제) 2014년03월12일
심사청구일자 2015년10월14일
(85) 번역문제출일자 2015년10월14일
(65) 공개번호 10-2015-0132419
(43) 공개일자 2015년11월25일
(86) 국제출원번호 PCT/US2014/024608
(87) 국제공개번호 WO 2014/150941
국제공개일자 2014년09월25일
(30) 우선권주장
61/799,902 2013년03월15일 미국(US)
(56) 선행기술조사문헌
US06557095 B1*
US20100161948 A1*
*는 심사관에 의하여 인용된 문헌

(73) 특허권자
인텔 코포레이션
미합중국 캘리포니아 95054 산타클라라 미션 칼리지 블러바드 2200
(72) 발명자
압달라, 모하마드
미국 95132 캘리포니아주 산 호세 선크레스트 애비뉴 3868
(74) 대리인
양영준, 백만기

전체 청구항 수 : 총 21 항 심사관 : 서광훈

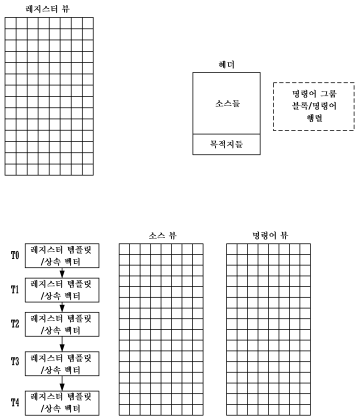
(54) 발명의 명칭 **레지스터 뷰, 소스 뷰, 명령어 뷰, 및 복수의 레지스터 템플릿을 가진 마이크로프로세서 아키텍처를 이용하여 명령어들의 블록들을 실행하는 방법**

(57) 요 약

레지스터 뷰, 소스 뷰, 명령어 뷰, 및 복수의 레지스터 템플릿을 가진 마이크로프로세서 아키텍처를 이용하여 명령어들의 블록들을 실행하는 방법이 개시된다. 이 방법은 글로벌 프런트 엔드를 이용하여 착신 명령어 시퀀스를 수신하는 단계; 상기 명령어들을 그룹화하여 명령어 블록들을 형성하는 단계; 복수의 레지스터 템플릿을 이용하

(뒷면에 계속)

대 표 도 - 도2



여, 상기 레지스터 템플릿에 상기 명령어 블록들에 대응하는 블록 번호들을 채우는 것에 의해 명령어 목적지들 및 명령어 소스들을 추적하는 단계 - 상기 명령어 블록들에 대응하는 상기 블록 번호들은 상기 명령어들의 블록들 사이의 상호 종속성들을 나타냄 -; 레지스터 뷰 데이터 구조를 이용하는 단계 - 상기 레지스터 뷰 데이터 구조는 상기 명령어 블록들에 대응하는 목적지들을 저장함 -; 소스 뷰 데이터 구조를 이용하는 단계 - 상기 소스 뷰 데이터 구조는 상기 명령어 블록들에 대응하는 소스들을 저장함 -; 및 명령어 뷰 데이터 구조를 이용하는 단계 - 상기 명령어 뷰 데이터 구조는 상기 명령어 블록들에 대응하는 명령어들을 저장함 - 를 포함한다.

(52) CPC특허분류
G06F 9/30 (2013.01)
G06F 9/3009 (2013.01)
G06F 9/5005 (2013.01)

명 세 서

청구범위

청구항 1

레지스터 뷰, 소스 뷰, 명령어 뷰 및 복수의 레지스터 템플릿을 가진 마이크로프로세서 아키텍처를 이용하여 명령어들의 블록들을 실행하는 방법으로서,

글로벌 프런트 엔드(global front end)를 이용하여 착신 명령어 시퀀스(incoming instruction sequence)를 수신하는 단계와,

상기 명령어들을 그룹화하여 명령어 블록들을 형성하는 단계와,

레지스터 템플릿을 이용하여, 상기 명령어 블록들에 대응하는 블록 번호들을 상기 레지스터 템플릿에 채우는(populating) 것에 의해 명령어 목적지들(instruction destinations)을 추적하는 단계 - 상기 명령어 블록들에 대응하는 상기 블록 번호들은 상기 명령어들의 블록들 사이의 상호 종속성들(interdependencies)을 나타내고, 상기 추적은, 착신 명령어 블록이, 각자의 블록 번호를, 상기 착신 명령어 블록에 의하여 참조되는 목적지 레지스터들에 대응하는 상기 레지스터 템플릿의 필드들에 기입하는 것을 포함함 - 와,

상기 착신 명령어 블록이 그 자신의 레지스터 소스들에 대응하는 상기 레지스터 템플릿의 필드들을 판독하여 그 명령어 소스들을 검색(retrieve)하는 단계와,

레지스터 뷰 데이터 구조를 이용하는 단계 - 상기 레지스터 뷰 데이터 구조는 상기 명령어 블록들에 대응하는 명령어 목적지들을 저장함 - 와,

소스 뷰 데이터 구조를 이용하는 단계 - 상기 소스 뷰 데이터 구조는 상기 명령어 블록들에 대응하는 명령어 소스들을 저장함 - 와,

명령어 뷰 데이터 구조를 이용하는 단계 - 상기 명령어 뷰 데이터 구조는 상기 명령어 블록들에 대응하는 명령어들을 저장함 - 와,

상기 소스 뷰 데이터 구조에 연결되고 복수의 디스패치 포트에 연결된 선택기 어레이를 이용하여, 상기 소스 뷰 데이터 구조가 준비 상태에 있다고 표시하는 복수의 상기 명령어 블록을 선택하고, 상기 준비 상태에 있는 상기 복수의 명령어 블록을 디스패치 하기 위해 상기 복수의 디스패치 포트를 활성화하는 단계

를 포함하는

방법.

청구항 2

제1항에 있어서,

상기 레지스터 뷰 데이터 구조, 상기 소스 뷰 데이터 구조 및 상기 명령어 뷰 데이터 구조는 스케줄러 아키텍처를 포함하는

방법.

청구항 3

제1항에 있어서,

상기 명령어 블록들에 의해 참조되는 레지스터들에 관한 정보가 상기 레지스터 뷰 데이터 구조에 저장되는

방법.

청구항 4

제1항에 있어서,

상기 명령어 블록들에 의해 참조되는 명령어 소스들에 관한 정보가 상기 소스 뷰 데이터 구조에 저장되는

방법.

청구항 5

제1항에 있어서,

상기 명령어 블록들에 의해 참조되는 명령어들에 관한 정보가 상기 명령어 뷰 데이터 구조에 저장되는

방법.

청구항 6

제1항에 있어서,

상기 레지스터 템플릿은 상기 명령어 블록들에 의해 참조되는 종속성(dependency) 및 상속(inheritance) 정보를 저장하는 데이터 구조들을 더 포함하는 상속 벡터들을 포함하는

방법.

청구항 7

제1항에 있어서,

상기 소스 뷰 데이터 구조는 특정 블록이 디스패치될 수 있는 때를 결정하는

방법.

청구항 8

컴퓨터 시스템에 의해 실행될 때 상기 컴퓨터 시스템으로 하여금 레지스터 뷰, 소스 뷰, 명령어 뷰 및 복수의 레지스터 템플릿을 가진 마이크로프로세서 아키텍처를 이용하여 명령어들의 블록들을 실행하기 위한 동작들을 수행하게 하는 컴퓨터 판독 가능 코드를 기록한 비일시적인 컴퓨터 판독 가능 저장 매체로서, 상기 동작들은

글로벌 프런트 엔드를 이용하여 착신 명령어 시퀀스를 수신하는 단계와,

상기 명령어들을 그룹화하여 명령어 블록들을 형성하는 단계와,

복수의 레지스터 템플릿을 이용하여, 상기 명령어 블록들에 대응하는 블록 번호들을 상기 레지스터 템플릿에 채우는 것에 의해 명령어 목적지들을 추적하는 단계 - 상기 명령어 블록들에 대응하는 상기 블록 번호들은 상기 명령어들의 블록들 사이의 상호 종속성들을 나타내고, 상기 추적은, 착신 명령어 블록이, 각자의 블록 번호를, 상기 착신 명령어 블록에 의하여 참조되는 목적지 레지스터들에 대응하는 상기 레지스터 템플릿의 필드들에 기입하는 것을 포함함 -와,

상기 착신 명령어 블록이 그 레지스터 소스들에 대응하는 상기 레지스터 템플릿의 필드들을 판독하여 그 명령어 소스들을 검색(retrieve)하는 단계와,

레지스터 뷰 데이터 구조를 이용하는 단계 - 상기 레지스터 뷰 데이터 구조는 상기 명령어 블록들에 대응하는

명령어 목적지들을 저장함 - 와,

소스 뷰 데이터 구조를 이용하는 단계 - 상기 소스 뷰 데이터 구조는 상기 명령어 블록들에 대응하는 명령어 소스들을 저장함 - 와,

명령어 뷰 데이터 구조를 이용하는 단계 - 상기 명령어 뷰 데이터 구조는 상기 명령어 블록들에 대응하는 명령어들을 저장함 - 와,

상기 소스 뷰 데이터 구조에 연결되고 복수의 디스패치 포트에 연결된 선택기 어레이를 이용하여, 상기 소스 뷰 데이터 구조가 준비 상태에 있다고 표시하는 복수의 상기 명령어 블록을 선택하고, 상기 준비 상태에 있는 상기 복수의 명령어 블록을 디스패치 하기 위해 상기 복수의 디스패치 포트를 활성화하는 단계

를 포함하는

컴퓨터 판독 가능 저장 매체.

청구항 9

제8항에 있어서,

상기 레지스터 뷰 데이터 구조, 상기 소스 뷰 데이터 구조 및 상기 명령어 뷰 데이터 구조는 스케줄러 아키텍처를 포함하는

컴퓨터 판독 가능 저장 매체.

청구항 10

제8항에 있어서,

상기 명령어 블록들에 의해 참조되는 레지스터들에 관한 정보가 상기 레지스터 뷰 데이터 구조에 저장되는

컴퓨터 판독 가능 저장 매체.

청구항 11

제8항에 있어서,

상기 명령어 블록들에 의해 참조되는 명령어 소스들에 관한 정보가 상기 소스 뷰 데이터 구조에 저장되는

컴퓨터 판독 가능 저장 매체.

청구항 12

제8항에 있어서,

상기 명령어 블록들에 의해 참조되는 명령어들에 관한 정보가 상기 명령어 뷰 데이터 구조에 저장되는

컴퓨터 판독 가능 저장 매체.

청구항 13

제8항에 있어서,

상기 레지스터 템플릿은 상기 명령어 블록들에 의해 참조되는 종속성 및 상속 정보를 저장하는 데이터 구조들을 더 포함하는 상속 벡터들을 포함하는

컴퓨터 판독 가능 저장 매체.

청구항 14

제8항에 있어서,

상기 소스 뷰 데이터 구조는 언제 특정 블록이 디스패치될 수 있는지를 결정하는

컴퓨터 판독 가능 저장 매체.

청구항 15

컴퓨터 판독 가능 코드를 갖는 컴퓨터 판독 가능 저장 매체에 연결되는 프로세서를 갖는 컴퓨터 시스템으로서, 상기 컴퓨터 판독 가능 코드는, 상기 프로세서에 의해 실행될 때, 상기 컴퓨터 시스템으로 하여금 레지스터 뷰, 소스 뷰, 명령어 뷰 및 복수의 레지스터 템플릿을 가진 아키텍처를 구현하게 하고, 상기 아키텍처는,

글로벌 프런트 엔드를 이용하여 착신 명령어 시퀀스를 수신하고,

상기 명령어들을 그룹화하여 명령어 블록들을 형성하고,

레지스터 템플릿을 이용하여, 상기 명령어 블록들에 대응하는 블록 번호들을 상기 레지스터 템플릿에 채우는 것에 의해 명령어 목적지들을 추적하고 - 상기 명령어 블록들에 대응하는 상기 블록 번호들은 상기 명령어들의 블록들 사이의 상호 종속성들을 나타내고, 상기 추적은, 착신 명령어 블록이, 각자의 블록 번호를, 상기 착신 명령어 블록에 의하여 참조되는 목적지 레지스터들에 대응하는 상기 레지스터 템플릿의 필드들에 기입하는 것을 포함하며, 상기 착신 명령어 블록은 그 레지스터 소스들에 대응하는 상기 레지스터 템플릿의 필드들을 판독하여 그 명령어 소스들을 검색(retrieve)함 -,

레지스터 뷰 데이터 구조를 이용하고 - 상기 레지스터 뷰 데이터 구조는 상기 명령어 블록들에 대응하는 명령어 목적지들을 저장함 -,

소스 뷰 데이터 구조를 이용하고 - 상기 소스 뷰 데이터 구조는 상기 명령어 블록들에 대응하는 명령어 소스들을 저장함 -,

명령어 뷰 데이터 구조를 이용하며 - 상기 명령어 뷰 데이터 구조는 상기 명령어 블록들에 대응하는 명령어들을 저장함 - ,

상기 소스 뷰 데이터 구조에 연결되고 복수의 디스패치 포트에 연결된 선택기 어레이를 이용하여, 상기 소스 뷰 데이터 구조가 준비 상태에 있다고 표시하는 복수의 상기 명령어 블록을 선택하고, 상기 준비 상태에 있는 상기 복수의 명령어 블록을 디스패치 하기 위해 상기 복수의 디스패치 포트를 활성화하는,

컴퓨터 시스템.

청구항 16

제15항에 있어서,

상기 레지스터 뷰 데이터 구조, 상기 소스 뷰 데이터 구조 및 상기 명령어 뷰 데이터 구조는 스케줄러 아키텍처를 포함하는

컴퓨터 시스템.

청구항 17

제15항에 있어서,

상기 명령어 블록들에 의해 참조되는 레지스터들에 관한 정보가 상기 레지스터 뷰 데이터 구조에 저장되는

컴퓨터 시스템.

청구항 18

제15항에 있어서,

상기 명령어 블록들에 의해 참조되는 소스들에 관한 정보가 상기 소스 뷰 데이터 구조에 저장되는

컴퓨터 시스템.

청구항 19

제15항에 있어서,

상기 명령어 블록들에 의해 참조되는 명령어들에 관한 정보가 상기 명령어 뷰 데이터 구조에 저장되는

컴퓨터 시스템.

청구항 20

제15항에 있어서,

상기 레지스터 템플릿은 상기 명령어 블록들에 의해 참조되는 종속성 및 상속 정보를 저장하는 데이터 구조들을 더 포함하는 상속 벡터들을 포함하는

컴퓨터 시스템.

청구항 21

컴퓨터 시스템에서, 명령어들의 블록들 사이의 상호 종속성들을 추적하기 위해 레지스터 템플릿을 이용하여 명령어들을 실행하기 위한 방법으로서,

글로벌 프런트 엔드를 이용하여 착신 명령어 시퀀스를 수신하는 단계와,

상기 착신 명령어 시퀀스의 명령어들을 그룹화하여 명령어 블록들을 형성하는 단계와,

레지스터 템플릿을 이용하여, 상기 명령어 블록들에 대응하는 블록 번호들을 상기 레지스터 템플릿에 채우는 것에 의해 명령어 목적지들을 추적하는 단계 - 상기 명령어 블록들에 대응하는 상기 블록 번호들은 상기 명령어들의 블록들 사이의 상호 종속성들을 나타내고, 상기 추적은, 착신 명령어 블록이, 각자의 블록 번호를, 상기 착신 명령어 블록에 의하여 참조되는 목적지 레지스터들에 대응하는 상기 레지스터 템플릿의 필드들에 기입하는 것을 포함함 -와,

상기 착신 명령어 블록이 그 레지스터 소스들에 대응하는 상기 레지스터 템플릿의 필드들을 판독하여 그 명령어 소스들을 검색(retrieve)하는 단계와,

소스 뷰 데이터 구조를 이용하는 단계 - 상기 소스 뷰 데이터 구조는 상기 명령어 블록들에 대응하는 명령어 소스들을 저장함 - 와,

상기 소스 뷰 데이터 구조에 연결되고 복수의 디스패치 포트에 연결된 선택기 어레이를 이용하여, 상기 소스 뷰 데이터 구조가 준비 상태에 있다고 표시하는 복수의 상기 명령어 블록을 선택하고, 상기 준비 상태에 있는 상기 복수의 명령어 블록을 디스패치 하기 위해 상기 복수의 디스패치 포트를 활성화하는 단계

를 포함하는

방법.

발명의 설명

기 술 분 야

[0001] 이 출원은 Mohammad A. Abdallah에 의해, 2013년 3월 15일에 출원되고, 그 전부가 본 명세서에 포함되는, 발명의 명칭이 "A METHOD FOR EXECUTING BLOCKS OF INSTRUCTIONS USING A MICROPROCESSOR ARCHITECTURE HAVING A REGISTER VIEW, SOURCE VIEW, INSTRUCTION VIEW, AND A PLURALITY OF REGISTER TEMPLATES"인, 동시 계류중이고 공동 양도된 미국 특허 가출원 번호 제61/799,902호에 대한 우선권을 주장한다.

[0002] <관련 출원과의 교차 참조>

[0003] 이 출원은 Mohammad A. Abdallah에 의해, 2007년 4월 12일에 출원되고, 그 전부가 본 명세서에 포함되는, 발명의 명칭이 "APPARATUS AND METHOD FOR PROCESSING AN INSTRUCTION MATRIX SPECIFYING PARALLEL INDEPENDENT OPERATIONS"인, 동시 계류중이고 공동 양도된 미국 특허 출원 번호 제2009/0113170호와 관련이 있다.

[0004] 이 출원은 Mohammad A. Abdallah에 의해, 2007년 11월 14일에 출원되고, 그 전부가 본 명세서에 포함되는, 발명의 명칭이 "APPARATUS AND METHOD FOR PROCESSING COMPLEX INSTRUCTION FORMATS IN A MULTITHREADED ARCHITECTURE SUPPORTING VARIOUS CONTEXT SWITCH MODES AND VIRTUALIZATION SCHEMES"인, 동시 계류중이고 공동 양도된 미국 특허 출원 번호 제2010/0161948호와 관련이 있다.

[0005] <발명의 분야>

[0006] 본 발명은 일반적으로 디지털 컴퓨터 시스템에 관한 것으로, 보다 상세하게는, 명령어 시퀀스를 구성하는 명령어들을 선택하기 위한 시스템 및 방법에 관한 것이다.

배 경 기 술

[0007] 의존적이거나 완전히 독립적인 다수의 작업들을 처리하는 데 프로세서가 필요하다. 이러한 프로세서의 내부 상태는 보통 프로그램 실행의 각각의 특정의 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.

[0008] 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 머신/프로세서의 상태가 저장되어야만 한다. 새로운 함수가 종료되면, 그의 상태가 폐기될 수 있고 이전의 컨텍스트의 상태가 복원될 것이며 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 수백 개의 사이클들을 포함한다.

[0009] 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 단점, 즉, 하드웨어로 지원되는 각각의 부가적인 스레드에 대한 모든 아키텍처 상태 요소들(즉, 레지스터들)을 복제하는 것의 면적, 능력(power) 및 복잡도를 가진다. 그에 부가하여, 소프트웨어 스레드들의 수가 명시적으로 지원되는 하드웨어 스레드들의 수를 초과하는 경우, 컨텍스트 전환이 여전히 수행되어야 한다.

[0010] 이것은 많은 수의 스레드를 요구하는 미세 입도 기반으로(on a fine granularity basis) 병렬 처리(parallelism)가 필요할 때 흔해진다. 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 비-명시적으로/용이하게 병렬화되는/스레딩되는 소프트웨어 코드들에 대해 최신의 컴파일러 또는 사용자 병렬화 기법들을 사용한 이러한 코드들의 자동 병렬화의 어려움을 수반한다.

발명의 내용

[0011] 일 실시예에서, 본 발명은 레지스터 뷰, 소스 뷰, 명령어 뷰, 및 복수의 레지스터 템플릿을 가진 마이크로프로세서 아키텍처를 이용하여 명령어들의 블록들을 실행하는 방법으로서 구현된다. 이 방법은 글로벌 프런트 엔드(global front end)를 이용하여 착신 명령어 시퀀스를 수신하는 단계; 상기 명령어들을 그룹화하여 명령어 블록들을 형성하는 단계; 복수의 레지스터 템플릿을 이용하여, 상기 레지스터 템플릿에 상기 명령어 블록들에 대응하는 블록 번호들을 채우는(populating) 것에 의해 명령어 목적지들(instruction destinations) 및 명령어 소스들(instruction sources)을 추적하는 단계 - 상기 명령어 블록들에 대응하는 상기 블록 번호들은 상기 명령어 블록들 사이의 상호 종속성들을 나타냄 -; 레지스터 뷰 데이터 구조를 이용하는 단계 - 상기 레지스터 뷰 데이터 구조는 상기 명령어 블록들에 대응하는 목적지들을 저장함 -; 소스 뷰 데이터 구조를 이용하는 단계 - 상기 소스 뷰 데이터 구조는 상기 명령어 블록들에 대응하는 소스들을 저장함 -; 및 명령어 뷰 데이터 구조를 이용하는 단계 - 상기 명령어 뷰 데이터 구조는 상기 명령어 블록들에 대응하는 명령어들을 저장함 - 를 포함한다.

[0012] 전술한 내용은 요약이므로, 필요에 의해, 세부 사항의 간소화, 일반화 및 생략을 포함한다; 따라서, 통상의 기술자들은 이 요약이 예시적일 뿐이고 어떤 점에서도 제한하고자 하는 것은 아니라는 것을 알 것이다. 청구범위에 의해서만 정의되는, 본 발명의 다른 양태들, 창의적 특징들, 및 이점들은 아래에 기술되는 비제한적인 상세 설명에서 명백해질 것이다.

도면의 간단한 설명

[0013] 본 발명은 첨부 도면들에서 제한으로서가 아니라 예로서 도시되어 있으며, 도면들에서 같은 참조 번호들은 유사한 요소들을 가리킨다.

도 1은 명령어들을 블록으로 그룹화하고 레지스터 템플릿을 이용하여 명령어들 사이의 종속성들을 추적하는 프로세스의 개요도를 보여준다.

도 2는 본 발명의 일 실시예에 따른 레지스터 뷰, 소스 뷰, 및 명령어 뷰의 개요도를 보여준다.

도 3은 본 발명의 일 실시예에 따른 예시적인 레지스터 템플릿과 이 레지스터 템플릿으로부터의 정보에 의해 소스 뷰가 채워지는 방법을 설명하는 도면을 보여준다.

도 4는 소스 뷰 내의 종속성 브로드캐스팅을 위한 제1 실시예를 설명하는 도면을 보여준다. 이 실시예에서, 각각의 열은 명령어 블록을 포함한다.

도 5는 소스 뷰 내의 종속성 브로드캐스팅을 위한 제2 실시예를 설명하는 도면을 보여준다.

도 6은 본 발명의 일 실시예에 따른 커밋 포인터로부터 시작하여 대응하는 포트 할당들을 브로드캐스팅하는 디스패치를 위한 준비 블록들의 선택을 설명하는 도면을 보여준다.

도 7은 본 발명의 일 실시예에 따른 도 6에 설명된 선택기 어레이를 구현하는 데 이용되는 덧셈기 트리 구조를 보여준다.

도 8은 선택기 어레이 덧셈기 트리의 예시적인 로직을 보다 상세히 보여준다.

도 9는 본 발명의 일 실시예에 따른 선택기 어레이를 구현하기 위한 덧셈기 트리의 병렬 구현을 보여준다.

도 10은 본 발명의 일 실시예에 따른 도 9로부터의 덧셈기 X가 캐리 세이브 덧셈기(carry save adder)들을 이용하여 구현될 수 있는 방법을 설명하는 예시도를 보여준다.

도 11은 본 발명의 일 실시예에 따른 커밋 포인터로부터 시작하여 선택기 어레이 덧셈기들을 이용하여 스케줄링하기 위해 준비 비트들을 마스킹하기 위한 마스킹 실시예를 보여준다.

도 12는 본 발명의 일 실시예에 따른 레지스터 뷰 엔트리들이 레지스터 템플릿들에 의해 채워지는 방법의 개요도를 보여준다.

도 13은 본 발명의 일 실시예에 따른 감소된 레지스터 뷰 풋프린트에 대한 제1 실시예를 보여준다.

도 14는 본 발명의 일 실시예에 따른 감소된 레지스터 뷰 풋프린트에 대한 제2 실시예를 보여준다.

도 15는 본 발명의 일 실시예에 따른 스냅샷(snapshot)들 간의 델타의 예시적인 포맷을 보여준다.

도 16은 본 발명의 일 실시예에 따른 명령어 블록들의 할당들에 따라 레지스터 템플릿 스냅샷들을 생성하기 위한 프로세스의 도면을 보여준다.

도 17은 본 발명의 일 실시예에 따른 명령어 블록들의 할당들에 따라 레지스터 템플릿 스냅샷들을 생성하기 위한 프로세스의 다른 도면을 보여준다.

도 18은 본 발명의 일 실시예에 따른 이전 레지스터 템플릿으로부터 후속 레지스터 템플릿을 생성하는 직렬 구현을 구현하기 위한 하드웨어의 개요도를 보여준다.

도 19는 본 발명의 일 실시예에 따른 이전 레지스터 템플릿으로부터 후속 레지스터 템플릿을 생성하는 병렬 구현을 구현하기 위한 하드웨어의 개요도를 보여준다.

도 20은 본 발명의 일 실시예에 따른 명령어 블록 기반 실행을 위한 하드웨어와, 그것이 소스 뷰, 명령어 뷰, 레지스터 템플릿들, 및 레지스터 뷰와 함께 동작하는 방법에 대한 개요도를 보여준다.

도 21은 본 발명의 일 실시예에 따른 청킹 아키텍처(chunking architecture)의 예를 보여준다.

도 22는 본 발명의 일 실시예에 따른 스레드들이 그들의 블록 번호들 및 스레드 ID에 따라 할당되는 방법에 대한 묘사를 보여준다.

도 23은 본 발명의 일 실시예에 따른 멀티스레드 실행(multithreaded execution)을 관리하기 위하여 물리적 저장 위치들을 가리키는 스레드 기반 포인터 맵들을 이용하는 스케줄러의 구현을 보여준다.

도 24는 본 발명의 일 실시예에 따른 스레드 기반 포인터 맵들을 이용하는 스케줄러의 다른 구현을 보여준다.

도 25는 본 발명의 일 실시예에 따른 스레드들에 대한 실행 리소스들의 동적인 캘린더 기반 할당에 대한 도면을 보여준다.

도 26은 본 발명의 일 실시예에 따른 이중 디스패치 프로세스를 도시한다.

도 27은 본 발명의 일 실시예에 따른 이중 디스패치 일시적 곱셈-누산을 도시한다.

도 28은 본 발명의 일 실시예에 따른 이중 디스패치 아키텍처적으로 보이는 상태 곱셈-덧셈(dual dispatch architecturally visible state multiply-add)을 도시한다.

도 29는 본 발명의 일 실시예에 따른 그룹화된 실행 유닛들의 프로세스에서의 실행을 위한 명령어 블록들의 페치 및 형성의 개요도를 보여준다.

도 30은 본 발명의 일 실시예에 따른 명령어 그룹화의 예시도를 보여준다. 도 30의 실시예에서는 2개의 명령어가 제3의 보조 연산과 함께 도시되어 있다.

도 31은 본 발명의 일 실시예에 따른 블록 스택 내의 절반 블록 쌍들이 실행 블록 유닛들에 매핑되는 방법을 보여준다.

도 32는 본 발명의 일 실시예에 따른 제1 레벨 레지스터 파일로서 중간 블록 결과들의 저장을 묘사하는 도면을 보여준다.

도 33은 본 발명의 일 실시예에 따른 홀수/짝수 포트 스케줄러를 보여준다.

도 34는 4개의 실행 유닛이 스케줄러 어레이로부터 결과들을 수신하고 임시 레지스터 파일 세그먼트에 출력들을 기입하는 것으로 도시된 도 33의 보다 상세한 버전을 보여준다.

도 35는 본 발명의 일 실시예에 따른 게스트 플래그 아키텍처 에뮬레이션을 묘사하는 도면을 보여준다.

도 36은 본 발명의 일 실시예에 따른 머신의 프런트 엔드, 스케줄러 및 실행 유닛들과 중앙 플래그 레지스터를 설명하는 도면을 보여준다.

도 37은 본 발명의 실시예들에 의해 구현되는 중앙 플래그 레지스터 에뮬레이션 프로세스에 대한 도면을 보여준다.

도 38은 게스트 설정에서 중앙 플래그 레지스터 거동을 에뮬레이션하는 프로세스(3800)의 단계들의 순서도를 보여준다.

발명을 실시하기 위한 구체적인 내용

[0014] 본 발명이 하나의 실시예와 관련하여 기술되어 있지만, 본 발명이 본 명세서에 기재된 구체적인 형태들로 제한되는 것으로 의도되어 있지 않다. 그와 달리, 본 발명이 첨부된 청구항들에 의해 한정되는 본 발명의 범주 내에 타당하게 포함될 수 있는 이러한 대안들, 수정들 및 등가물들을 포함하는 것으로 의도되어 있다.

[0015] 이하의 상세한 설명에서, 구체적인 방법 순서들, 구조들, 요소들 및 연결들과 같은 수많은 구체적인 상세들이 기재되어 있다. 그렇지만, 이 구체적인 상세들 및 다른 구체적인 상세들이 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.

[0016] 본 명세서 내에서 "하나의 실시예" 또는 "일 실시예"에 대한 언급들은 그 실시예와 관련하여 기술된 특정의 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 꼭 동일한 실시예를 말하는 것이 아니며 다른 실시예들과 상호 배타적인 별도의 또는 대안의 실시예들도 아니다. 더욱이, 일부 실시예들에서는 나타내고 있을 수 있지만 다른 실시예들에서는 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.

[0017] 이하의 상세한 설명의 어떤 부분들은 컴퓨터 메모리 내의 데이터 비트들에 대한 동작들의 절차들, 단계들, 논리 블록들, 처리 및 다른 심볼 표현들로 제시되어 있다. 이 설명들 및 표현들은 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 모순 없는 단계들 또는 명령어들의 시퀀스인 것으로 생각된다. 단계들은 물리적 양의 물리적 조작을 필요로 하는 것이다. 보통, 꼭 그렇지는 않지만, 이 양들은 컴퓨터 판독 가능 저장 매체의 전기 또는 자기 신호의 형태를 취하고, 컴퓨터 시스템에서 저장, 전송, 결합, 비교, 및 다른 방식으로 조작될 수 있다. 대체적으로 흔히 사용되기 때문에, 이 신호들을 비트, 값, 요소, 심볼, 문자, 용어, 숫자 등으로 지칭하는 것이 때로는 편리한 것으로 밝혀졌다.

[0018] 그렇지만, 이 용어들 및 유사한 용어들 모두가 적절한 물리적 양들과 연관되어 있고 이 양들에 적용되는 편리한 라벨들에 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 판독 가능 매체 내의 물리적(전자적) 양으로 표현된 데이터를 조작하고 컴퓨터 시스템 메모리 또는 레지스터 또는 다른 이러한 정보 저장, 전송 또는 디스플레이 디바이스 내의 물리적 양으로 유사하게 표현되는 다른 데이터로 변환하는 컴퓨터 시스템 또는 유사한 전자 컴퓨팅 디바이스의 동작 및 프로세스를 말한다는 것을 잘 알 것이다.

[0019] 도 1은 명령어들을 블록으로 그룹화하고 레지스터 템플릿을 이용하여 명령어들 사이의 종속성들을 추적하는 프로세스의 개요도를 보여준다.

[0020] 도 1은 헤더와 보디를 가진 명령어 블록을 보여준다. 블록은 명령어들의 그룹으로부터 생성된다. 블록은 명령어들의 그룹을 캡슐화하는 엔티티를 포함한다. 마이크로프로세서의 본 실시예에서, 추상화의 레벨은 개별 명령어들 대신에 블록들로 상승된다. 블록들이 개별 명령어들 대신에 디스패치를 위해 처리된다. 각 블록은 블록 번호로 표시된다. 그렇게 함으로써 머신의 비순차 관리 작업이 크게 단순화된다. 한 가지 중요한 특징은 머신의 관리 오버헤드를 크게 증가시키지 않고 처리되는 더 많은 수의 명령어들을 관리하는 방법을 찾는 것이다.

[0021] 본 발명의 실시예들은 명령어 블록들, 레지스터 템플릿들 및 상속 벡터(inheritance vector)들을 구현함으로써 이 목적을 달성한다. 도 1에 도시된 블록에서, 블록의 헤더는 블록의 명령어들의 모든 소스들과 목적지들 및 그 소스들의 출처(예컨대, 어느 블록들로부터인지)를 열거하고 캡슐화한다. 헤더는 레지스터 템플릿을 업데이트하는 목적지들을 포함한다. 헤더에 포함된 소스들은 레지스터 템플릿에 저장된 블록 번호들로 연결(concatenate)될 것이다.

[0022] 비순차로 처리되는 명령어들의 수는 비순차 머신의 관리 복잡도를 결정한다. 더 많은 비순차 명령어들은 더 큰 복잡도로 이어진다. 소스들은 프로세서의 비순차 디스패치 윈도에서 이전 명령어들의 목적지들과 비교할 필요가 있다.

[0023] 도 1에 도시된 바와 같이, 레지스터 템플릿은 R0에서 R63까지 각 레지스터에 대한 필드들을 갖는다. 블록들은 그 각자의 블록 번호들을 블록 목적지들에 대응하는 레지스터 템플릿 필드들에 기입한다. 각각의 블록은 레지스터 템플릿으로부터 그의 레지스터 소스들을 나타내는 레지스터 필드들을 판독한다. 블록이 리타이어(retire)하고 그의 목적지 레지스터 콘텐츠를 레지스터 파일에 기입할 때, 그의 번호가 레지스터 템플릿으로부터 삭제된다. 이것은 그 레지스터들이 레지스터 파일 자체로부터 소스들로서 판독될 수 있다는 것을 의미한다.

[0024] 본 실시예에서, 레지스터 템플릿은 블록이 할당될 때마다 머신의 각 사이클에서 업데이트된다. 새로운 템플릿 업데이트들이 생성됨에 따라, 레지스터 템플릿들의 이전 스냅샷들이 블록마다 하나씩 어레이(예컨대, 도 2에 도시된 레지스터 뷰)에 저장된다. 이 정보는 대응하는 블록이 리타이어될 때까지 유지된다. 이것은 머신이 (예컨대, 마지막으로 알려진 종속성 상태를 획득하는 것에 의해) 예측 오류(miss-predictions)와 플러시(flushes)로부터 매우 신속히 복구되게 한다.

[0025] 일 실시예에서, 레지스터 뷰에 저장된 레지스터 템플릿들은 연속적인 스냅샷들 사이의 델타(스냅샷들 사이의 증분적 변화들)만을 저장함으로써 압축될 수 있다(그렇게 함으로써 저장 공간을 절약한다). 이렇게 하여 머신은 축소된 레지스터 뷰(shrunk register view)를 획득한다. 분기 명령어를 가진 블록들에 대한 템플릿들만을 저장함으로써 추가 압축이 획득될 수 있다.

[0026] 분기 예측 오류 외에 복구 지점이 요구된다면, 먼저 분기 복구 지점에서 복구가 획득되고, 그 후 머신이 복구 지점 후에 찾는 것에 도달할 때까지 명령어들을 할당하는 것에서 벗어나서(명령어들을 실행하지는 않고) 상태가 재구축될 수 있다.

[0027] 일 실시예에서, 본 명세서에서 사용된 용어 "레지스터 템플릿"은 Mohammad Abdallah에 의해, 2012년 3월 23일에 출원되고, 그 전부가 본 명세서에 포함되는, 발명의 명칭이 "EXECUTING INSTRUCTION SEQUENCE CODE BLOCKS BY USING VIRTUAL CORES INSTANTIATED BY PARTITIONABLE ENGINES"인, 선행 출원되고 공동 양도된 특허 출원 번호 제13428440호에서 기술된 용어 "상속 벡터들"과 동의어라는 점에 유의해야 한다.

[0028] 도 2는 본 발명의 일 실시예에 따른 레지스터 뷰, 소스 뷰, 및 명령어 뷰의 개요도를 보여준다. 이 도면은 (예컨대, 소스 뷰, 명령어 뷰, 레지스터 뷰 등을 가진) 스케줄러 아키텍처의 일 실시예를 보여준다. 앞서 언급한 구조들 중 하나 이상을 결합하거나 분할하는 것에 의해 동일한 기능을 달성하는 스케줄러 아키텍처의 다른 구현들도 가능하다.

[0029] 도 2는 레지스터 템플릿들의 동작 및 머신 상태의 유지를 지원하는 기능 엔티티들을 도시한다. 도 2의 왼쪽은 레지스터 템플릿들(T0 내지 T4)을 보여주고, 여기서 화살표들은 하나의 레지스터 템플릿/상속 벡터로부터 다음 것으로의 정보의 상속을 나타낸다. 레지스터 뷰, 소스 뷰, 및 명령어 뷰 각각은 명령어들의 블록들과 관련되는 정보를 저장하기 위한 데이터 구조들을 포함한다. 도 2는 또한 헤더를 갖는 예시적인 명령어 블록과 이 명령어 블록이 머신의 레지스터들에 대한 소스들과 목적지들 양자를 포함하는 방법을 보여준다. 블록들에 의해 참조되는 레지스터들에 관한 정보는 레지스터 뷰 데이터 구조에 저장된다. 블록들에 의해 참조되는 소스들에 관한 정보는 소스 뷰 데이터 구조에 저장된다. 블록들에 의해 참조되는 명령어들 자체에 관한 정보는 명령어 뷰 데이터 구조에 저장된다. 레지스터 템플릿들/상속 벡터들 자체는 블록들에 의해 참조되는 종속성 및 상속 정보를 저장하는 데이터 구조들을 포함한다.

[0030] 도 3은 본 발명의 일 실시예에 따른 예시적인 레지스터 템플릿과 이 레지스터 템플릿으로부터의 정보에 의해 소스 뷰가 채워지는 방법을 설명하는 도면을 보여준다.

[0031] 본 실시예에서, 소스 뷰의 목표는 특정 블록들이 디스패치될 수 있는 때를 결정하는 것이라는 점에 유의해야 한다. 블록이 디스패치될 때 그것은 그의 블록 번호를 모든 나머지 블록들에 브로드캐스팅한다. 다른 블록들(예컨대, 비교)의 소스들에 대한 임의의 일치(match)들은 준비 비트(예컨대, 또는 임의의 다른 유형의 표시자)가 설정되게 한다. 모든 준비 비트들이 설정되면(예컨대, AND 게이트) 블록은 디스패치될 준비가 된다. 블록들이 그것들이 종속하는 다른 블록들의 준비에 기초하여 디스패치된다.

[0032] 다수의 블록이 디스패치를 위한 준비가 되면, 가장 오래된 블록이 더 신생의(younger) 블록들에 앞서 디스패치를 위해 선택된다. 예를 들어, 일 실시예에서 처음 찾기 회로(find first circuit)를 이용하여 커밋 포인터에의 근접에 기초하여 가장 오래된 블록을 찾고 커밋 포인터에의 상대적 근접에 기초하여 후속 블록들을 찾을 수 있다(예컨대, 각 블록의 준비 비트에 대해 작업하여).

[0033] 다시 도 3을 참고하면, 이 예에서, 블록 20이 도착할 때 생성된 레지스터 템플릿 스냅샷이 검사되고 있다. 전

술한 바와 같이, 레지스터 템프릿은 R0에서 R63까지 각 레지스터에 대한 필드들을 갖는다. 블록들은 그 각자의 블록 번호들을 블록 목적지들에 대응하는 레지스터 템플릿 필드들에 기입한다. 각각의 블록은 레지스터 템플릿으로부터 그의 레지스터 소스들을 대표하는 레지스터 필드들을 판독한다. 첫 번째 번호는 레지스터에 기입한 블록이고 두 번째 번호는 그 블록의 목적지 번호이다.

[0034] 예를 들어, 블록 20이 도착할 때, 그것은 레지스터 템플릿의 스냅샷을 판독하고 레지스터 템플릿에서 자신의 레지스터 소스들을 조회하여 그의 소스들 각각에 기입한 마지막 블록을 결정하고 그의 목적지들이 이전 레지스터 템플릿 스냅샷에 행하는 업데이트들에 따라 소스 뷰를 채운다. 후속 블록들은 그 자신의 목적지들로 레지스터 템플릿을 업데이트할 것이다. 이것은 도 3의 하부 좌측에 도시되어 있는데, 여기서 블록 20은 그의 소스들: 소스 1, 소스 2, 소스 3, 내지 소스 8을 채운다.

[0035] 도 4는 소스 뷰 내의 종속성 브로드캐스팅을 위한 제1 실시예를 설명하는 도면을 보여준다. 이 실시예에서, 각 열은 명령어 블록을 포함한다. 블록이 할당될 때 그것은 그것의 소스들이 그 블록들에 대한 종속성을 가질 때마다 모든 블록의 열들에 (예컨대, 0을 기입함으로써) 표시한다. 임의의 다른 블록이 디스패치될 때 그의 번호는 그 블록과 관련되는 정확한 열을 가로질러 브로드캐스팅된다. 1을 기입하는 것은 그 블록에 대한 종속성이 없음을 나타내는 디폴트 값이라는 점에 유의해야 한다.

[0036] 블록 내의 모든 준비 비트들이 준비되어 있을 때, 그 블록은 디스패치되고 그것의 번호는 모든 나머지 블록들에 브로드캐스팅된다. 블록 번호는 다른 블록들의 소스들에 저장된 모든 번호들과 비교한다. 일치가 있다면, 그 소스에 대한 준비 비트는 설정된다. 예를 들어, 소스 1에서 브로드캐스팅된 블록 번호가 11이면 블록 20의 소스 1에 대한 준비 비트는 설정될 것이다.

[0037] 도 5는 소스 뷰 내의 종속성 브로드캐스팅을 위한 제2 실시예를 설명하는 도면을 보여준다. 이 실시예는 블록들에 의해 조직되는 것과 대조적으로 소스들에 의해 조직된다. 이것은 소스 뷰 데이터 구조에 걸쳐서 소스들 S1 내지 S8에 의해 도시되어 있다. 상기 도 4와 관련해 기술된 것과 유사한 방식으로, 도 5의 실시예에서, 블록 내의 모든 준비 비트들이 준비되어 있을 때, 그 블록은 디스패치되고 그것의 번호는 모든 나머지 블록들에 브로드캐스팅된다. 블록 번호는 다른 블록들의 소스들에 저장된 모든 번호들과 비교한다. 일치가 있다면, 그 소스에 대한 준비 비트는 설정된다. 예를 들어, 소스 1에서 브로드캐스팅된 블록 번호가 11이면 블록 20의 소스 1에 대한 준비 비트는 설정될 것이다.

[0038] 도 5의 실시예는 또한 커밋 포인터와 할당 포인터 사이의 블록들에서만 비교들이 가능하게 되는 방법을 보여준다. 모든 다른 블록들은 무효하다.

[0039] 도 6은 본 발명의 일 실시예에 따른 커밋 포인터로부터 시작하여 대응하는 포트 할당들을 브로드캐스팅하는 디스패치를 위한 준비 블록들의 선택을 설명하는 도면을 보여준다. 소스 뷰 데이터 구조는 도 6의 왼쪽에 도시되어 있다. 명령어 뷰 데이터 구조는 도 6의 오른쪽에 도시되어 있다. 소스 뷰와 명령어 뷰 사이에 선택기 어레이가 도시되어 있다. 이 실시예에서, 선택기 어레이는 4개의 디스패치 포트(P1 내지 P4)를 통해 사이클마다 4개의 블록을 디스패치한다.

[0040] 전술한 바와 같이, 블록들은 커밋 포인터로부터 랩 어라운드(wrap around)하여 할당 포인터까지(예컨대, 오래된 블록들을 먼저 디스패치하는 것을 지키려고 하면서) 디스패치를 위해 선택된다. 선택기 어레이는 커밋 포인터로부터 시작하여 처음 4개의 준비 블록을 찾기 위해 이용된다. 가장 오래된 준비 블록들을 디스패치하는 것이 바람직하다. 일 실시예에서, 선택기 어레이는 덧셈기 트리 구조를 이용하여 구현될 수 있다. 이것은 아래 도 7에서 설명될 것이다.

[0041] 도 6은 또한 명령어 뷰 내의 엔트리들을 통과한 4개의 포트 각각에 선택기 어레이가 결합되는 방법을 보여준다. 이 실시예에서, 포트 인에이블들로서 포트 결합들은 4개의 포트 중 하나가 활성화될 수 있게 하고 해당 명령어 뷰 엔트리가 아래로 디스패치 포트까지 그리고 계속해서 실행 유닛들까지 통과하게 한다. 게다가, 전술한 바와 같이, 디스패치된 블록들은 소스 뷰를 통하여 브로드캐스트 백(broadcast back) 된다. 디스패치를 위한 선택된 블록들의 블록 번호들은 브로드캐스트 백 된다(4까지). 이것은 도 6의 가장 오른쪽에 도시되어 있다.

[0042] 도 7은 본 발명의 일 실시예에 따른 도 6에 설명된 선택기 어레이를 구현하는 데 이용되는 덧셈기 트리 구조를 보여준다. 도시된 덧셈기 트리는 선택기 어레이의 기능을 구현한다. 덧셈기 트리는 처음 4개의 준비 블록을 고르고 이들을 디스패치를 위한 4개의 이용 가능한 포트(예컨대, 판독 포트 1 내지 판독 포트 4)에 올려놓는다. 어떤 중재도 이용되지 않는다. 특정 포트를 구체적으로 인에이블하기 위해 이용되는 실제 로직은 엔트리 번호 1에 명백히 도시되어 있다. 명확성을 위해, 그 로직은 다른 엔트리들에서는 구체적으로 도시되어 있지 않다.

이렇게 하여, 도 7은 블록 디스패치를 위한 각 특정 포트의 직접 선택이 구현되는 방법에 대한 하나의 구체적인 실시예를 보여준다. 그러나, 대안적으로, 우선 순위 인코더들을 이용하는 실시예가 구현될 수 있다는 점에 유의해야 한다.

[0043] 도 8은 선택기 어레이 덧셈기 트리의 예시적인 로직을 보다 상세히 보여준다. 도 8의 실시예에서는, 범위 초과 비트(range exceed bit)에 대한 로직이 도시되어 있다. 범위 초과 비트는 4개 초과의 블록이 디스패치를 위해 선택되지 않도록 보장하고, 제5 블록이 준비되어 있다면 범위 초과 비트는 처음 4개의 블록도 준비되어 있다면 제5 블록이 디스패치되지 않게 할 것이다. 합계 비트들은 S0 내지 S3이고 직렬 구현에서 다음 덧셈기 스테이지로 전파뿐만 아니라 디스패치 포트를 인에이블하기 위해 이용된다는 점에 유의해야 한다.

[0044] 도 9는 본 발명의 일 실시예에 따른 선택기 어레이를 구현하기 위한 덧셈기 트리의 병렬 구현을 보여준다. 이 병렬 구현은 각 덧셈기로부터의 합계를 다음 것으로 전달하지 않는다. 병렬 구현에서, 각 덧셈기는 다중 입력 캐리 세이브 덧셈기 트리(multi-input carry save adder trees)와 같은 다중 입력 덧셈 구현을 이용하여 모든 그것의 필요한 입력들을 직접 이용한다. 예를 들어, 덧셈기 "X"는 이전의 입력들 모두를 합산한다. 이 병렬 구현은 보다 빠른 계산 시간들(예컨대, 단일 사이클)을 실행하기 위하여 바람직하다.

[0045] 도 10은 본 발명의 일 실시예에 따른 도 9로부터의 덧셈기 X가 캐리 세이브 덧셈기들을 이용하여 구현될 수 있는 방법을 설명하는 예시도를 보여준다. 도 10은 단일 사이클에 32개 입력을 덧셈할 수 있는 구조를 보여준다. 이 구조는 4×2 캐리 세이브 덧셈기들(4-by-2 carry save adders)을 이용하여 합해진다.

[0046] 도 11은 본 발명의 일 실시예에 따른 커밋 포인터로부터 시작하여 선택기 어레이 덧셈기들을 이용하여 스케줄링하기 위해 준비 비트들을 마스킹하기 위한 마스킹 실시예를 보여준다. 이 구현에서, 선택기 어레이 덧셈기들은 커밋 포인터에서 시작하여 잠재적으로 랩 어라운드하여 할당 포인터까지 디스패치할 처음 4개의 준비 블록을 선택하려고 하고 있다. 이 구현에서, 다중 입력 병렬 덧셈기들이 이용된다. 게다가, 이 구현에서 이들 원형 버퍼의 소스가 이용된다.

[0047] 도 11은 준비 비트들이 2개의 마스크 각각과 함께 AND 연산되고(개별적으로 또는 별도로) 2개의 덧셈기 트리에 병렬로 적용되는 방법을 보여준다. 처음 4개의 블록은 2개의 덧셈기 트리를 이용하여 그리고 4의 임계값과 비교함으로써 선택된다. "X" 마크들은 "해당 덧셈기 트리에 대한 선택 어레이로부터 배제함"을 나타내고 따라서 "X" 값은 0이다. 한편 "Y" 마크들은 "해당 덧셈기 트리에 대한 선택 어레이에 포함함"을 나타내고 따라서 "Y" 값은 1이다.

[0048] 도 12는 본 발명의 일 실시예에 따른 레지스터 뷰 엔트리들이 레지스터 템플릿들에 의해 채워지는 방법의 개요도를 보여준다.

[0049] 전술한 바와 같이, 레지스터 뷰 엔트리들은 레지스터 템플릿들에 의해 채워진다. 레지스터 뷰는 시퀀스 내의 각 블록에 대한 레지스터 템플릿들의 스냅샷들을 저장한다. 추측(speculation)이 유효하지 않을 때(예컨대, 분기 예측 오류), 레지스터 뷰는 무효 추측 지점 전의 최근 유효 스냅샷을 가진다. 머신은 그 레지스터 뷰 엔트리를 판독하고 그것을 레지스터 템플릿의 베이스에 로딩함으로써 그의 상태를 마지막 유효 스냅샷으로 롤 백(roll back) 할 수 있다. 레지스터 뷰의 각 엔트리는 모든 레지스터 상속 상태들을 보여준다. 예를 들어 도 12의 실시예에서, 블록 F에 대한 레지스터 뷰가 무효하다면, 머신 상태는 보다 이른 마지막 유효 레지스터 템플릿 스냅샷으로 롤 백될 수 있다.

[0050] 도 13은 본 발명의 일 실시예에 따른 감소된 레지스터 뷰 풋프린트에 대한 제1 실시예를 보여준다. 레지스터 뷰 엔트리들을 저장하는 데 필요한 메모리의 양은 분기 명령어들을 포함하는 레지스터 뷰 템플릿 스냅샷들만을 저장함으로써 감소될 수 있다. 예외가 발생할 때(예컨대, 추측이 유효하지 않음, 분기 예측 오류, 등등), 마지막 유효 스냅샷은 예외에 앞서 발생한 분기 명령어로부터 재구축될 수 있다. 마지막 유효 스냅샷을 구축하기 위하여 예외 이전의 분기로부터 예외까지 명령어들이 페치된다. 명령어들은 페치되지만 이들은 실행되지 않는다. 도 13에 도시된 바와 같이, 분기 명령어들을 포함하는 스냅샷들만이 감소된 레지스터 뷰에 저장된다. 이것은 레지스터 템플릿 스냅샷들을 저장하는 데 필요한 메모리의 양을 크게 감소시킨다.

[0051] 도 14는 본 발명의 일 실시예에 따른 감소된 레지스터 뷰 풋프린트에 대한 제2 실시예를 보여준다. 레지스터 뷰 엔트리들을 저장하는 데 필요한 메모리의 양은 스냅샷들의 순차적 서브세트(예컨대, 모든 4개의 스냅샷 중 하나)만을 저장함으로써 감소될 수 있다. 연속적인 스냅샷들 간의 변화는 전체 연속적인 스냅샷들보다 비교적 적은 양의 메모리를 이용하여 원래 스냅샷으로부터 "델타"로서 저장될 수 있다. 예외가 발생할 때(예컨대, 추측이 유효하지 않음, 분기 예측 오류, 등등), 마지막 유효 스냅샷은 예외에 앞서 발생한 원래 스냅샷으로부터

재구축될 수 있다. 예외에 앞서 발생한 원래 스냅샷으로부터의 "델타" 및 연속적인 스냅샷들은 마지막 유효 스냅샷을 재구축하는 데 이용된다. 초기 원래 상태는 요구되는 스냅샷의 상태에 도달하기 위해 델타들을 누적할 수 있다.

[0052] 도 15는 본 발명의 일 실시예에 따른 스냅샷들 간의 델타의 예시적인 포맷을 보여준다. 도 15는 원래 스냅샷과 2개의 델타를 보여준다. 하나의 델타에서, R5와 R6는 B3에 의해 업데이트되고 있는 유일한 레지스터들이다. 엔트리들의 나머지는 변경되지 않는다. 다른 델타에서, R1과 R7은 B2에 의해 업데이트되고 있는 유일한 레지스터들이다. 엔트리들의 나머지는 변경되지 않는다.

[0053] 도 16은 본 발명의 일 실시예에 따른 명령어 블록들의 할당들에 따라 레지스터 템플릿 스냅샷들을 생성하기 위한 프로세스의 도면을 보여준다. 이 실시예에서, 도 16의 왼쪽은 2개의 디멀티플렉서(de-multiplexer)를 보여주고, 도 16의 최상부에는 스냅샷 레지스터 템플릿이 있다. 도 16은 이전 레지스터 템플릿으로부터 후속 레지스터 템플릿을 생성하기 위한 도면을 보여준다(예컨대, 직렬 구현).

[0054] 이 직렬 구현은 명령어 블록들의 할당에 따라 레지스터 템플릿 스냅샷들이 생성되는 방법을 보여준다. 그 스냅샷들은 (예컨대, 도 1 내지 도 4에 기술된 바와 같이) 종속성 추적뿐만 아니라 (예컨대, 도 12 내지 도 15에 기술된 바와 같이) 예측 오류들/예외들을 처리하기 위해 레지스터 뷰를 업데이트하는 데 이용되는 최근 레지스터 아키텍처 상태들의 업데이트를 캡처하는 역할을 한다.

[0055] 디멀티플렉서(de-mux)는 어느 착신 소스가 전달되는지를 선택함으로써 기능한다. 예를 들어, 레지스터 R2는 제2 출력에서 1로 디멀티플렉싱할 것이고, R8은 제7 출력에서 1로 디멀티플렉싱할 것이고, 기타 등등이다.

[0056] 도 17은 본 발명의 일 실시예에 따른 명령어 블록들의 할당들에 따라 레지스터 템플릿 스냅샷들을 생성하기 위한 프로세스의 다른 도면을 보여준다. 도 17의 실시예는 또한 이전 레지스터 템플릿으로부터 후속 레지스터 템플릿을 생성하는 것을 보여준다. 도 17의 실시예는 또한 레지스터 템플릿 블록 상속의 예를 보여준다. 이 도면은 레지스터 템플릿이 할당된 블록 번호들로부터 업데이트되는 방법의 예를 보여준다. 예를 들어, 블록 Bf는 R2, R8, 및 R10을 업데이트한다. Bg는 R1 및 R9를 업데이트한다. 점선 화살표들은 값들이 이전 스냅샷으로부터 상속되는 것을 나타낸다. 이 프로세스는 블록 Bi까지 아래로 계속 진행된다. 따라서, 예를 들어, 어떤 스냅샷도 레지스터 R7을 업데이트하지 않았으므로, 그것의 원래 값 Bb는 맨 아래까지 전파될 것이다.

[0057] 도 18은 본 발명의 일 실시예에 따른 이전 레지스터 템플릿으로부터 후속 레지스터 템플릿을 생성하는 직렬 구현을 구현하기 위한 하드웨어의 개요도를 보여준다. 디멀티플렉서는 2개의 블록 번호 중 어느 것이 아래로 다음 스테이지로 전파될 것인지 일련의 2개의 입력 멀티플렉서를 제어하는 데 이용된다. 그것은 이전 스테이지로부터의 블록 번호 또는 현재 블록 번호일 수 있다.

[0058] 도 19는 본 발명의 일 실시예에 따른 이전 레지스터 템플릿으로부터 후속 레지스터 템플릿을 생성하는 병렬 구현을 구현하기 위한 하드웨어의 개요도를 보여준다. 이 병렬 구현은 이전 레지스터 템플릿으로부터 후속 레지스터 템플릿을 생성하기 위해 특수한 인코딩된 멀티플렉서 제어들을 이용한다.

[0059] 도 20은 본 발명의 일 실시예에 따른 명령어 블록 기반 실행을 위한 하드웨어와, 그것이 소스 뷰, 명령어 뷰, 레지스터 템플릿들, 및 레지스터 뷰와 함께 동작하는 방법에 대한 개요도를 보여준다.

[0060] 이 구현에서, 디스패처 내의 할당기 스케줄러는 머신의 프런트 엔드에 의해 페치된 명령어들을 수신한다. 이들 명령어는 앞서 설명한 방식으로 블록 형성을 거친다. 앞서 설명한 바와 같이 블록들은 레지스터 템플릿들을 생성하고 이들 레지스터 템플릿은 레지스터 뷰를 채우는 데 사용된다. 소스 뷰로부터 소스들은 레지스터 파일 계층 구조에 전송되고 전술한 방식으로 소스 뷰로의 브로드캐스트들이 존재한다. 명령어 뷰는 명령어들을 실행 유닛들에 전송한다. 명령어들은 레지스터 파일 계층 구조로부터 오는 명령어들에 의해 요구되는 소스들로서 실행 유닛들에 의해 실행된다. 그 후 이들 실행된 명령어는 실행 유닛에서 벗어나서 레지스터 파일 계층 구조로 전송된다.

[0061] 도 21은 본 발명의 일 실시예에 따른 청킹 아키텍처의 예를 보여준다. 청킹의 중요성은 그것이 도시된 4개의 멀티플렉서를 이용함으로써 각 스케줄러 엔트리로의 기입 포트의 수를 4에서 1로 줄이면서도, 버블들을 형성하지 않고 모든 엔트리들을 조밀하게 패킹한다는 것이다.

[0062] 청킹의 중요성은 다음의 예에 의해 관찰될 수 있다(예컨대, 각 사이클에서 블록들의 할당이 최상부 위치, 이 경우 B0에서 시작된다는 것에 주목한다). 사이클 1에서, 3개의 명령어 블록이 스케줄러 엔트리들에 할당되어야 하는 것으로 가정한다(예컨대, 3개의 블록은 스케줄러에서 처음 3개의 엔트리를 차지할 것이다). 다음 사이클

(예컨대, 사이클 2)에서 또 다른 2개의 명령어 블록이 할당되어야 한다. 스케줄러 어레이 엔트리들에서 버블들을 생성하는 것을 피하기 위해, 스케줄러 어레이 엔트리들은 4개의 기입 포트를 지원하도록 구축되어야 한다. 이것은 전력 소비, 타이밍, 면적 등의 관점에서 비용이 많이 든다. 상기 청킹 구조는 어레이들에 할당하기 전에 멀티플렉싱 구조를 이용함으로써 하나의 기입 포트만을 갖도록 모든 스케줄러 어레이들을 단순화한다. 상기 예에서, 사이클 2에서 B0는 마지막 멀티플렉서(mux)에 의해 선택될 것이고 사이클 2에서 B1은 첫 번째 멀티플렉서에 의해 선택될 것이다(예컨대, 왼쪽에서 오른쪽으로 진행한다).

[0063] 이렇게 하여, 엔트리 청크에 대해 각각은 엔트리당 하나의 기입 포트와 엔트리당 4개의 판독 포트만을 필요로 한다. 멀티플렉서들이 구현되어야 하기 때문에 비용에서 트레이드오프가 있지만, 그 비용은 매우 많은 엔트리들이 있을 수 있으므로 엔트리당 4개의 기입 포트를 구현하지 않아도 되는 것으로부터 여러 배 절약된다.

[0064] 도 21은 또한 중간 할당 버퍼를 보여준다. 스케줄러 어레이들이 그것들에 송신된 모든 청크들을 수용할 수 없다면, 그것들은 중간 할당 버퍼에 임시로 저장될 수 있다. 스케줄러 어레이들이 자유 공간을 가질 때, 청크들은 중간 할당 버퍼로부터 스케줄러 어레이들로 전송될 것이다.

[0065] 도 22는 본 발명의 일 실시예에 따른 스레드들이 그들의 블록 번호들 및 스레드 ID에 따라 할당되는 방법에 대한 묘사를 보여준다. 블록들은 전술한 바와 같은 청킹 구현을 통해 스케줄러 어레이에 할당된다. 스레드 블록들 각각은 블록 번호를 이용하여 그 자신들 사이에 순차적 순서를 유지한다. 상이한 스레드들로부터의 블록들은 인터리빙될 수 있다(예컨대, 스레드 Th1에 대한 블록들과 스레드 Th2에 대한 블록들은 스케줄러 어레이에서 인터리빙된다). 이렇게 하여, 상이한 스레드들로부터의 블록들은 스케줄러 어레이 내에 존재한다.

[0066] 도 23은 본 발명의 일 실시예에 따른 멀티스레드 실행을 관리하기 위하여 물리적 저장 위치들을 가리키는 스레드 기반 포인터 맵들을 이용하는 스케줄러의 구현을 보여준다. 이 실시예에서, 스레드들의 관리는 스레드 맵들의 제어를 통하여 구현된다. 예를 들어 여기서 도 23은 스레드 1 맵과 스레드 2 맵을 보여준다. 이 맵들은 개별 스레드의 블록들의 위치를 추적한다. 맵 .2 물리적 저장 위치들 내의 엔트리들 맵 내의 엔트리들은 그 스레드에 속하는 블록들에 할당된다. 이 구현에서, 각각의 스레드는 양쪽 스레드들에 대해 카운트하는 할당 카운터를 가진다. 전체 카운트는 N을 2로 나눈 값을 초과할 수 없다(예컨대, 이용 가능한 공간 초과). 할당 카운터들은 풀(pool)로부터의 전체 엔트리들의 할당에서 공정성을 구현하기 위해 조절 가능한 임계값들을 가진다. 할당 카운터들은 하나의 스레드가 이용 가능한 공간 모두를 이용하는 것을 방지할 수 있다.

[0067] 도 24는 본 발명의 일 실시예에 따른 스레드 기반 포인터 맵들을 이용하는 스케줄러의 다른 구현을 보여준다. 도 24는 커밋 포인터와 할당 포인터 사이의 관계를 보여준다. 도시된 바와 같이, 각각의 스레드는 커밋 포인터와 할당 포인터를 갖고 화살표는 스레드 2에 대한 리얼리티 포인터가 물리적 저장소를 랩 어라운드하여 블록들 B1 및 B2를 할당할 수 있지만, 스레드 2에 대한 커밋 포인터가 아래로 이동할 때까지는 블록 B9를 할당할 수 없는 방법을 보여준다. 이것은 스레드 2의 커밋 포인터의 위치와 스트라이크스루(strikethrough)에 의해 도시되어 있다. 도 24의 오른쪽은 커밋 포인터가 시계 반대 방향으로 이동함에 따라 커밋 포인터와 블록들의 할당 간의 관계를 보여준다.

[0068] 도 25는 본 발명의 일 실시예에 따른 스레드들에 대한 실행 리소스들의 동적인 캘린더 기반 할당에 대한 도면을 보여준다. 각 스레드의 순방향 진행에 기초하여 할당 카운터들을 이용하여 공정성이 동적으로 제어될 수 있다. 양쪽 스레드들이 상당한 순방향 진행(substantial forward progress)을 하고 있다면, 양쪽 할당 카운터들은 동일한 임계값(예컨대, 9)으로 설정된다. 그러나 하나의 스레드가, 예를 들어 L2 캐시 부적중(cache miss) 또는 그러한 이벤트들을 겪으면서, 느린 순방향 진행을 한다면, 임계값 카운터들의 비율은 여전히 상당한 순방향 진행을 하고 있는 스레드에 유리하게 조절될 수 있다. 하나의 스레드가 스톨(stall)되거나 중단된다면(예컨대, OS 또는 IO 응답을 기다리는 대기 또는 스핀 상태에 있다면) 비율은 대기 상태의 해제를 신호하기 위해 중단된 스레드에 대해 예약되는 단일 리턴 엔트리(return entry)는 제외하고 다른 스레드에 맞추어 완전히 조절될 수 있다.

[0069] 일 실시예에서, 프로세스는 50% : 50%의 비율로 시작된다. 블록 22에 대해 L2 캐시 부적중이 검출되면, 파이프라인의 프런트 엔드는 파이프라인으로의 임의의 추가 페치 또는 스레드 2 블록들의 스케줄러로의 할당을 스톨한다. 스케줄러로부터 스레드 2 블록들이 리타이어되면, 그 엔트리들은 스레드 할당의 새로운 동적 비율이 달성되는 지점까지는 스레드 1 할당을 위해 이용 가능하게 될 것이다. 예를 들어, 최근에 리타이어된 스레드 2 블록들 중 3개가 스레드 2 대신에 스레드 1로의 할당을 위해 풀에 리턴될 것이고, 따라서 스레드 1 대 스레드 2 비율은 75% : 25%가 된다.

[0070] 파이프라인의 전방에서 스레드 2 블록들의 스톨은 (예컨대, 스톨된 스레드 2 블록들을 통과함으로써 스레드 1 블록들에 의해) 그것들을 우회할 어떤 하드웨어 메커니즘도 없다면 파이프라인의 전방으로부터 그 블록들을 플러싱하는 것을 요구할지도 모른다는 점에 유의해야 한다.

[0071] 도 26은 본 발명의 일 실시예에 따른 이중 디스패치 프로세스를 도시한다. 다중 디스패치는 일반적으로 그 블록을 가진 상이한 명령어들이 실행 유닛들을 통한 각 패스에서 실행할 수 있도록 (다수의 명령어를 안에 가진) 블록을 여러 번 디스패치하는 것을 포함한다. 하나의 예는 결과의 데이터를 소비하는 후속 디스패치가 뒤따르는 어드레스 계산 명령어의 디스패치일 것이다. 또 다른 예는, 부동 소수점 연산일 것이고, 여기서 제1 부분은 고정점 연산으로서 실행되고 제2 부분은 반올림(rounding), 플래그 생성/계산, 지수 조정(exponent adjustment) 또는 기타 유사한 것을 수행함으로써 연산을 완료하도록 실행된다. 블록들은 단일 엔티티로서 자동으로 할당되고, 커밋되고, 리타이어된다.

[0072] 다중 디스패치의 주요 이점은 다수의 개별 블록들을 머신 윈도(machine window)에 할당하는 것을 피하고, 그렇게 함으로써 머신 윈도를 효과적으로 더 크게 만든다는 것이다. 더 큰 머신 윈도는 최적화 및 순서 재정렬을 위한 더 많은 기회를 의미한다.

[0073] 도 26의 하부 왼쪽을 보면, 명령어 블록이 도시되어 있다. 이 블록은 단일 사이클에서 디스패치될 수 없는데, 그 이유는 로드 어드레스 계산과 캐시들/메모리로부터 데이터를 리턴하는 로드 사이에 대기 시간이 있기 때문이다. 그래서 이 블록은 먼저 디스패치되고 그의 중간 결과는 일시적 상태로서 유지된다(그것의 결과는 아키텍처 상태에게 보이지 않은 채로 두 번째 디스패치로 그때그때 상황에 따라(on the fly) 전달된다). 첫 번째 디스패치는 LA의 디스패치 및 어드레스 계산에 이용되는 2개의 컴포넌트 1 및 2를 송신한다. 두 번째 디스패치는 캐시들/메모리로부터 데이터를 리턴하는 로드 시에 로드 데이터의 실행 부분들인 컴포넌트 3 및 4를 송신한다.

[0074] 도 26의 하부 오른쪽을 보면 부동 소수점 곱셈 누산 연산이 도시되어 있다. 하드웨어가 단일 위상에서 연산을 디스패치하기에 충분한 착신 소스들의 대역폭을 가지고 있지 않은 경우에는, 곱셈 누산 도면이 보여주는 바와 같이, 이중 디스패치가 이용된다. 첫 번째 디스패치는 도시된 바와 같은 고정점 곱셈이다. 두 번째 디스패치는 도시된 바와 같은 부동 소수점 덧셈 반올림이다. 이들 디스패치된 명령어들 양자가 실행될 때, 그것들은 부동 소수점 곱셈/누산을 효과적으로 수행한다.

[0075] 도 27은 본 발명의 일 실시예에 따른 이중 디스패치 일시적 곱셈-누산을 도시한다. 도 27에 도시된 바와 같이, 첫 번째 디스패치는 정수 32 비트 곱셈이고, 두 번째 디스패치는 정수 누산 덧셈이다. 첫 번째 디스패치와 두 번째 디스패치 사이에 전달되는 상태(곱셈의 결과)는 일시적이고 아키텍처적으로 보이지 않는다. 일 구현에서의 일시적 저장은 둘 이상의 곱셈기의 결과들을 유지할 수 있고 대응하는 곱셈 누산 쌍을 식별하기 위해 이들을 태그(tag)할 수 있고, 그렇게 함으로써 다수의 곱셈 누산 쌍들의 혼합이 임의의 방식(예컨대, 인터리빙 등)으로 디스패치되는 것을 가능하게 한다.

[0076] 다른 명령어들은 그들의 구현(예컨대, 부동 소수점 등)을 위해 이 동일한 하드웨어를 이용할 수 있다는 점에 유의한다.

[0077] 도 28은 본 발명의 일 실시예에 따른 이중 디스패치 아키텍처적으로 보이는 상태 곱셈-덧셈을 도시한다. 첫 번째 디스패치는 단정도(single precision) 곱셈이고, 두 번째 디스패치는 단정도 덧셈이다. 이 구현에서, 첫 번째 디스패치와 두 번째 디스패치 사이에 전달되는 상태 정보(예컨대, 곱셈의 결과)는 아키텍처적으로 보이는데, 그 이유는 이 저장은 아키텍처 상태 레지스터이기 때문이다.

[0078] 도 29는 본 발명의 일 실시예에 따른 그룹화된 실행 유닛들의 프로세스에서의 실행을 위한 명령어 블록들의 페치 및 형성의 개요도를 보여준다. 본 발명의 실시예들은 명령어들이 페치되고 하드웨어 또는 동적 변환기/JIT에 의해 블록들로서 형성되는 프로세스를 이용한다. 블록들 내의 명령어들은 블록 내의 이른 명령어(early instruction)의 결과가 블록 내의 후속 명령어의 소스에 공급되도록 조직된다. 이것은 명령어들의 블록에서 점선 화살표들로 도시되어 있다. 이 속성은 블록이 실행 블록의 스택된 실행 유닛들에서 효율적으로 실행되는 것을 가능하게 한다. 명령어들은 또한, 예를 들어 그것들이 동일한 소스를 공유하는 경우와 같이(이 도면에는 명시적으로 도시되지 않음), 그것들이 병렬로 실행될 수 있더라도 그룹화될 수 있다.

[0079] 블록들을 하드웨어로 형성하는 것의 하나의 대안은 이들을 소프트웨어로 (정적으로 또는 실행 시간에) 형성하는 것이고 이 경우 명령어 쌍들, 3중쌍들, 4중쌍들 등이 형성된다.

[0080] 명령어 그룹화 기능의 다른 구현들은 공동 양도된 미국 특허 8,327,115에서 찾아볼 수 있다.

[0081] 도 30은 본 발명의 일 실시예에 따른 명령어 그룹화의 예시도를 보여준다. 도 30의 실시예에서는 2개의 명령어가 제3의 보조 연산과 함께 도시되어 있다. 도 31의 왼쪽에서 명령어 블록은 상위 절반 블록/1 슬롯과 하위 절반 블록/1 슬롯을 포함한다. 최상부로부터 내려오는 수직 화살표들은 블록으로 들어오는 소스들을 나타내는 반면 하부로부터 내려가는 수직 화살표들은 메모리로 돌아가는 목적지들을 나타낸다. 도 3의 왼쪽에서 오른쪽으로 진행하여, 가능한 상이한 명령어 조합들이 도시되어 있다. 이 구현에서, 각각의 절반 블록은 3개의 소스를 수신할 수 있고 2개의 목적지를 전달할 수 있다. OP1과 OP2는 정상 연산들이다. 보조OP(AuxiliaryOP)들은 논리, 시프트, 이동, 부호 확장, 분기 등과 같은 보조 연산들이다. 블록을 2개의 절반으로 나누는 것의 이점은 각각의 절반 디스패치를 종속성 결정(dependency resolution)에 기초하여 단독으로 독립적으로 또는 다르게는 함께 하나의 블록으로서 동적으로(포트 이용을 위해 또는 리소스 제약 때문에) 갖고, 따라서 더 양호한 실행 시간들의 이용을 갖는 이점을 가능하게 하는 것이고, 동시에 2개의 절반이 하나의 블록에 대응하게 하는 것은 머신이 하나의 블록처럼 관리되도록(즉, 할당 및 리타이어먼트(retirement)에서) 2개의 절반 블록의 복잡도를 추상화하는 것을 가능하게 한다.

[0082] 도 31은 본 발명의 일 실시예에 따른 블록 스택 내의 절반 블록 쌍들이 실행 블록 유닛들에 매핑되는 방법을 보여준다. 실행 블록에 도시된 바와 같이, 각각의 실행 블록은 2개의 슬롯, 슬롯 1 및 슬롯 2를 갖는다. 그 목적은 첫 번째 절반 블록이 슬롯 1에서 실행되고 두 번째 절반 블록이 슬롯 2에서 실행되도록 블록을 실행 유닛들에 s 매핑하는 것이다. 그 목적은 각각의 절반 블록의 명령어 그룹이 다른 절반에 종속하지 않으면 2개의 절반 블록이 독립적으로 디스패치될 수 있게 하는 것이다. 최상부로부터 실행 블록으로 들어오는 쌍을 이룬 화살표들은 소스의 2개의 32비트 워드이다. 실행 블록을 떠나는 쌍을 이룬 화살표들은 목적지의 2개의 32비트 워드이다. 도 31의 왼쪽에서 오른쪽으로 진행하여, 실행 블록 유닛들에 스택될 수 있는 명령어들의 상이한 예시적인 조합들이 도시되어 있다.

[0083] 도 31의 최상부는 절반 블록들의 쌍들이 전체 블록 컨텍스트 또는 임의의 절반 블록 컨텍스트에서 실행되는 방법을 요약한다. s 실행 블록들 각각은 2개의 슬롯/절반 블록을 갖고 절반 블록들/실행 슬롯들 각각은 단일, 쌍을 이룬 또는 3중쌍을 이룬 그룹화된 연산들을 실행한다. 4개의 유형의 블록 실행 유형이 있다. 첫 번째는 병렬 절반들이다(이는 각 절반 블록이 그 자신의 소스들이 준비되어 있으면 독립적으로 실행되는 것을 가능하게 하지만 2개의 절반 블록은 양쪽 절반이 동시에 준비되어 있다면 하나의 실행 유닛에서 하나의 블록으로서 여전히 실행될 수 있다). 두 번째는 원자 병렬 절반들(atomic parallel halves)이다(이는 2개의 절반 사이에 종속성이 없기 때문에 병렬로 실행될 수 있는 절반 블록들을 언급하지만 이들은 함께 하나의 블록으로서 실행되도록 강제되는데, 그 이유는 2개의 절반 사이에 공유하는 리소스가 2개의 절반이 각 실행 블록에서 이용 가능한 리소스들의 제약 내에서 원자적으로 함께 실행되는 것이 바람직하게 또는 필요하게 만들기 때문이다). 세 번째 유형은 원자 직렬 절반들(atomic serial halves) s이다(이는 첫 번째 절반이 데이터를, 내부 저장과 함께 또는 내부 저장 없이 일시적 전송을 통하여, 두 번째 절반에 전송할 것을 요구한다). 네 번째 유형은 (이중 디스패치에서와 같이) 순차적 절반들이고 여기서 두 번째 절반은 첫 번째 절반에 종속하고 첫 번째 사이클보다 나중 사이클에서 디스패치되고 이중 디스패치 경우와 유사하게, 종속성 결정을 위해 추적되는, 외부 저장을 통하여 데이터를 전송한다.

[0084] 도 32는 본 발명의 일 실시예에 따른 제1 레벨 레지스터 파일로서 중간 블록 결과들의 저장을 묘사하는 도면을 보여준다. 각각의 레지스터 그룹은 하나의 64 비트 레지스터를 지원하기 위해 2개의 32 비트 레지스터를 이용함으로써 32 비트 결과들뿐만 아니라 64 비트 결과들이 지원될 수 있는 명령어 블록(2개의 절반 블록을 나타냄)을 나타낸다. 블록당 저장소는 가상 블록 저장소를 가정하고, 이는 상이한 블록들로부터의 2개의 절반 블록이 동일한 가상 블록 저장소에 기입할 수 있다는 것을 의미한다. 2개의 절반 블록의 결합된 결과들의 저장소가 하나의 가상 블록 저장소를 구성한다.

[0085] 도 33은 본 발명의 일 실시예에 따른 홀수/짝수 포트 스케줄러를 보여준다. 이 구현에서, 결과 저장소는 비대칭이다. 결과 저장소의 일부는 절반 블록당 3개의 64 비트 결과 레지스터인 반면 다른 것들은 절반 블록당 하나의 64 비트 결과 레지스터이지만, 대안의 구현은 절반 블록당 대칭 저장소를 이용할 수 있으며 추가로 도 32에 기술된 바와 같이 64비트 및 32비트 분할을 이용할 수도 있다. 이들 실시예에서, 저장소는 블록마다가 아니라 절반 블록마다 할당된다. 이 구현은 디스패치에 필요한 포트들을 홀수 또는 짝수로서 이용함으로써 그 포트들의 수를 감소시킨다.

[0086] 도 34는 4개의 실행 유닛이 스케줄러 어레이로부터 결과들을 수신하고 임시 레지스터 파일 세그먼트에 출력들을 기입하는 것으로 도시된 도 33의 보다 상세한 버전을 보여준다. 포트들은 짝수 및 홀수 간격을 두고 부착된다.

스케줄링 어레이의 왼쪽은 블록 번호들을 보여주고 오른쪽은 절반 블록 번호들을 보여준다.

[0087] 각각의 코어는 스케줄링 어레이로의 짝수 및 홀수 포트들을 가지며, 여기서 각각의 포트는 홀수 또는 짝수 절반 블록 위치에 연결된다. 하나의 구현에서, 짝수 포트들 및 이들의 대응하는 절반 블록들은 홀수 포트들 및 이들의 대응하는 절반 블록들과 상이한 코어에 존재할 수 있다. 다른 구현에서, 짝수 및 홀수 포트들은 이 도면에 도시된 바와 같이 다수의 상이한 코어들에 걸쳐 분산될 것이다. Mohammad Abdallah에 의해, 2012년 3월 23일에 출원되고, 그 전부가 본 명세서에 포함되는, 발명의 명칭이 "EXECUTING INSTRUCTION SEQUENCE CODE BLOCKS BY USING VIRTUAL CORES INSTANTIATED BY PARTITIONABLE ENGINES"인, 선행 출원되고 공동 양도된 특허 출원 번호 제13428440호에 기술된 바와 같이, 코어들은 물리적 코어들 또는 가상 코어들일 수 있다.

[0088] 특정 유형의 블록들에서는, 블록의 하나의 절반이 그 블록의 다른 절반과 독립적으로 디스패치될 수 있다. 다른 유형의 블록들에서는, 블록의 양쪽 절반이 동일한 실행 블록 유닛들에 동시에 디스패치될 필요가 있다. 또 다른 유형의 블록들에서는, 블록의 2개의 절반이 순차적으로(첫 번째 절반 뒤에 두 번째 절반) 디스패치될 필요가 있다.

[0089] 도 35는 본 발명의 일 실시예에 따른 게스트 플래그 아키텍처 에뮬레이션을 묘사하는 도면을 보여준다. 도 35의 왼쪽은 5개의 플래그를 가진 중앙 플래그 레지스터(centralized flag register)를 보여준다. 도 35의 오른쪽은 분산된 플래그 레지스터들을 가진 분산된 플래그 아키텍처를 보여주는데 여기서 플래그들은 레지스터 자체들 사이에 분산된다.

[0090] 아키텍처 에뮬레이션 중에, 분산된 플래그 아키텍처가 중앙 게스트 플래그 아키텍처의 거동을 에뮬레이션하는 것이 필요하다. 분산된 플래그 아키텍처는 또한 데이터 레지스터와 관련된 플래그 필드와는 대조적으로 다수의 독립적인 플래그 레지스터들을 이용하여 구현될 수 있다. 예를 들어, 데이터 레지스터들은 R0 내지 R15로서 구현될 수 있는 반면 독립적인 플래그 레지스터들은 F0 내지 F3로서 구현될 수 있다. 이 경우 그 플래그 레지스터들은 데이터 레지스터들과 직접 관련되지 않는다.

[0091] 도 36은 본 발명의 일 실시예에 따른 머신의 프런트 엔드, 스케줄러 및 실행 유닛들과 중앙 플래그 레지스터를 설명하는 도면을 보여준다. 이 구현에서, 프런트 엔드는 착신 명령어들을 그것들이 게스트 명령어 플래그들을 업데이트하는 방식에 기초하여 분류한다. 일 실시예에서, 게스트 명령어들은 4개의 네이티브(native) 명령어 유형들, T1, T2, T3, 및 T4로 분류된다. T1-T4는 각 게스트 명령어 유형이 업데이트하는 플래그 필드들이 어떤 것인지를 지시하는 명령어 유형들이다. 게스트 명령어 유형들은 이들의 유형에 기초하여 상이한 게스트 명령어 플래그들을 업데이트한다. 예를 들어, 논리 게스트 명령어들은 T1 네이티브 명령어들을 업데이트한다.

[0092] 도 37은 본 발명의 실시예들에 의해 구현되는 중앙 플래그 레지스터 에뮬레이션 프로세스에 대한 도면을 보여준다. 도 37의 액터들은 최근 업데이트 유형 테이블, 리네이밍 테이블 확장, 물리 레지스터들, 및 분산된 플래그 레지스터들을 포함한다. 이제 도 38의 순서도에 의해 도 37을 설명한다.

[0093] 도 38은 게스트 설정에서 중앙 플래그 레지스터 거동을 에뮬레이션하는 프로세스(3800)의 단계들의 순서도를 보여준다.

[0094] 단계 3801에서, 프런트 엔드/동적 변환기(하드웨어 또는 소프트웨어)는 착신 명령어들을 이들이 게스트 명령어 플래그들을 업데이트하는 방식에 기초하여 분류한다. 일 실시예에서, 게스트 명령어들은 4개의 플래그 아키텍처 유형들, T1, T2, T3, 및 T4로 분류된다. T1-T4는 각 게스트 명령어 유형이 업데이트하는 플래그 필드들이 어떤 것인지를 지시하는 명령어 유형들이다. 게스트 명령어 유형들은 이들의 유형에 기초하여 상이한 게스트 플래그들을 업데이트한다. 예를 들어, 논리 게스트 명령어들은 T1 유형 플래그들을 업데이트하고, 시프트 게스트 명령어들은 T2 유형 플래그들을 업데이트하고, 산술 게스트 명령어들은 T3 유형 플래그들을 업데이트하고, 특수 게스트 명령어들은 유형 T4 플래그들을 업데이트한다. 게스트 명령어들은 아키텍처 명령어 표현일 수 있는 반면 네이티브는 머신이 내부적으로 실행하는 것(예컨대, 마이크로코드)일 수 있다는 점에 유의해야 한다. 대안적으로, 게스트 명령어들은 에뮬레이션된 아키텍처로부터의 명령어들(예컨대, x86, 자바, ARM 코드 등)일 수 있다.

[0095] 단계 3802에서, 그 명령어 유형들이 각자의 게스트 플래그들을 업데이트하는 순서가 최근 업데이트 유형 테이블 데이터 구조에 기록된다. 일 실시예에서, 이 동작은 머신의 프런트 엔드에 의해 수행된다.

[0096] 단계 3803에서, 그 명령어 유형들이 스케줄러(할당/리네이밍 스테이지의 유효한(in-order) 부분)에 도달할 때, 스케줄러는 아키텍처 유형에 대응하는 암시적 물리 목적지(implicit physical destination)를 할당하고 그 할당

을 리네이밍/매핑 테이블 데이터 구조에 기록한다.

[0097] 그리고 단계 3804에서, 후속 게스트 명령어가 스케줄러 내의 할당/리네이밍 스테이지에 도달하고, 그 명령어가 게스트 플래그 필드들을 판독하기를 원할 때, (a) 머신은 판독을 수행하기 위해 어느 플래그 아키텍처 유형들이 액세스될 필요가 있는지를 결정한다. (b) 모든 필요한 플래그들이 (예컨대, 최근 업데이트 유형 테이블에 의해 결정되는 바와 같이) 동일한 최근 업데이트 플래그 유형에서 발견되면, 필요한 플래그들을 획득하기 위해 대응하는 물리 레지스터(예컨대, 그 최근 플래그 유형에 매핑하는 것)가 판독된다. (c) 모든 필요한 플래그들이 동일한 최근 업데이트 플래그 유형에서 발견될 수 없다면, 각 플래그는 개개의 최근 업데이트 플래그 유형에 매핑하는 대응하는 물리 레지스터로부터 판독될 필요가 있다.

[0098] 그리고 단계 3805에서, 최근 업데이트 플래그 유형 테이블에 의해 추적된 바와 같이, 마지막으로 업데이트된 그것의 최근 값을 유지하는 물리 레지스터로부터 각 플래그가 개별적으로 판독된다.

[0099] 최근 업데이트 유형이 다른 유형을 포함한다면 모든 서브세트 유형들이 슈퍼 세트 유형(super set type)의 동일한 물리 레지스터들에 매핑되어야 한다는 점에 유의해야 한다.

[0100] 리타이어먼트에서, 그 목적지 플래그 필드들은 복제된 중앙/게스트 플래그 아키텍처 레지스터와 합병된다. 복제(cloning)는 네이티브 아키텍처가 단일 레지스터 중앙 플래그 아키텍처와는 대조적으로 분산된 플래그 아키텍처를 이용한다는 사실 때문에 수행된다는 점에 유의해야 한다.

[0101] 특정 플래그 유형들을 업데이트하는 명령어들의 예들:

[0102] CF,OF,SF,ZR - 산술 명령어 및 로드/기입 플래그 명령어들

[0103] SF, ZF 및 조건부 CF - 논리들 및 시프트들

[0104] SF, ZF - 이동들/로드들, EXTR, 일부 곱셈들

[0105] ZF - POPCNT 및 STREX[P]

[0106] GE - SIMD 명령어들 ???

[0107] 특정 플래그들을 판독하는 조건들/예측들의 예들:

[0108] 0000 EQ Equal Z == 1

[0109] 0001 NE Not equal, or Unordered Z == 0

[0110] 0010 CS b Carry set, Greater than or equal, or Unordered C == 1

[0111] 0011 CC c Carry clear, Less than C == 0

[0112] 0100 MI Minus, negative, Less than N == 1

[0113] 0101 PL Plus, Positive or zero, Greater than or equal to, Unordered N == 00110 VS Overflow, Unordered V == 1

[0114] 0111 VC No overflow, Not unordered V == 0

[0115] 1000 HI Unsigned higher, Greater than, Unordered C == 1 and Z == 0

[0116] 1001 LS Unsigned lower or same, Less than or equal C == 0 or Z == 1

[0117] 1010 GE Signed greater than or equal, Greater than or equal N == V

[0118] 1011 LT Signed less than, Less than, Unordered N != V

[0119] 1100 GT Signed greater than, Greater than Z == 0 and N == V

[0120] 1101 LE Signed less than or equal, Less than or equal, Unordered Z == 1 or N != V

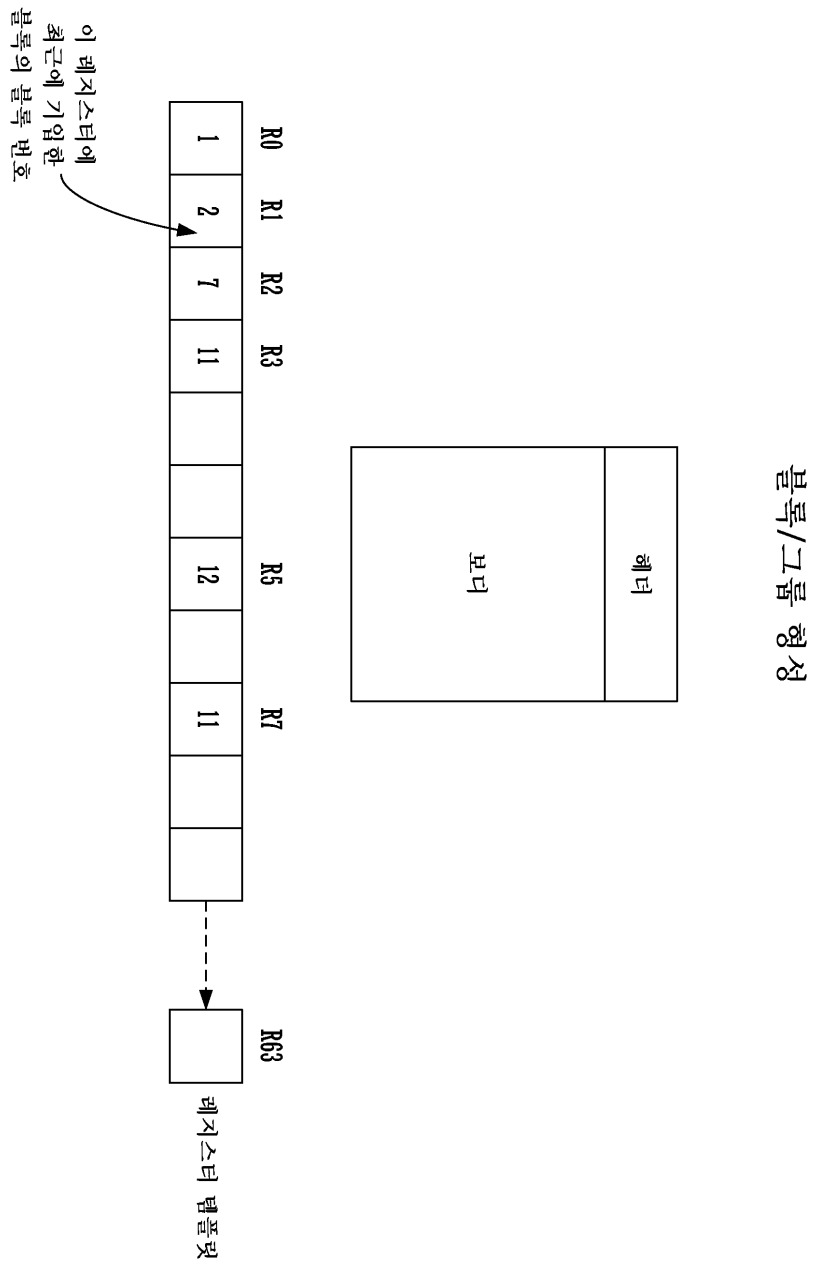
[0121] 1110 None (AL), Always (unconditional), Any flag set to any value.

[0122] 이상의 기재는, 설명을 위해, 구체적인 실시예들을 참조하여 기술되어 있다. 그렇지만, 이상의 예시된 논의는 전수적인 것으로 의도되어 있지도 않고 본 발명을 개시된 정확한 형태들로 제한하는 것으로도 의도되어 있지 않다. 이상의 개시 내용을 고려하여 많은 수정들 및 변형들이 가능하다. 본 발명의 원리들 및 그의 실제 응용들

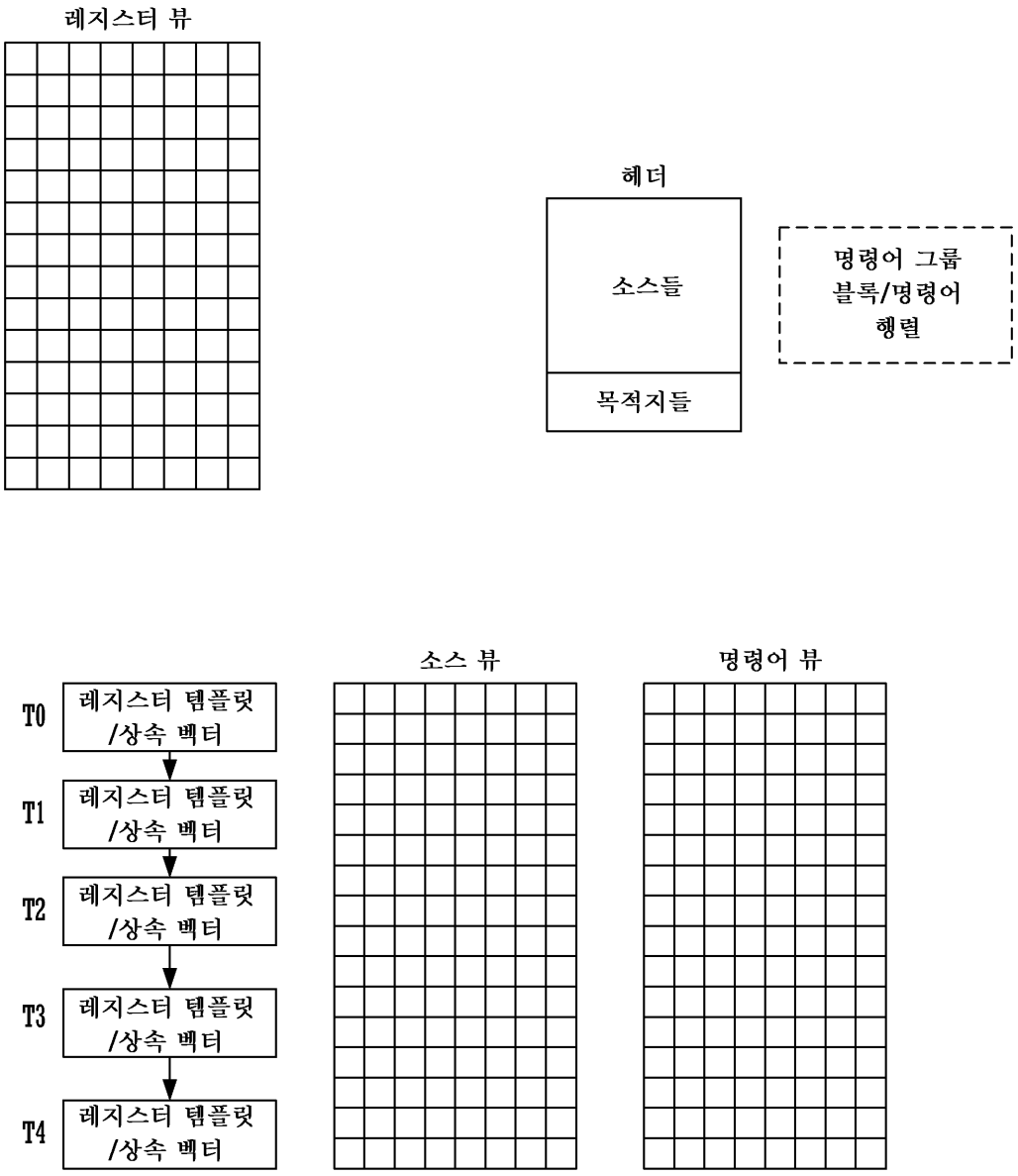
을 가장 잘 설명함으로써 다른 통상의 기술자들이 본 발명 및 생각되는 특정의 용도에 적합할 수 있는 다양한 수정들을 갖는 다양한 실시예들을 가장 잘 이용할 수 있게 해주기 위해 실시예들이 선택되어 기술되었다.

도면

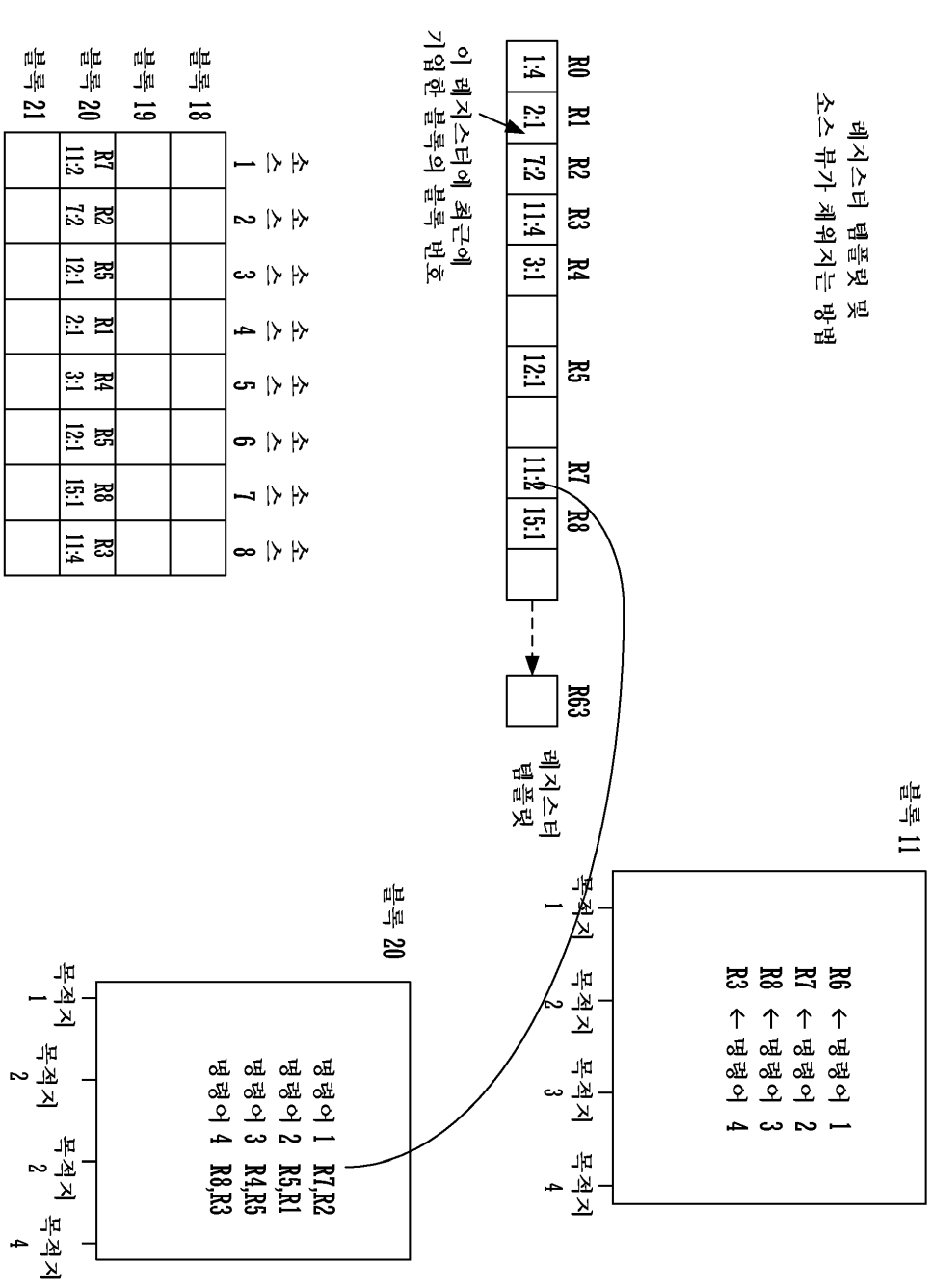
도면1



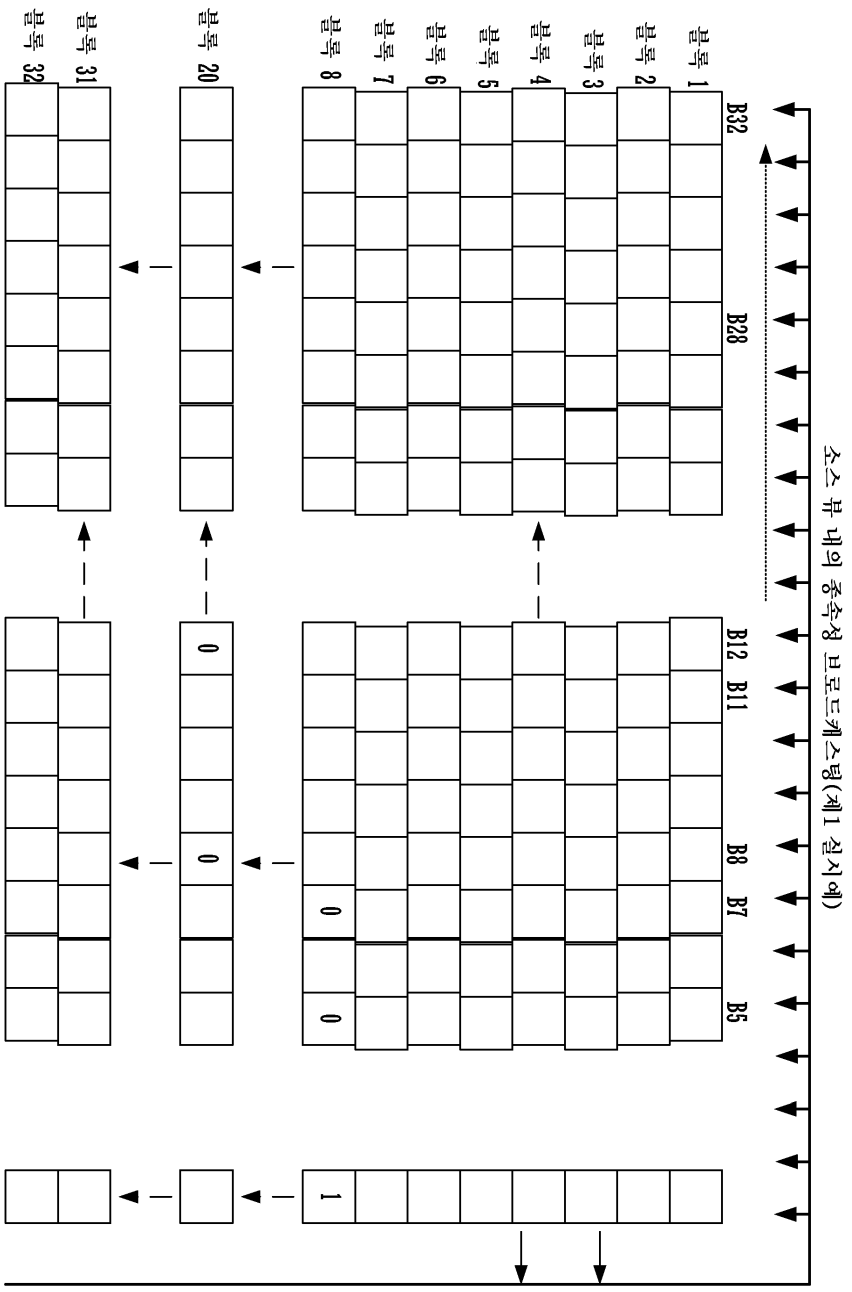
도면2


레지스터 뷰
헤더
소스들
목적지들
명령어 그룹
블록/명령어
행렬
T0
레지스터 템플릿
/상속 벡터
T1
레지스터 템플릿
/상속 벡터
T2
레지스터 템플릿
/상속 벡터
T3
레지스터 템플릿
/상속 벡터
T4
레지스터 템플릿
/상속 벡터
소스 뷰
명령어 뷰

도면3

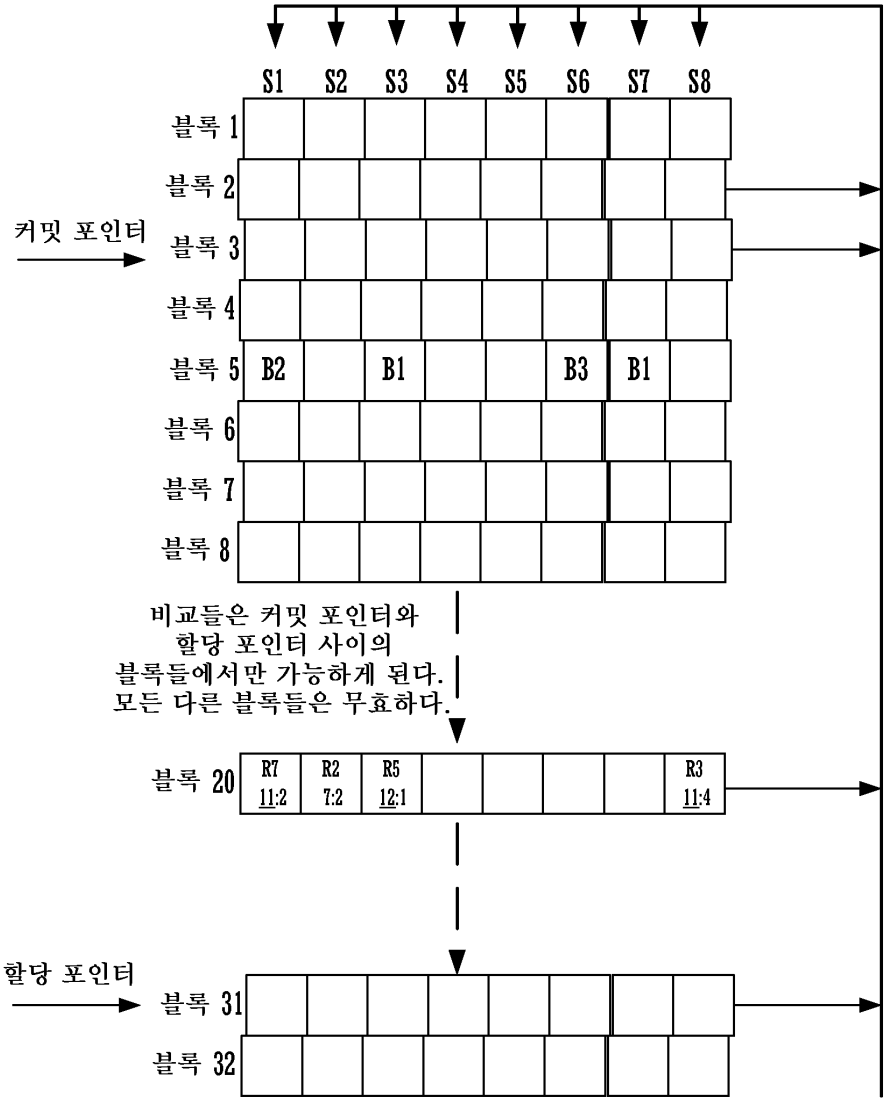

레지스터 탭플릿 및
소스 뷰가 체워지는 방법
블록 11
R6 ← 명령어 1
R7 ← 명령어 2
R8 ← 명령어 3
R3 ← 명령어 4
목적지 1
목적지 2
목적지 3
목적지 4
R0 R1 R2 R3 R4 R5 R7 R8
1:4 2:1 7:2 11:4 3:1 12:1 11:2 15:1
R63
레지스터
탭플릿
이 레지스터에 최근에
기입한 블록의 블록 번호
블록 20
명령어 1 R7,R2
명령어 2 R5,R1
명령어 3 R4,R5
명령어 4 R8,R3
목적지 1
목적지 2
목적지 2
목적지 4
소스 1
소스 2
소스 3
소스 4
소스 5
소스 6
소스 7
소스 8
블록 18
블록 19
블록 20
블록 21
R7 R2 R5 R1 R4 R5 R8 R3
11:2 7:2 12:1 2:1 3:1 12:1 15:1 11:4

도면4

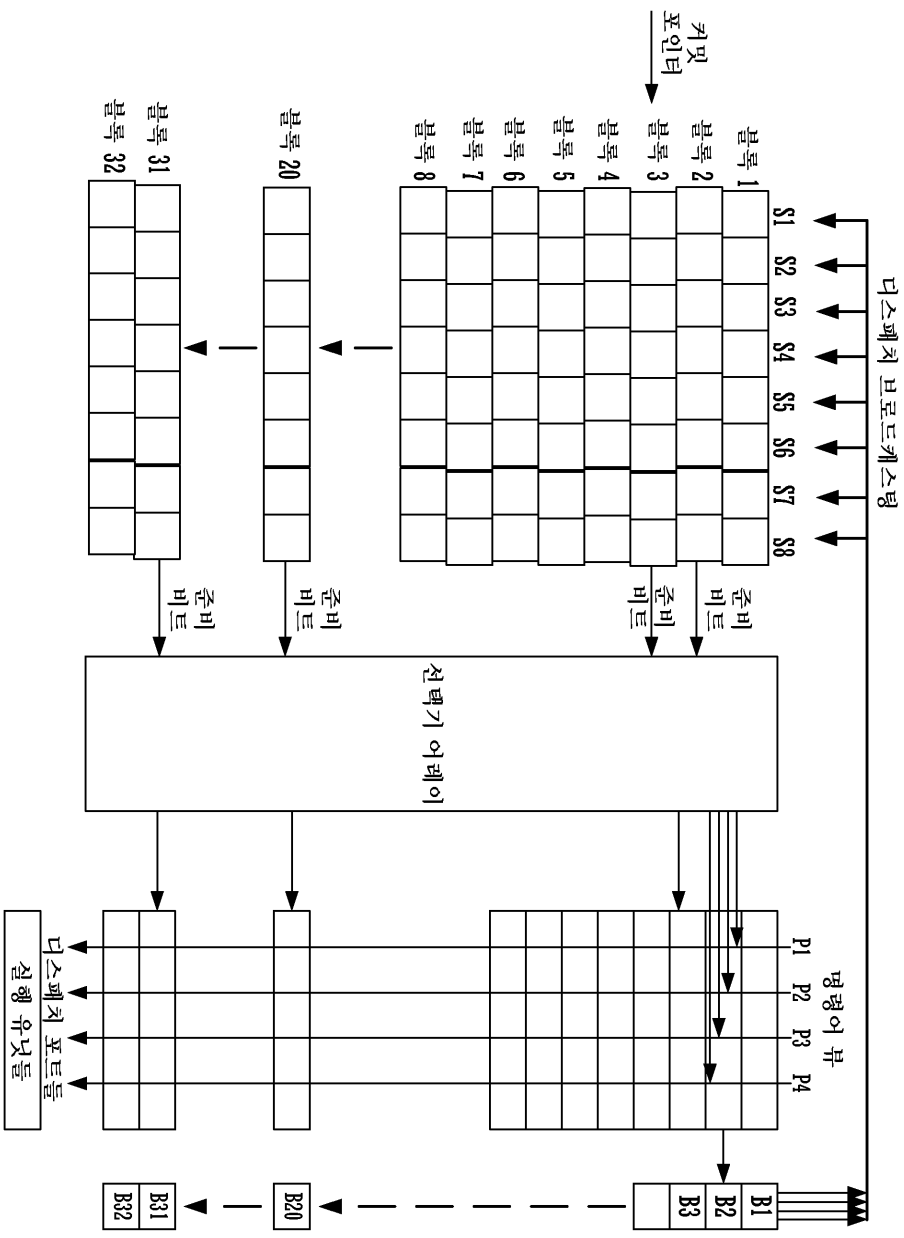

블록 1
블록 2
블록 3
블록 4
블록 5
블록 6
블록 7
블록 8
블록 20
블록 31
블록 32
B32
B28
B12
B11
B8
B7
B5
0
0
0
1
소스 뷰 내의 종속성 브로드캐스팅(제1 실시예)

도면5

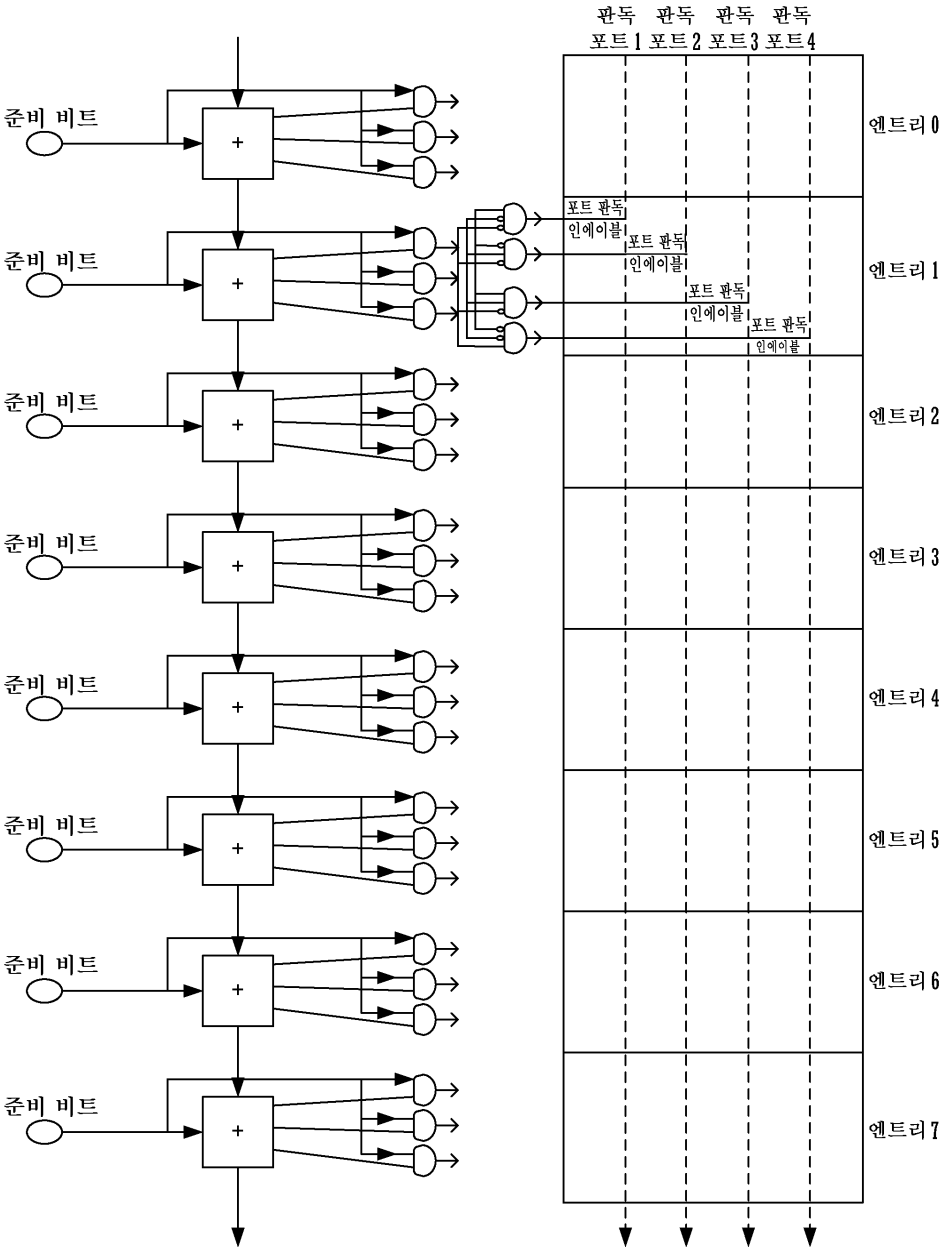
소스 뷰 내의 디스패치 브로드캐스팅(제2 실시예)



도면6

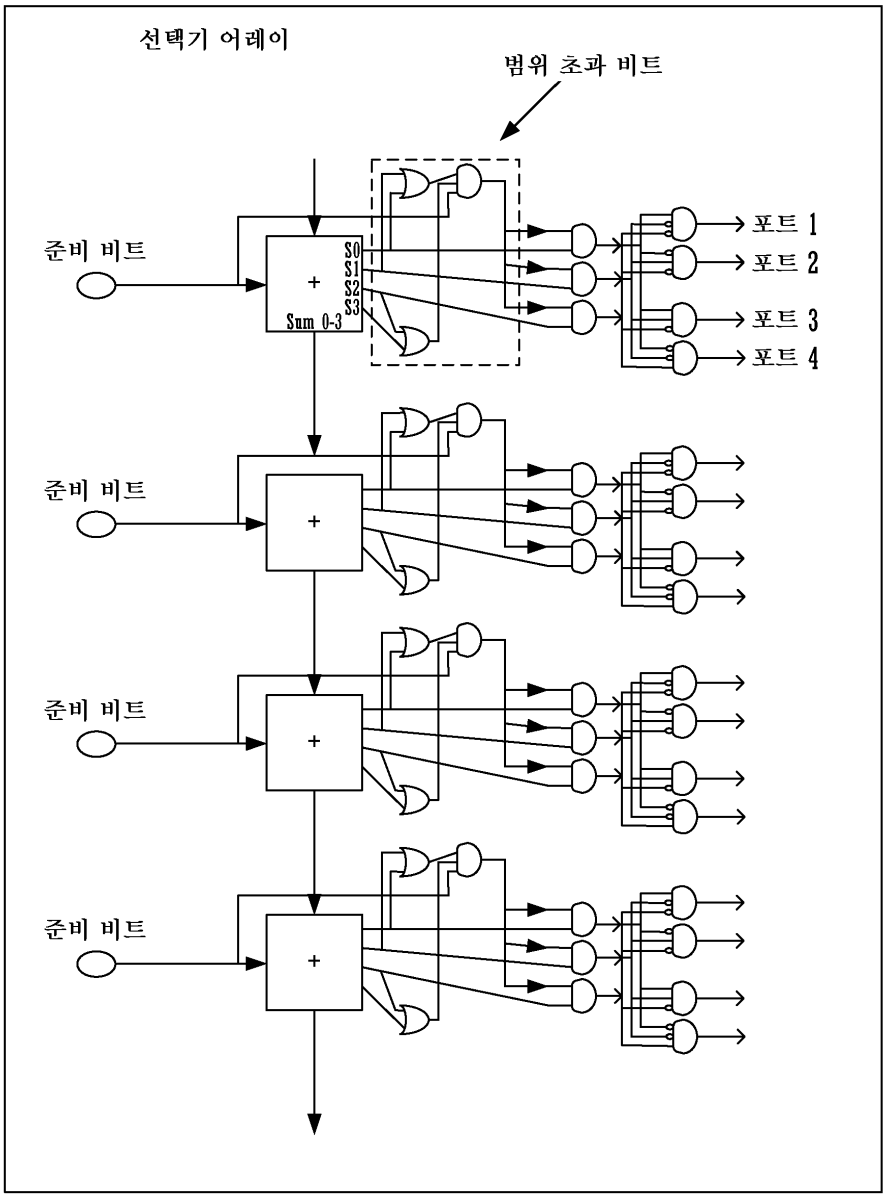

커밋 포인터
블록 1
블록 2
블록 3
블록 4
블록 5
블록 6
블록 7
블록 8
블록 20
블록 31
블록 32
S1
S2
S3
S4
S5
S6
S7
S8
디스패치 브로드캐스팅
준비 비트
준비 비트
준비 비트
준비 비트
선택기 어레이
P1
P2
P3
P4
명령어 뷰
B1
B2
B3
B20
B31
B32
디스패치 포트들
실행 유닛들

도면7


판독 포트 1
판독 포트 2
판독 포트 3
판독 포트 4
준비 비트
준비 비트
준비 비트
준비 비트
준비 비트
준비 비트
준비 비트
준비 비트
+
포트 판독 인에이블
포트 판독 인에이블
포트 판독 인에이블
포트 판독 인에이블
엔트리 0
엔트리 1
엔트리 2
엔트리 3
엔트리 4
엔트리 5
엔트리 6
엔트리 7

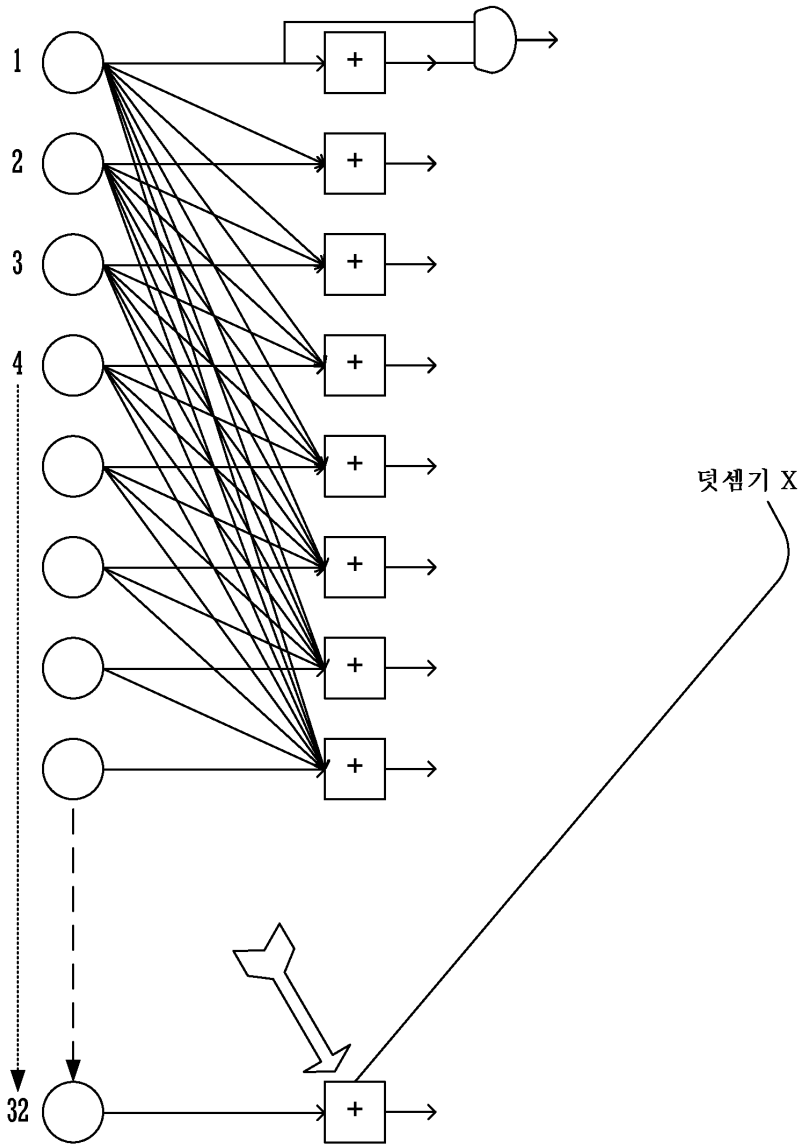
도면8

범위 초과 비트를 이용한 각각의 특정 포트의 직접 선택

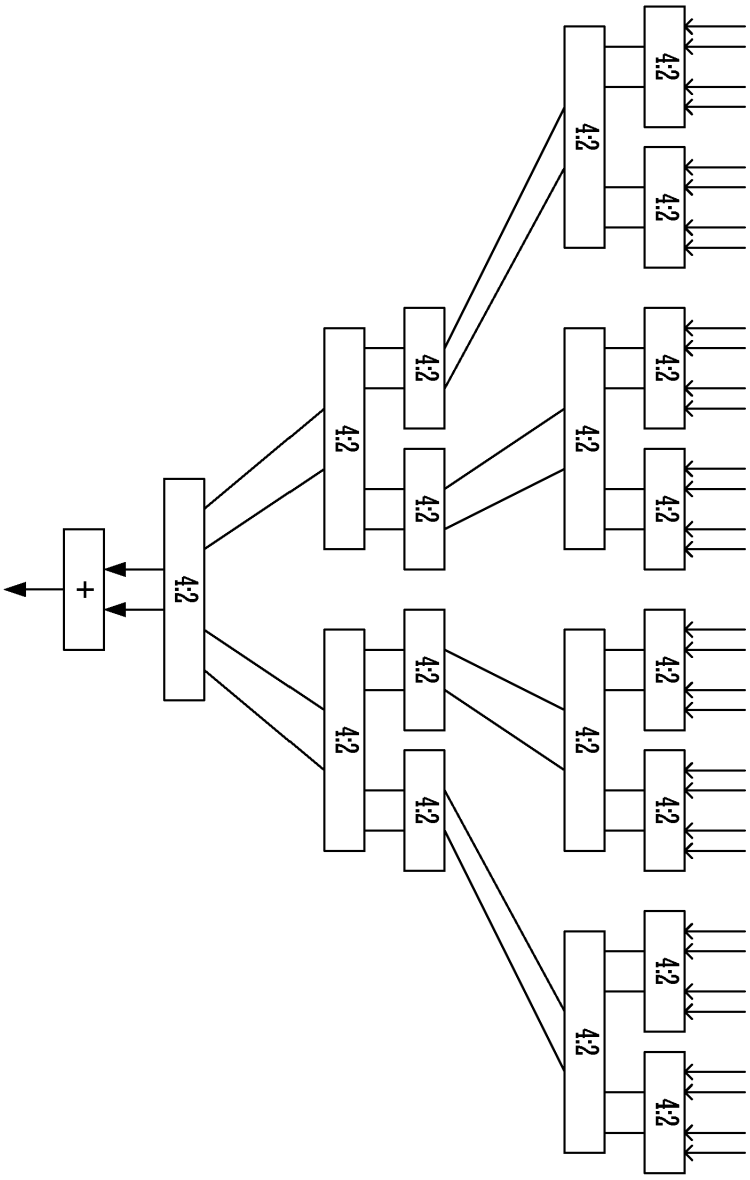


도면9

덧셈기 트리의 병렬 구현

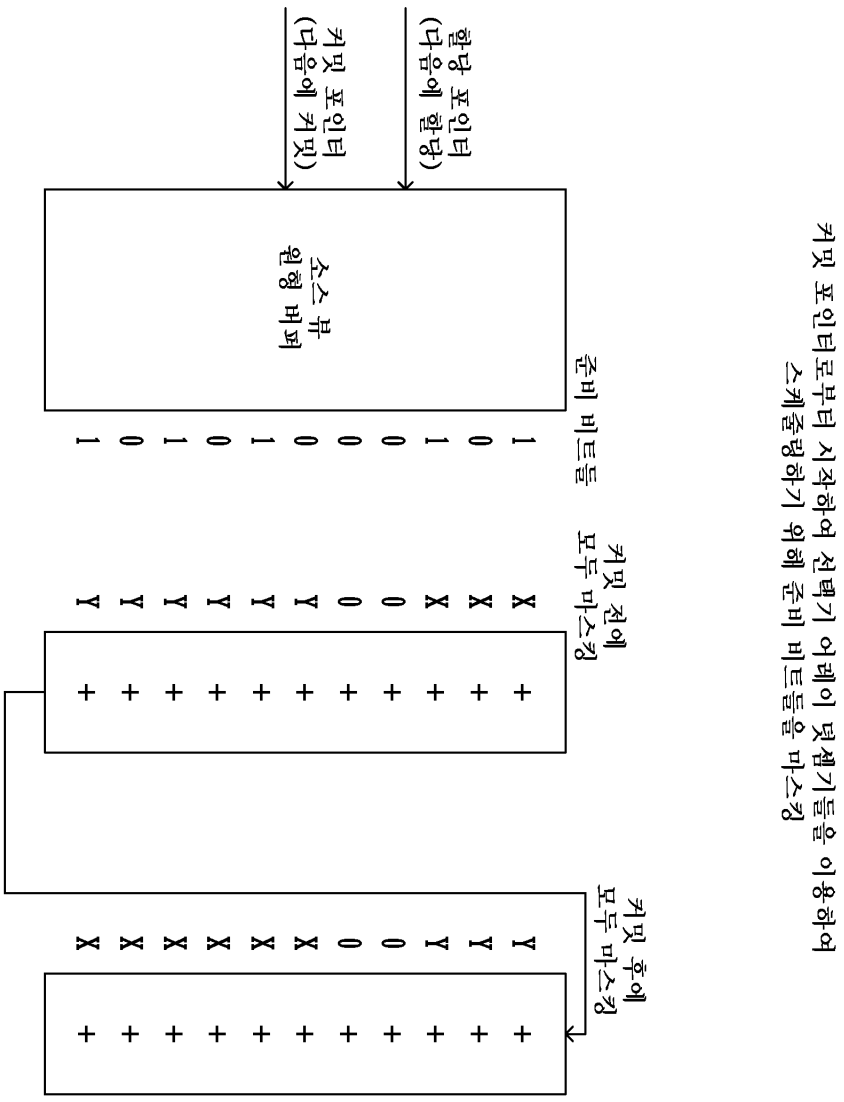


도면10



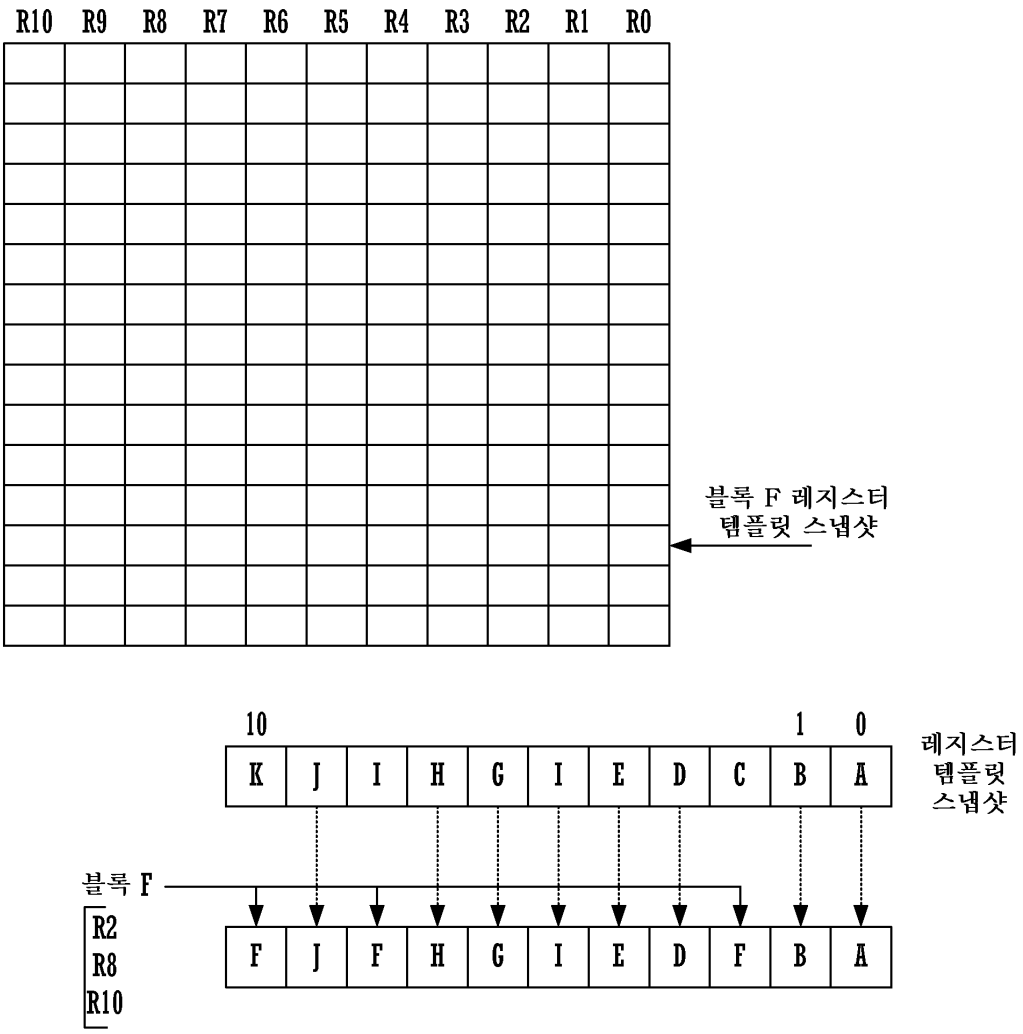
예시적인 덧셈기(x)의 캐리 세이브 덧셈기 병렬 구현

도면11

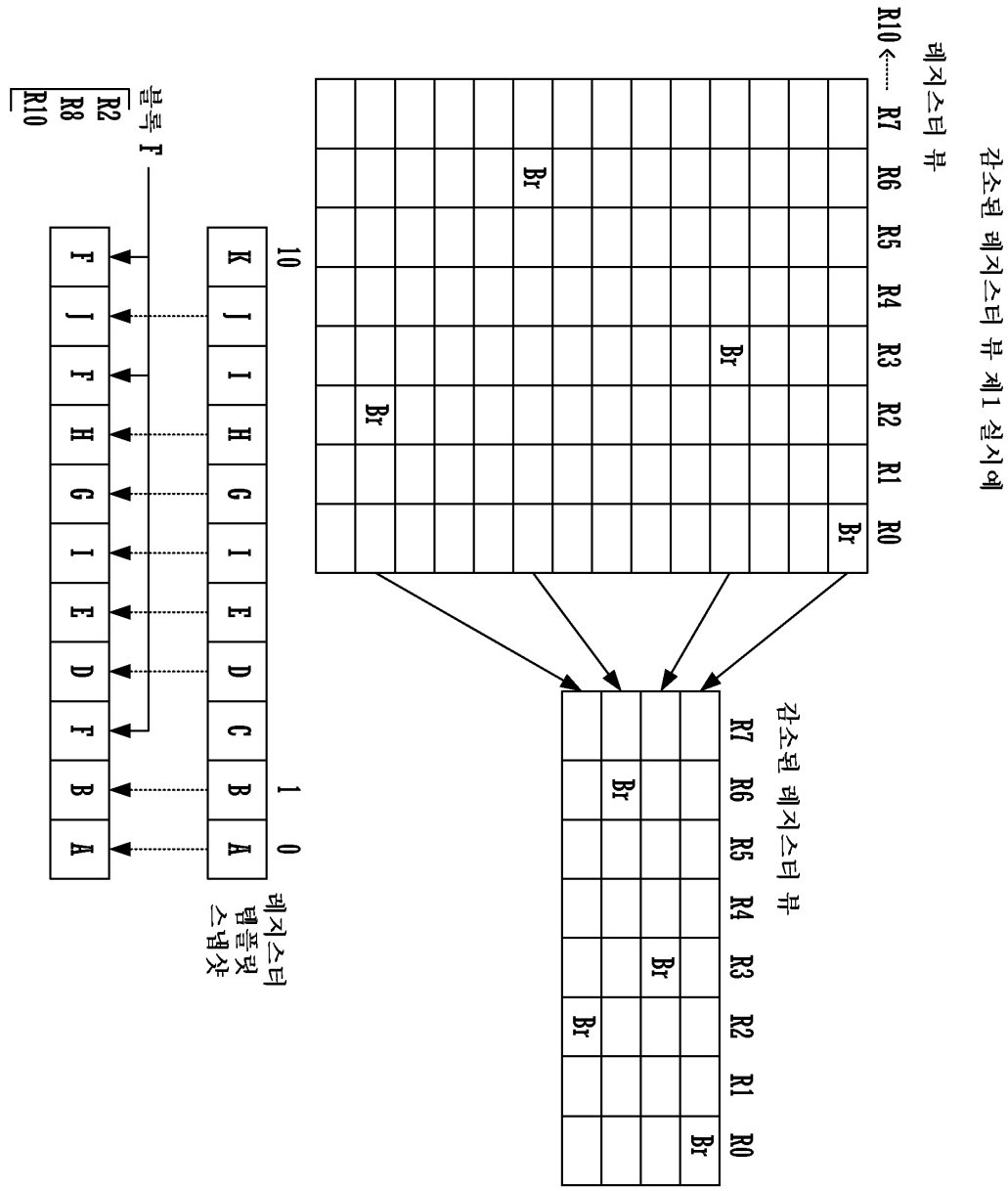

커밋 포인터로부터 시작하여 선택기 어레이 덧셀기들을 이용하여
스케줄링하기 위해 준비 비트들을 마스킹
할당 포인터
(다음에 할당)
커밋 포인터
(다음에 커밋)
소스 뷰
원형 버퍼
준비 비트들
커밋 전에
모두 마스킹
커밋 후에
모두 마스킹

도면12

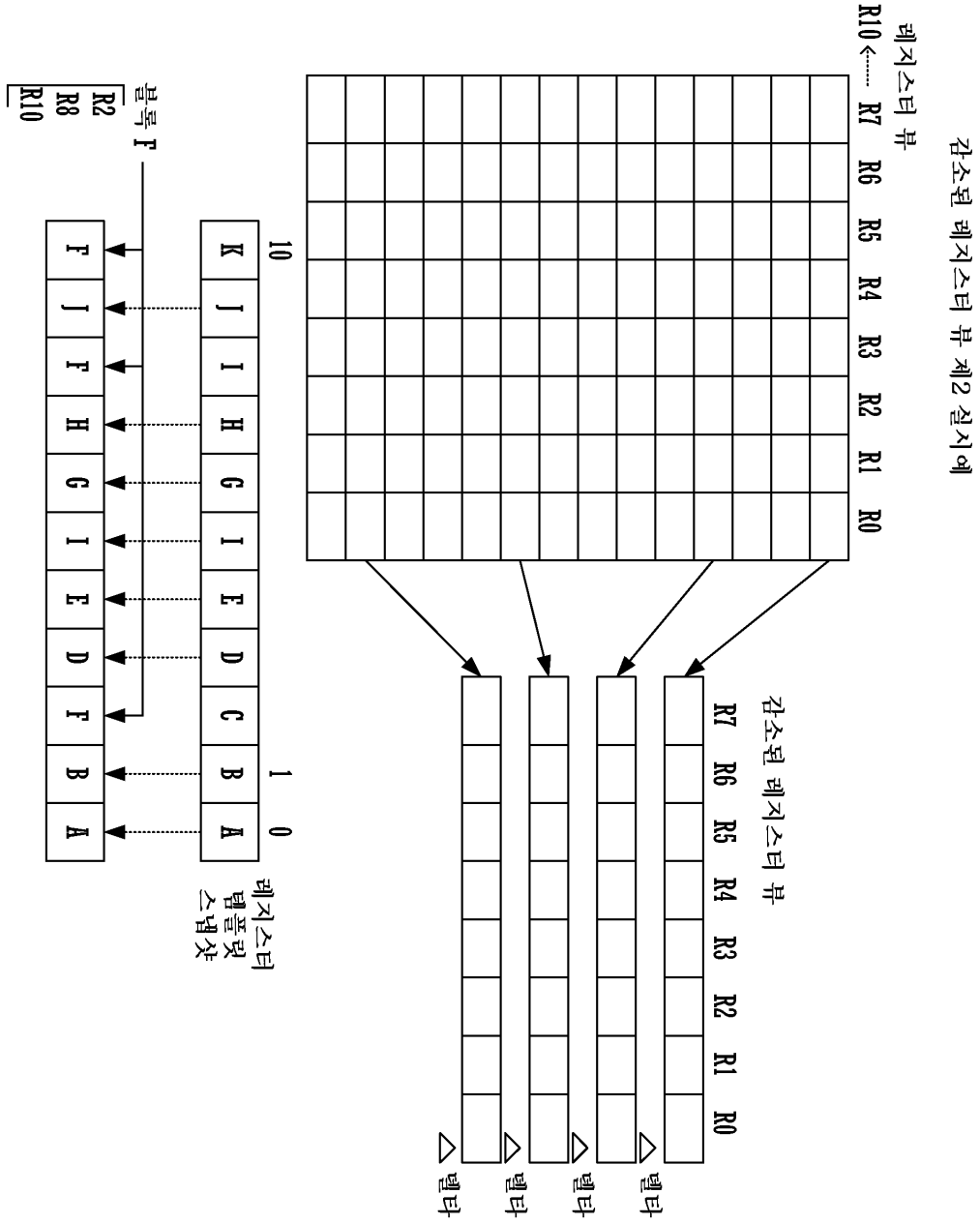
레지스터 템플릿들에 의해 채워진 레지스터 뷰 엔트리들



도면13


감소된 레지스터 뷰 제1 실시예
레지스터 뷰
R10 ..... R7 R6 R5 R4 R3 R2 R1 R0
Br
감소된 레지스터 뷰
R7 R6 R5 R4 R3 R2 R1 R0
Br
블록 F
R2
R8
R10
10 1 0
K J I H G I E D C B A
레지스터
탭플릿
스냅샷
F J F H G I E D F B A

도면14

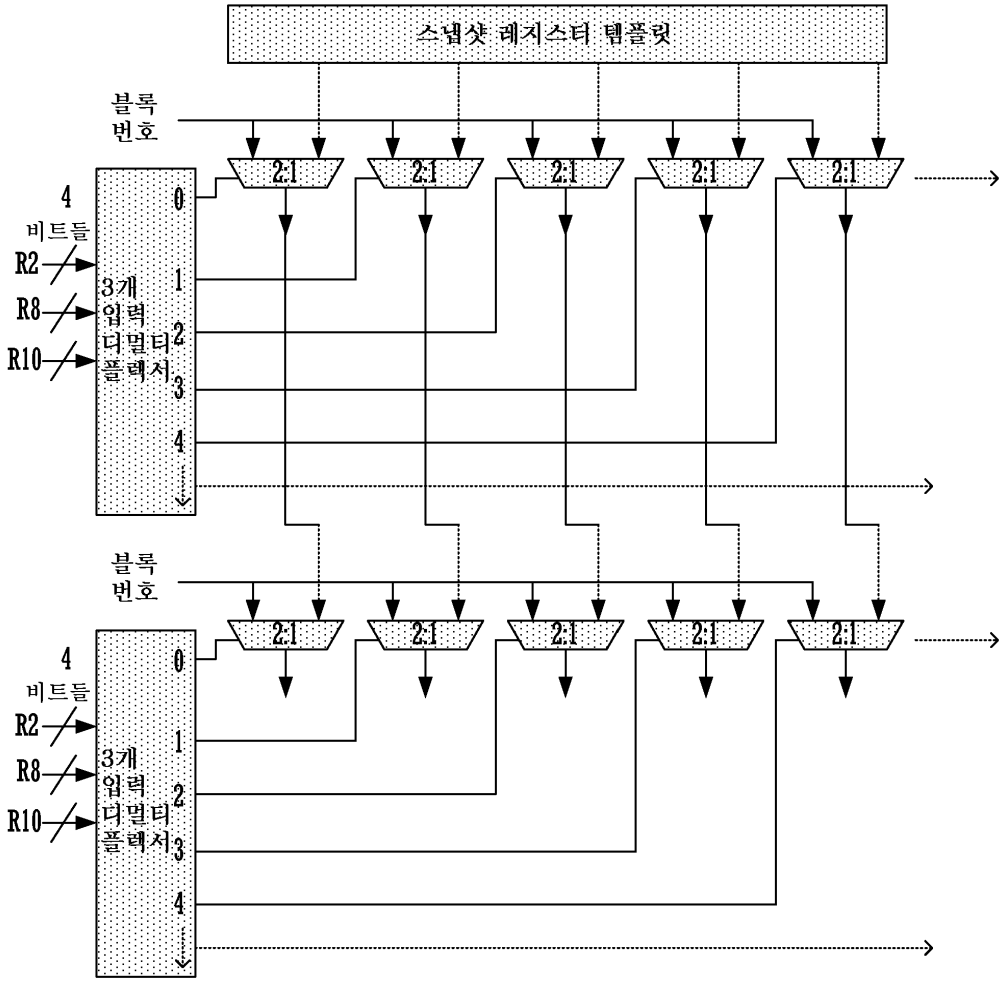

감소된 레지스터 뷰 제2 실시예
레지스터 뷰
R10
R7 R6 R5 R4 R3 R2 R1 R0
감소된 레지스터 뷰
R7 R6 R5 R4 R3 R2 R1 R0
멀티
멀티
멀티
멀티
블록 F
R2
R8
R10
K J I H G I E D C B A
10 1 0
레지스터
템플릿
스냅샷
F J F H G I E D F B A

도면15

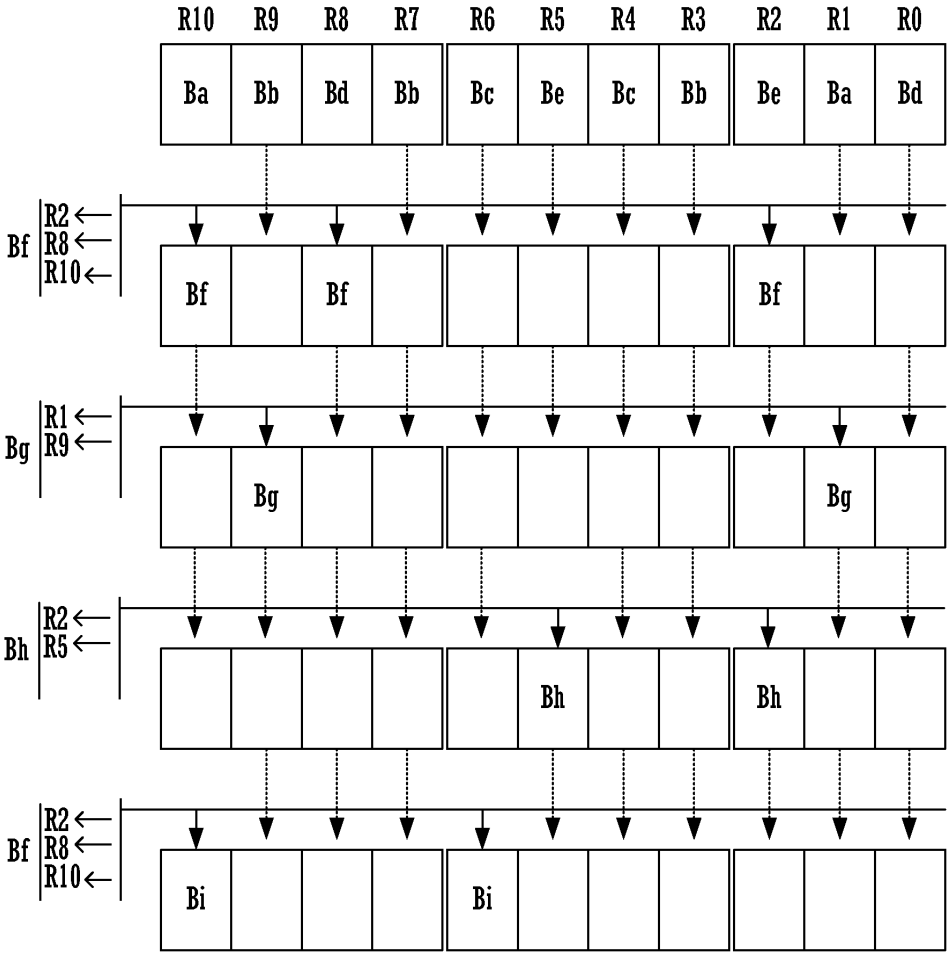
스냅샷들 사이의 델타의 포맷
원래 스냅샷
R63
R7 R6 R5 R4 R3 R2 R1 R0
B1
R1과 R7은 B2에 의해
업데이트되고 있는
유일한 레지스터들이다.
엔트리들의 나머지는
변경되지 않는다.
R7 R1 : B2
R5 R6 : B3
R5와 R6는 B3에 의해
업데이트되고 있는
유일한 레지스터들이다.
엔트리들의 나머지는
변경되지 않는다.

도면16

명령어 블록들의 할당에 따라 레지스터 템플릿 스냅샷들을 생성

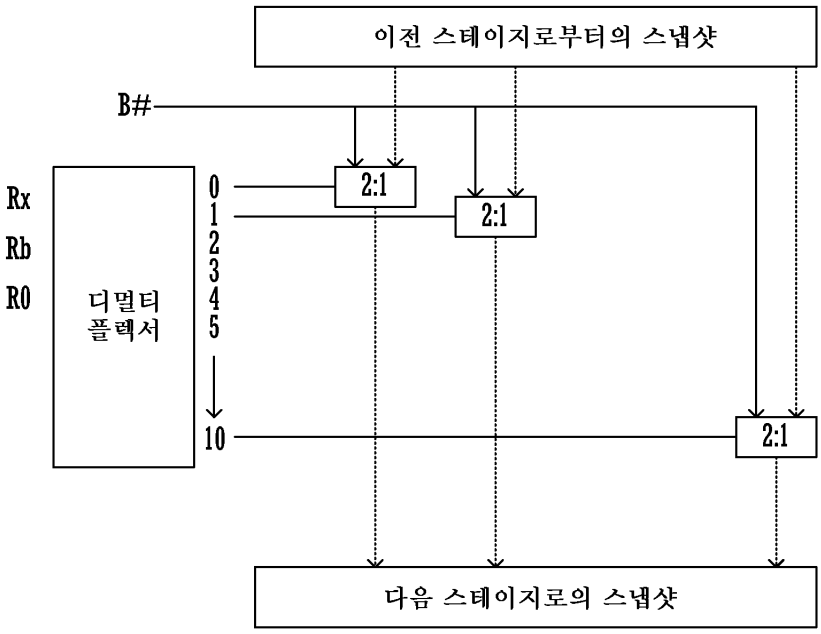


도면17

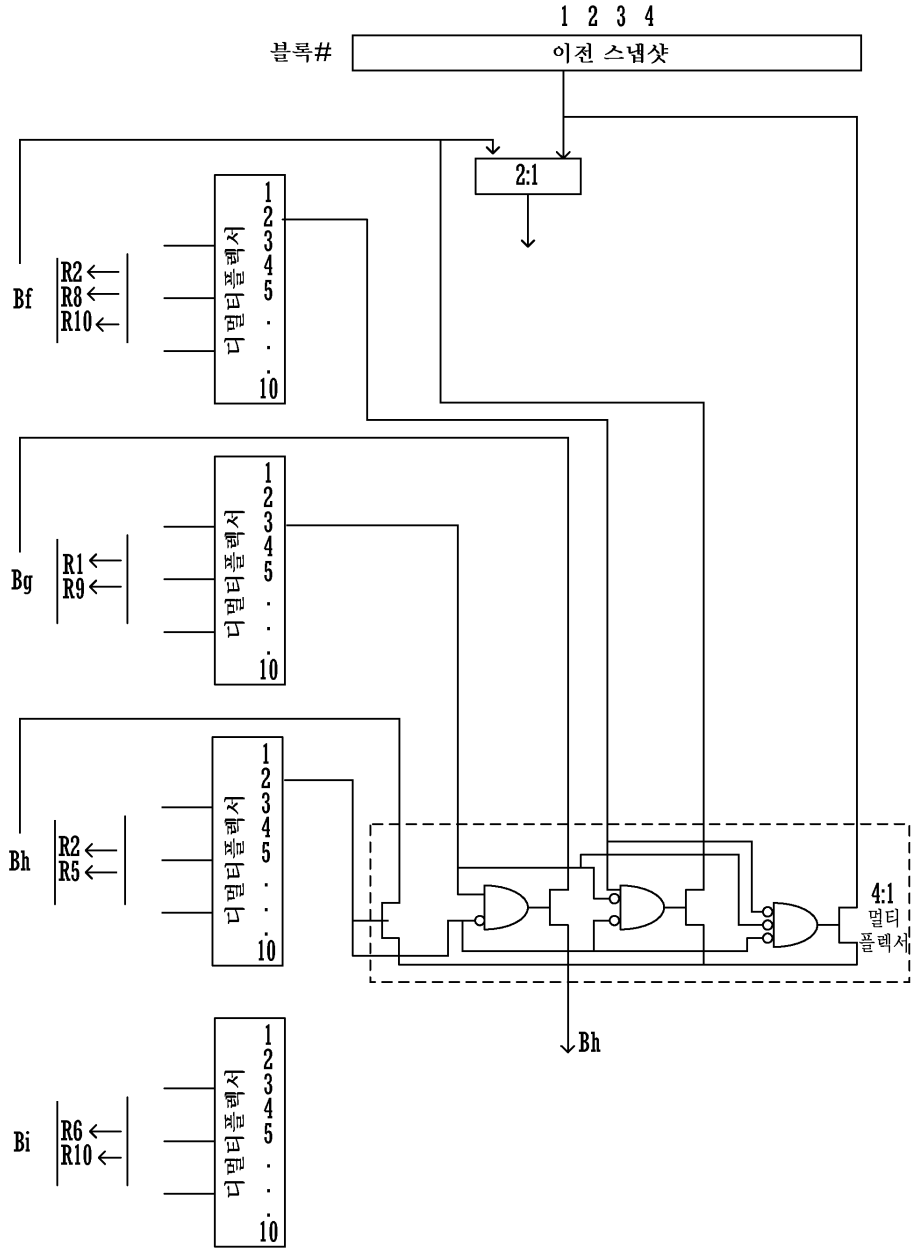

R10
R9
R8
R7
R6
R5
R4
R3
R2
R1
R0
Ba
Bb
Bd
Bb
Bc
Be
Bc
Bb
Be
Ba
Bd
Bf
R2
R8
R10
Bf
Bf
Bf
Bg
R1
R9
Bg
Bg
Bh
R2
R5
Bh
Bh
Bf
R2
R8
R10
Bi
Bi

도면18

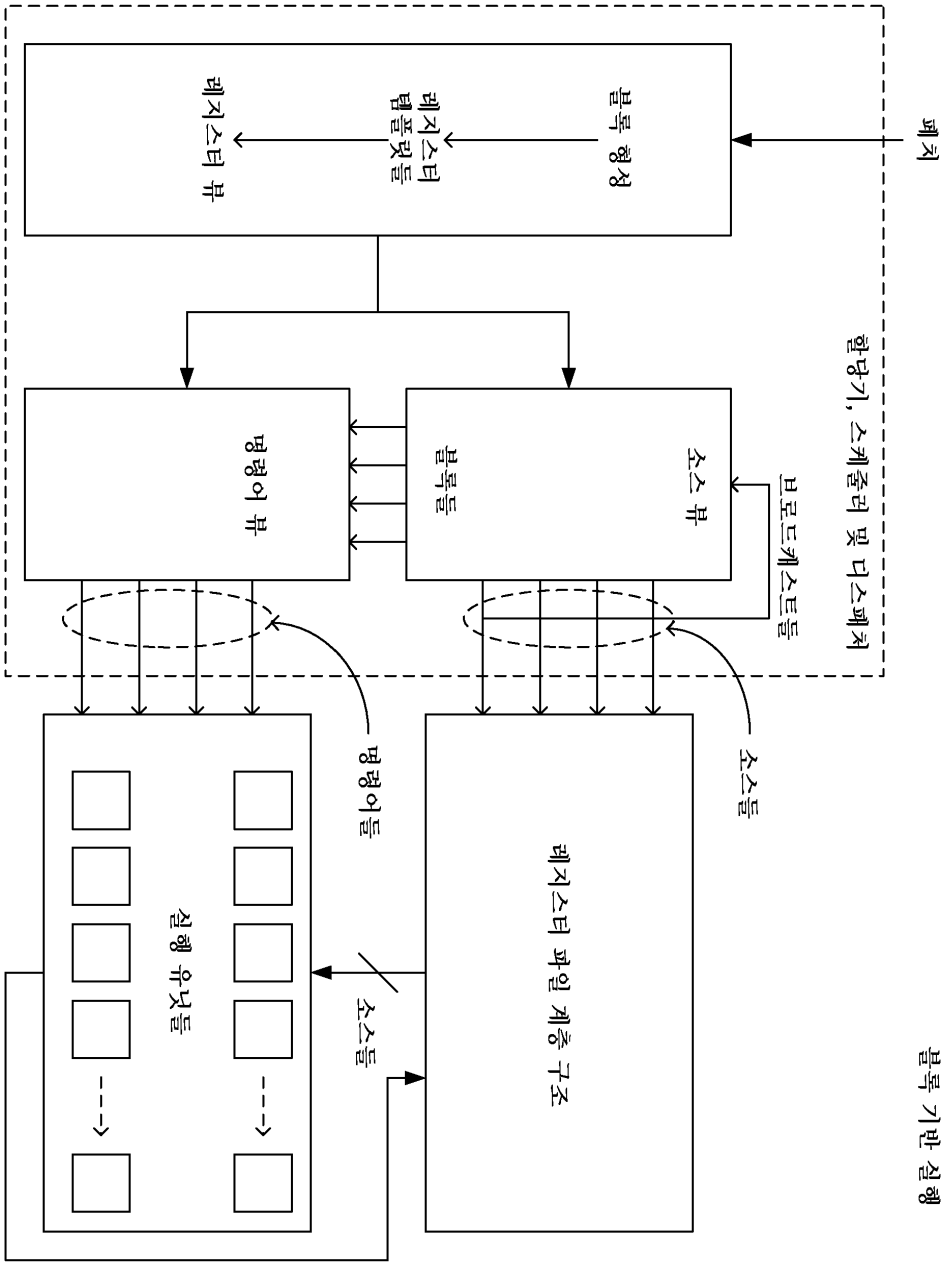
직렬 구현


이전 스테이지로부터의 스냅샷
B#
Rx
Rb
R0
디멀티
플렉서
0
1
2
3
4
5
10
2:1
2:1
2:1
다음 스테이지로의 스냅샷

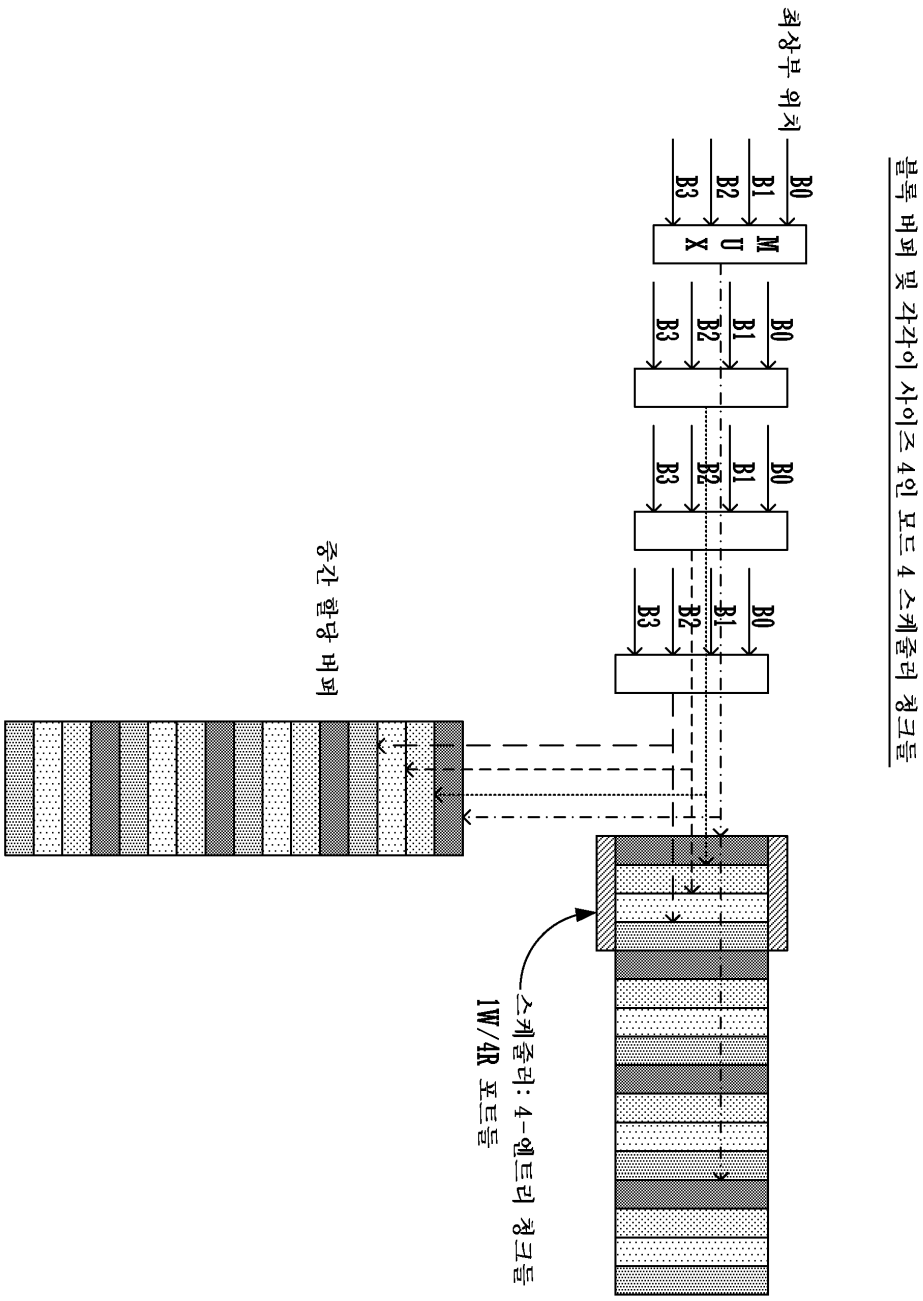
도면19


1 2 3 4
블록#
이전 스냅샷
2:1
Bf
R2
R8
R10
디멀티플렉서
1
2
3
4
5
10
Bg
R1
R9
Bh
R2
R5
4:1
멀티
플렉서
Bh
Bi
R6
R10

도면20

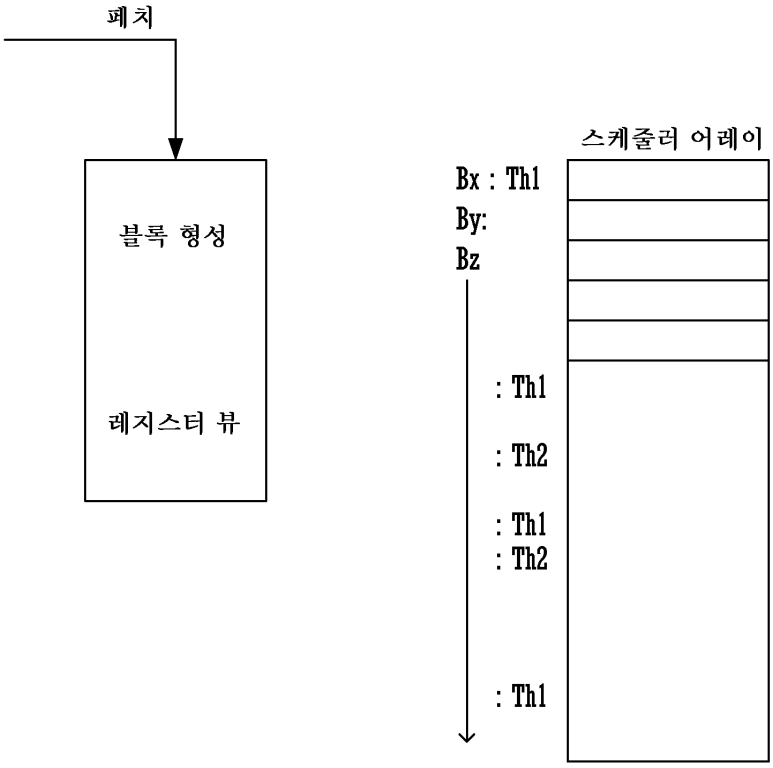

블록 기반 실행
페치
할당기, 스케줄러 및 디스패처
블록 형성
레지스터 템플릿들
레지스터 뷰
명령어 뷰
소스 뷰
블록들
브로드캐스트들
명령어들
소스들
레지스터 파일 계층 구조
소스들
실행 유닛들

도면21

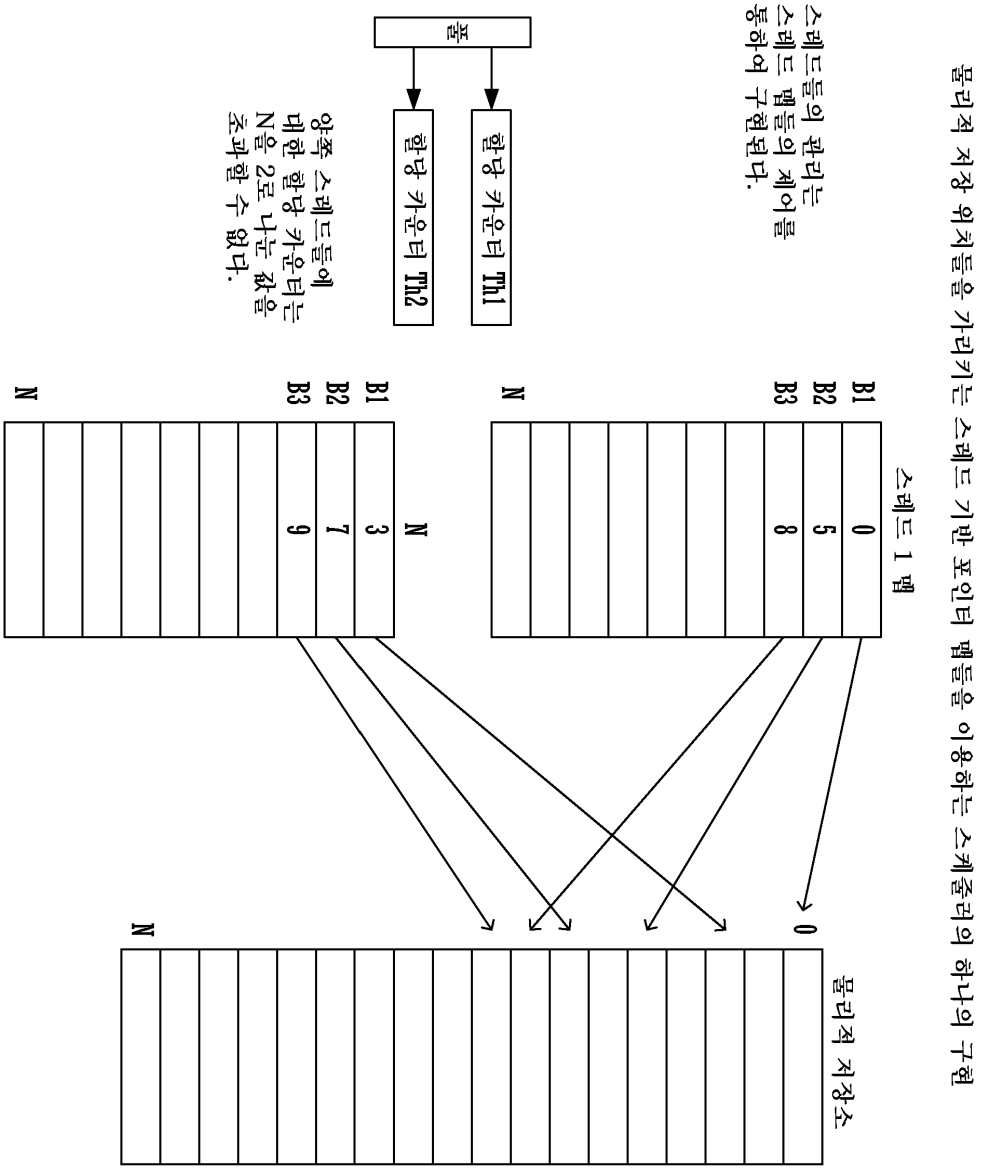

블록 버퍼 및 각각이 사이즈 4인 포드 4 스케줄러 청크들
최상부 위치
B0
B1
B2
B3
MUX
중간 할당 버퍼
스케줄러: 4-엔트리 청크들
1W/4R 포트들

도면22

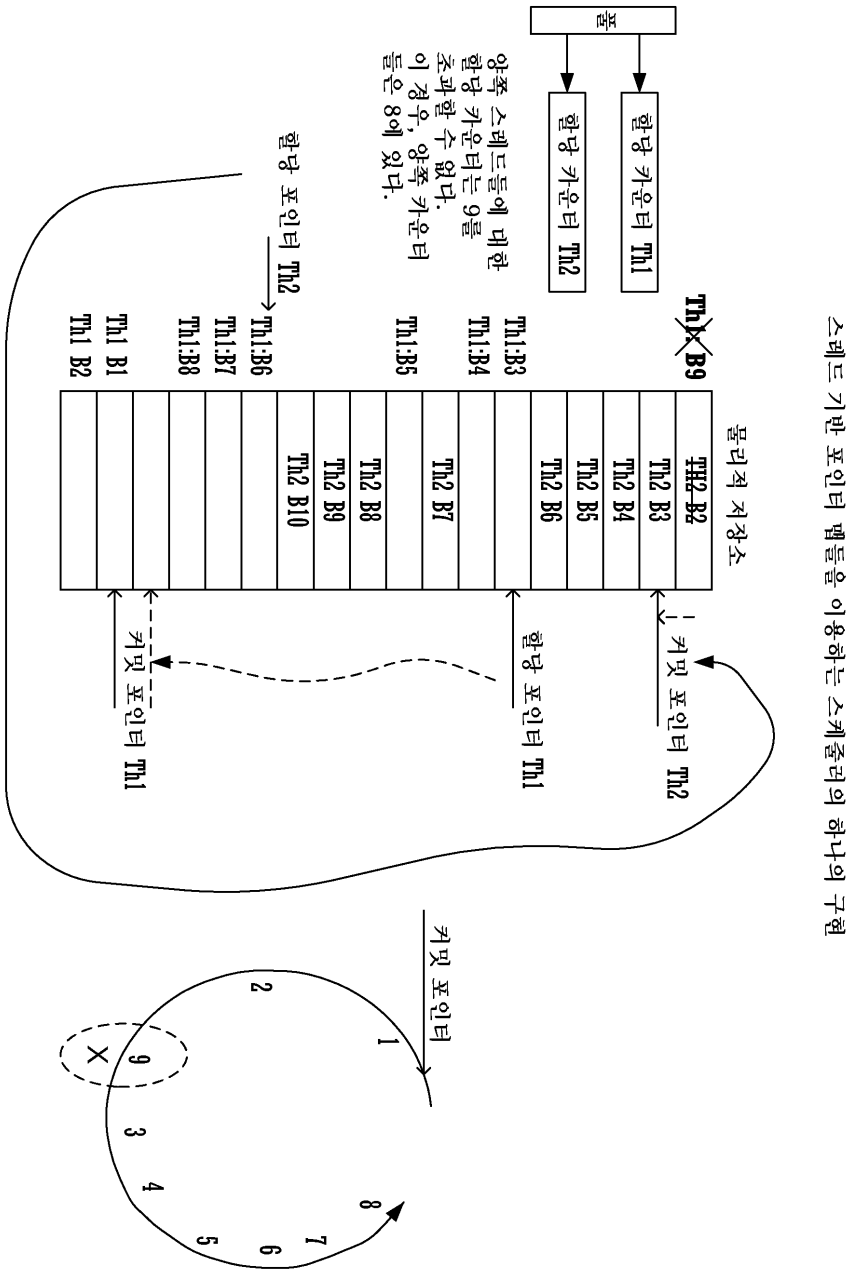
스레드들이 그들의 블록 번호들 및 스레드 ID에 따라
할당되는 방법에 대한 묘사



도면23

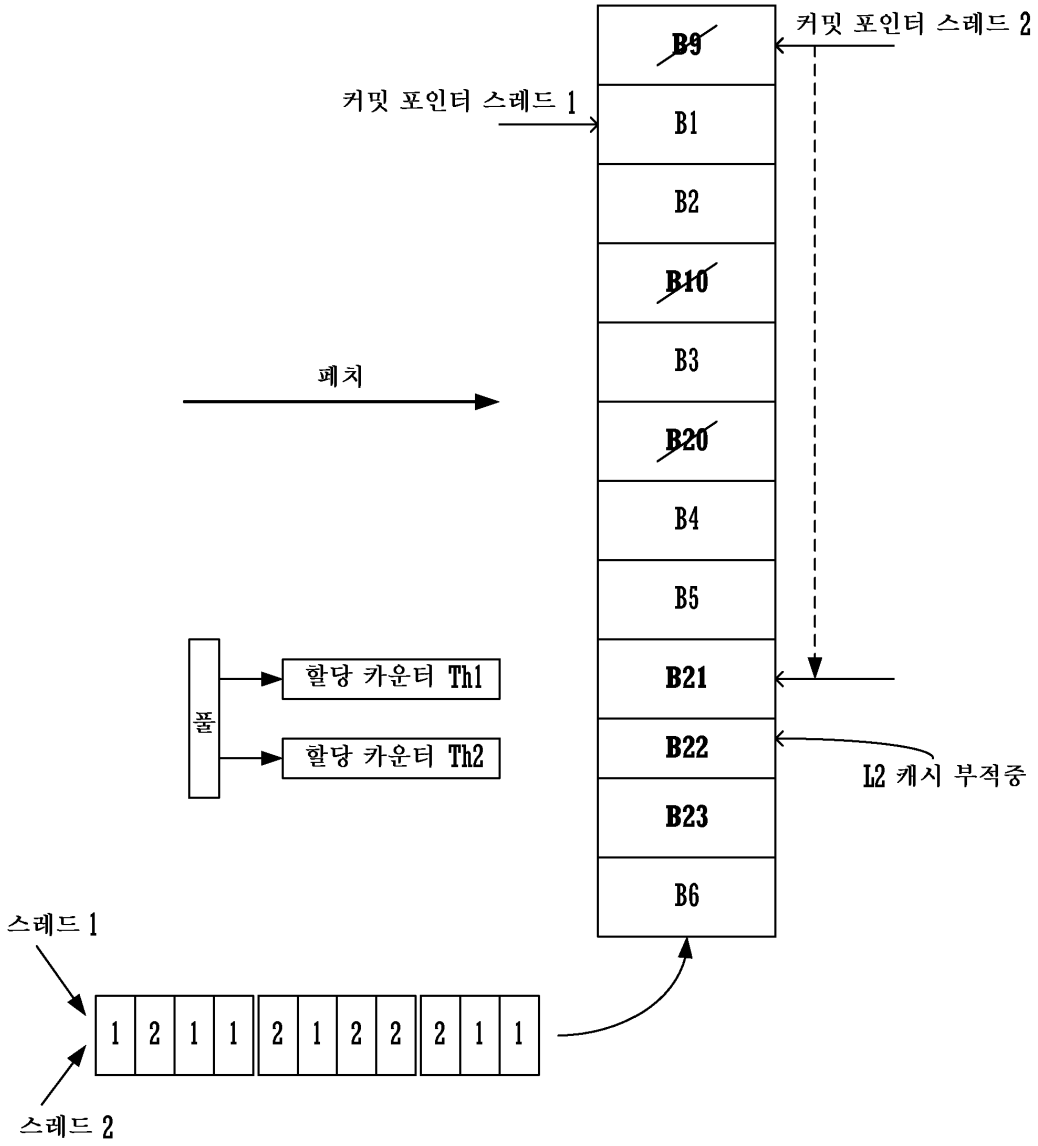

물리적 저장 위치들을 가리키는 스레드 기반 포인터 맵들을 이용하는 스케줄러의 하나의 구현
스레드들의 관리는
스레드 맵들의 제어를
통하여 구현된다.
풀
할당 카운터 Th1
할당 카운터 Th2
양쪽 스레드들에
대한 할당 카운터는
N을 2로 나눈 값을
초과할 수 없다.
스레드 1 맵
B1
B2
B3
N
0
5
8
3
7
9
물리적 저장소

도면24

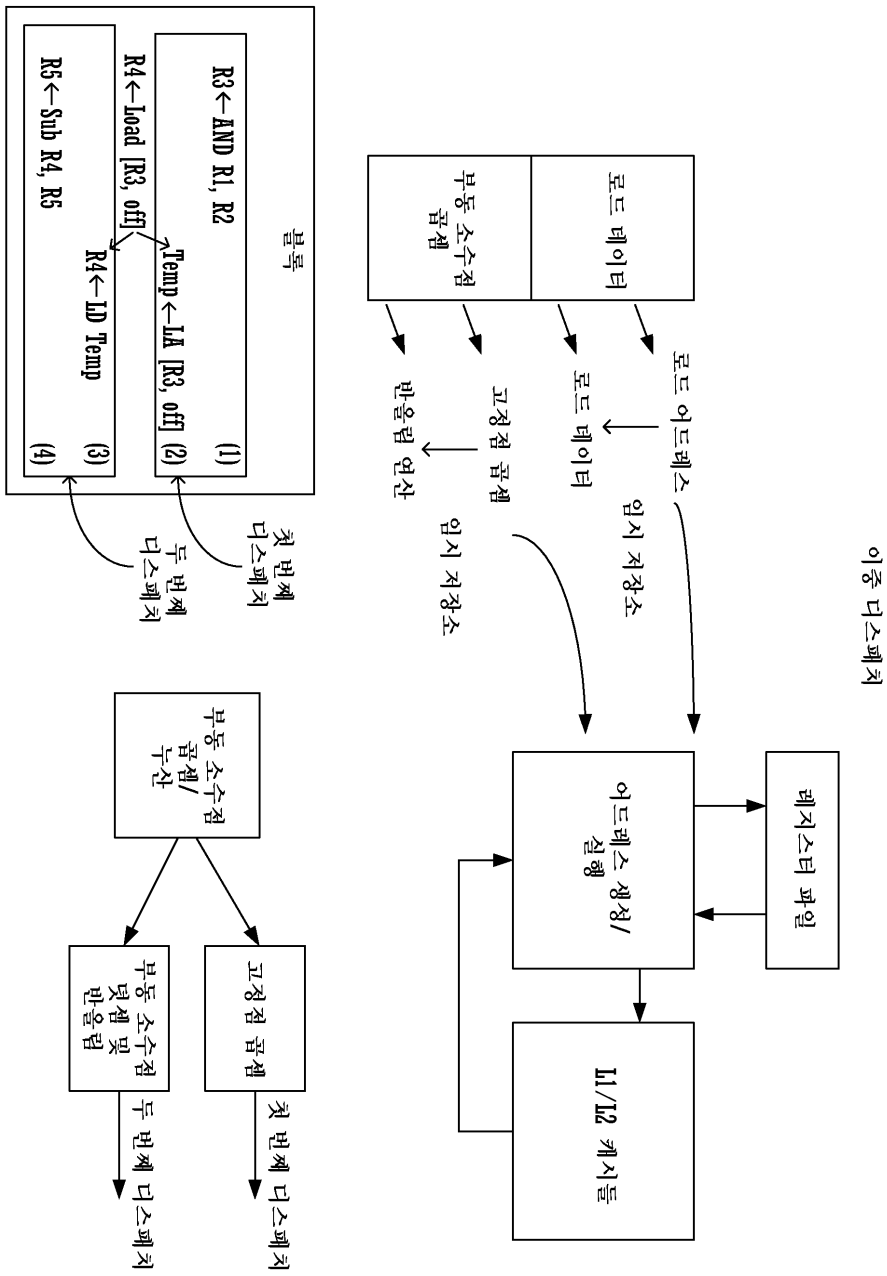

스레드 기반 포인터 맵들을 이용하는 스케줄러의 하나의 구현
풀
할당 카운터 Th1
할당 카운터 Th2
양쪽 스레드들에 대한
할당 카운터는 9를
조과할 수 없다.
이 경우, 양쪽 카운터
들은 8에 있다.
물리적 저장소
Th1 B1
Th1 B2
Th1:B3
Th1:B4
Th1:B5
Th1:B6
Th1:B7
Th1:B8
Th2 B2
Th2 B3
Th2 B4
Th2 B5
Th2 B6
Th2 B7
Th2 B8
Th2 B9
Th2 B10
커밋 포인터 Th2
할당 포인터 Th1
커밋 포인터 Th1
할당 포인터 Th2
커밋 포인터
1
2
3
4
5
6
7
8
9
X

도면25

스레드들에 대한 실행 리소스들의 동적인 카운터 기반 공정 할당

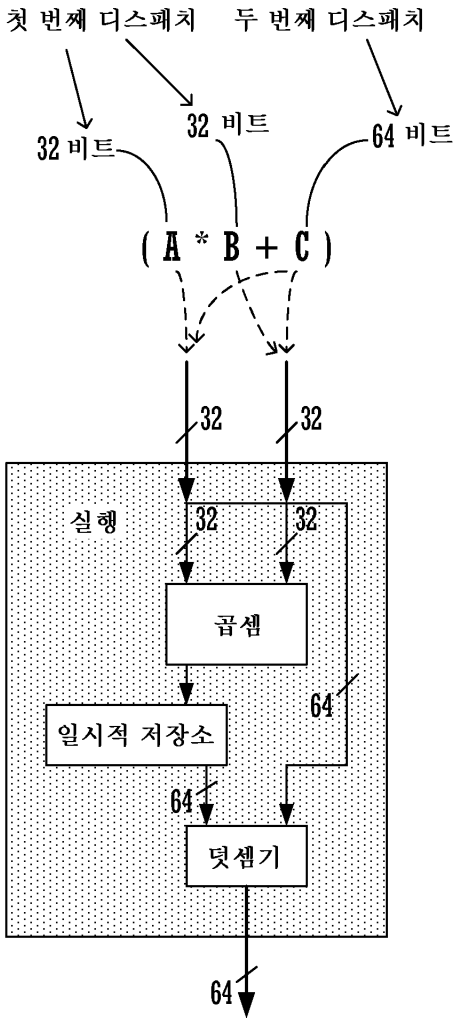


도면26


이중 디스패치
레지스터 파일
어드레스 생성/
실행
L1/L2 캐시들
로드 데이터
부동 소수점
곱셈
로드 어드레스
로드 데이터
임시 저장소
고정점 곱셈
반올림 연산
임시 저장소
블록
R3←AND R1, R2 (1)
R4←Load [R3, off] (2)
Temp←LA [R3, off]
R4←LD Temp (3)
R5←Sub R4, R5 (4)
첫 번째
디스패치
두 번째
디스패치
부동 소수점
곱셈/
누산
고정점 곱셈
부동 소수점
덧셈 및
반올림
첫 번째 디스패치
두 번째 디스패치

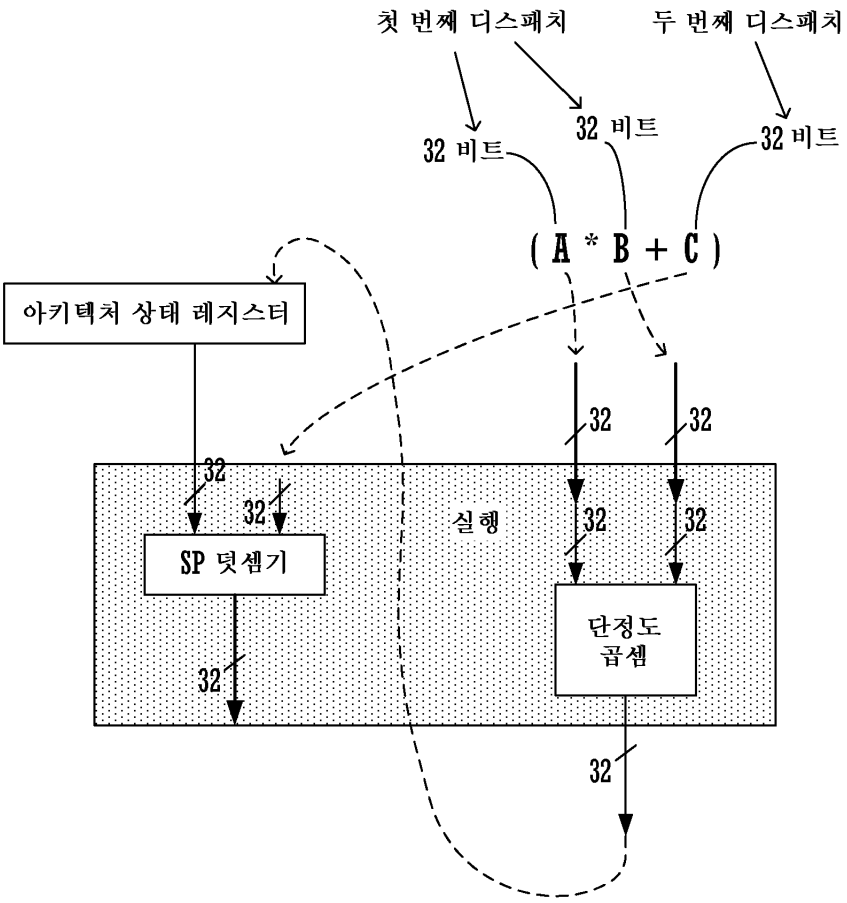
도면27

이중 디스패치 일시적 곱셈-누산

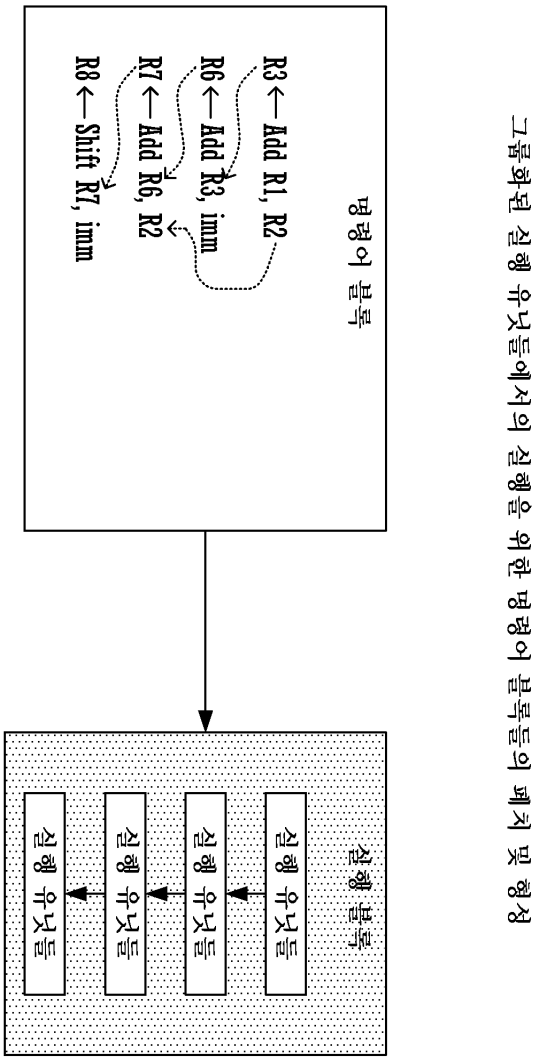


도면28

이중 디스패치 아키텍처적으로 보이는 상태 곱셈-덧셈



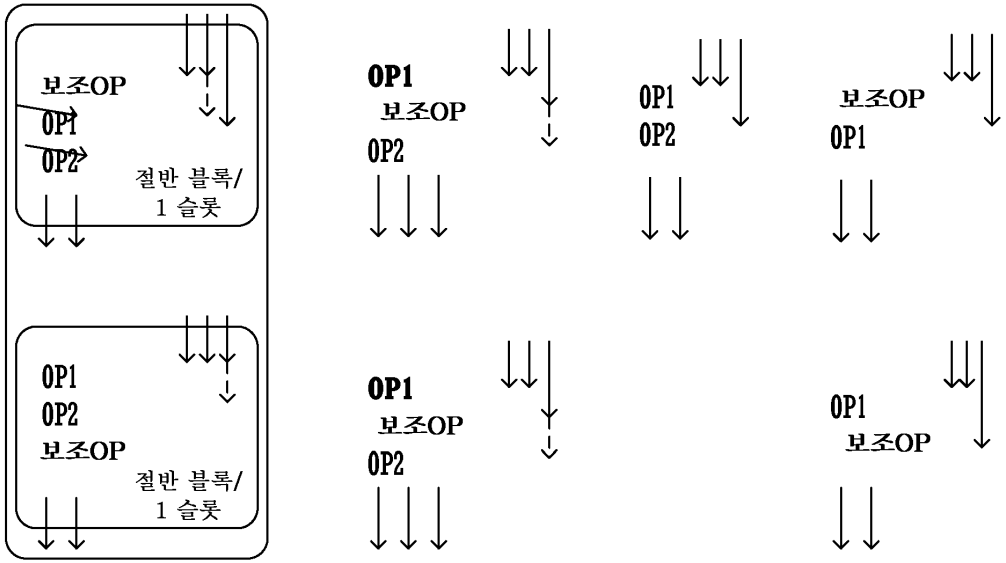
도면29



그룹화된 실행 유닛들에서의 실행을 위한 명령어 블록들의 페치 및 형성

명령어 그룹화

보조 OP: 논리, op 시프트; 이동, 부호 확장, 분기*, 등.



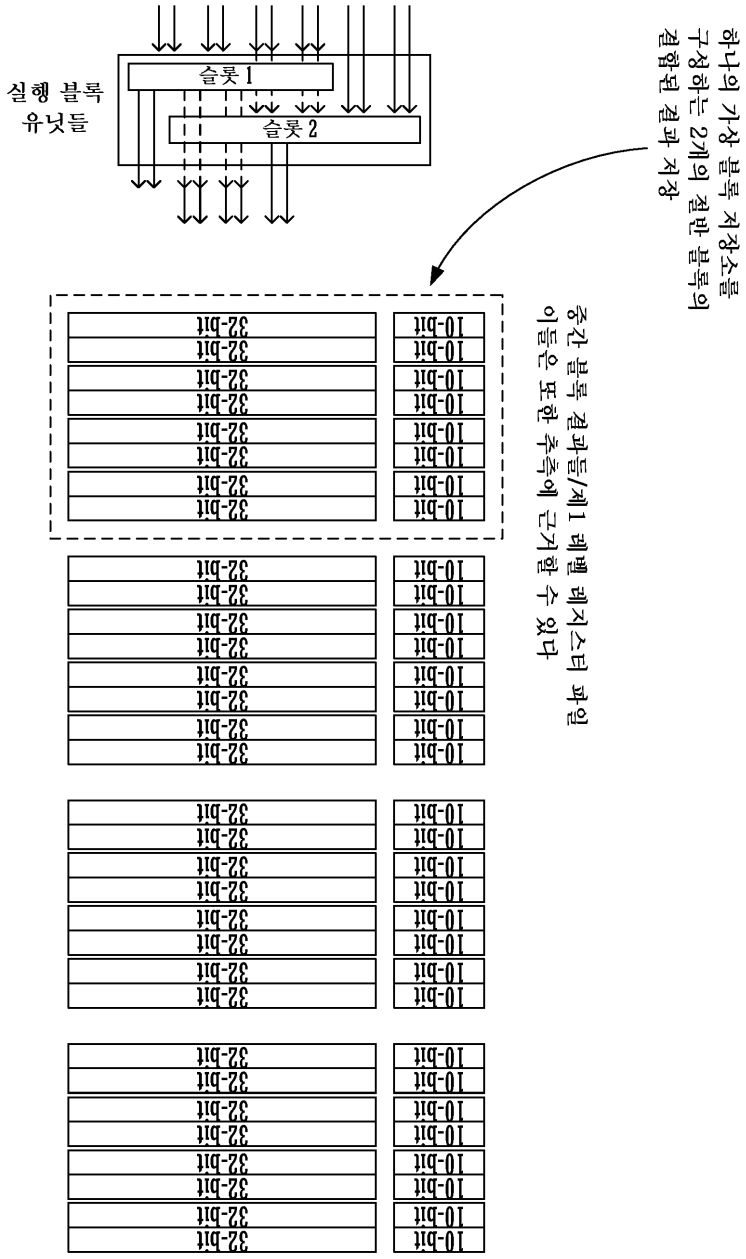
3개의 소스: 제4 소스는, 즉시, 공유, 등
2개의 목적지: 제3 목적지(소스/목적지 인코딩)

도면30

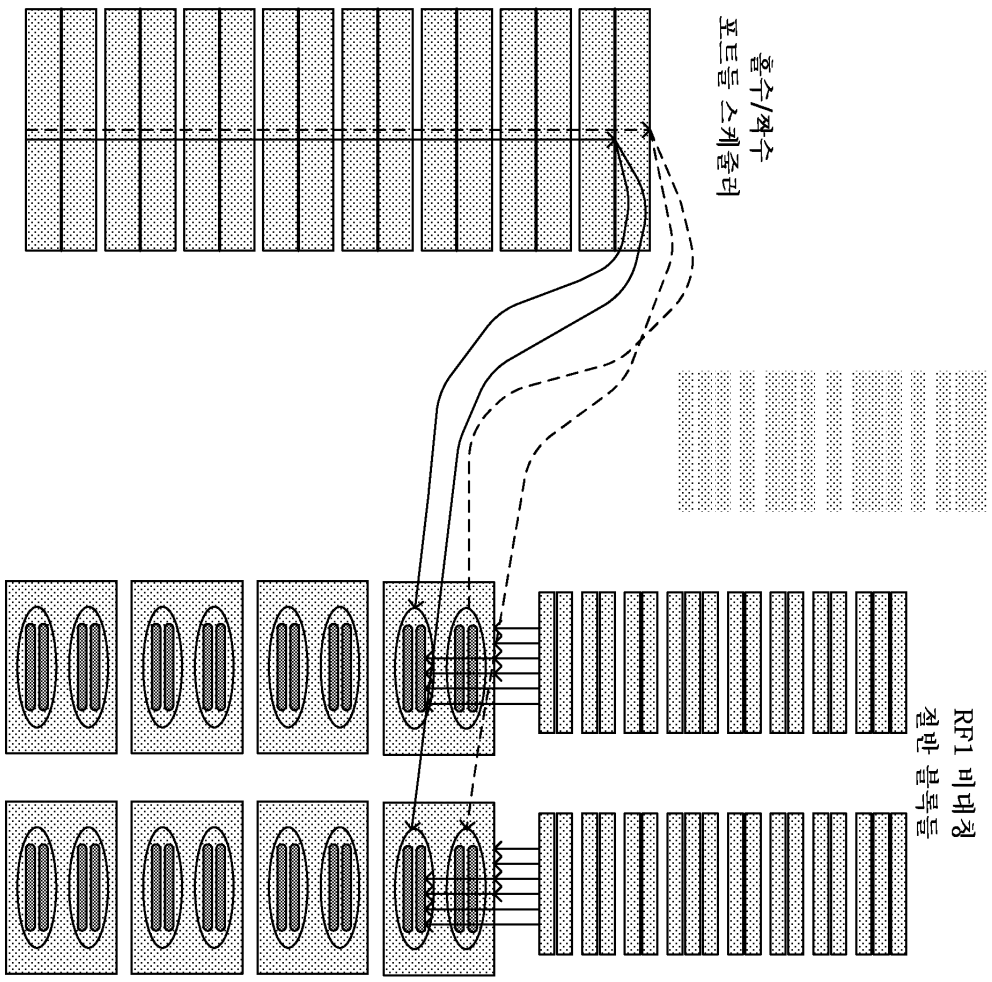
도면31

블록 또는 1/2 블록 컨테스트에서의 쌍들
슬롯들: 2개 슬롯/절반 블록들, 각각은 단일/쌍/3중쌍
블록 실행 유형들: 4
1: 병렬 절반들(독립적으로 실행)
2: 원자 절반들(병렬)(리소스 제약)
3: 원자 절반들(직렬)(일시적 전송/저장 없음)
4: 순차적 절반들(DD)
실행 블록
유닛들
슬롯 1
슬롯 2
OP1
보조OP
OP2
절반 블록/
1 슬롯
절반 블록/
1 슬롯

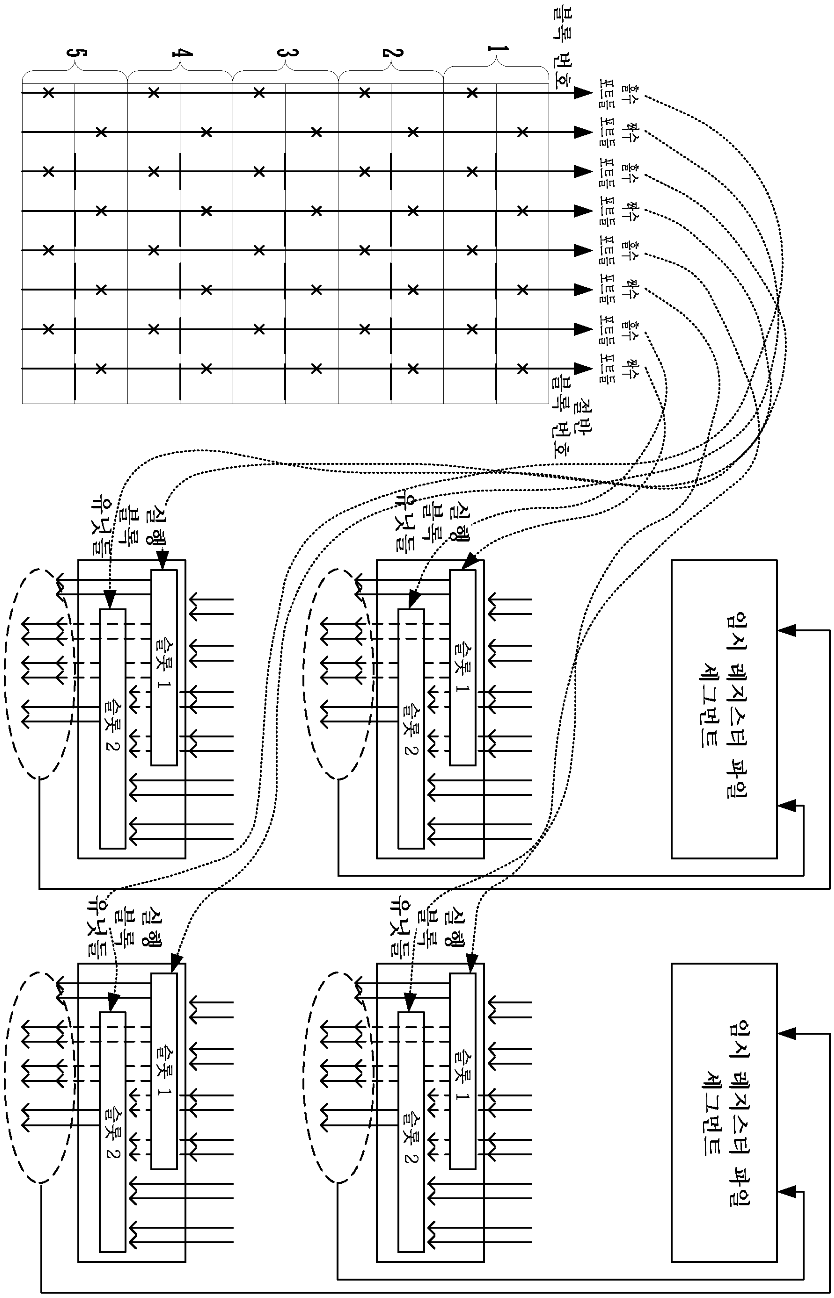
도면32


실행 블록
유닛들
슬롯 1
슬롯 2
하나의 가상 블록 저장소를
구성하는 2개의 절반 블록의
결합된 결과 저장
중간 블록 결과들/제 1 레벨 레지스터 파일
이들은 또한 추측에 근거할 수 있다
32-bit
10-bit

도면33

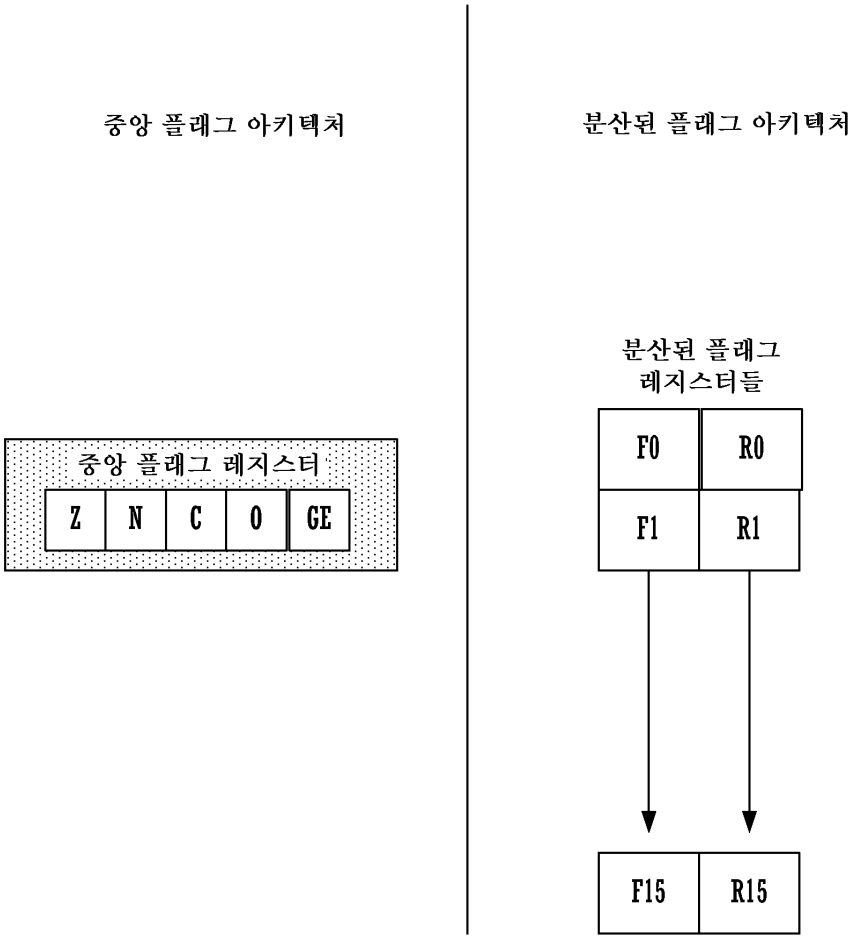


도면34

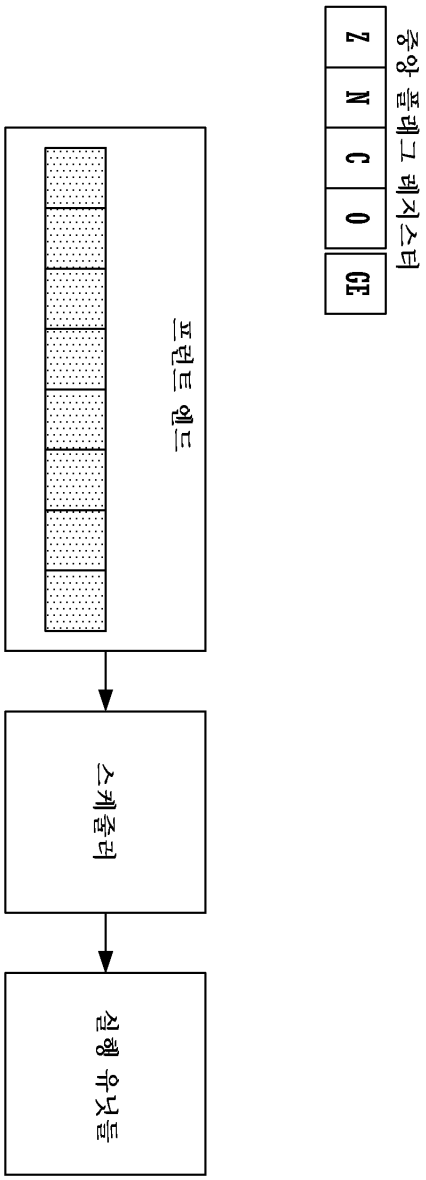


도면35

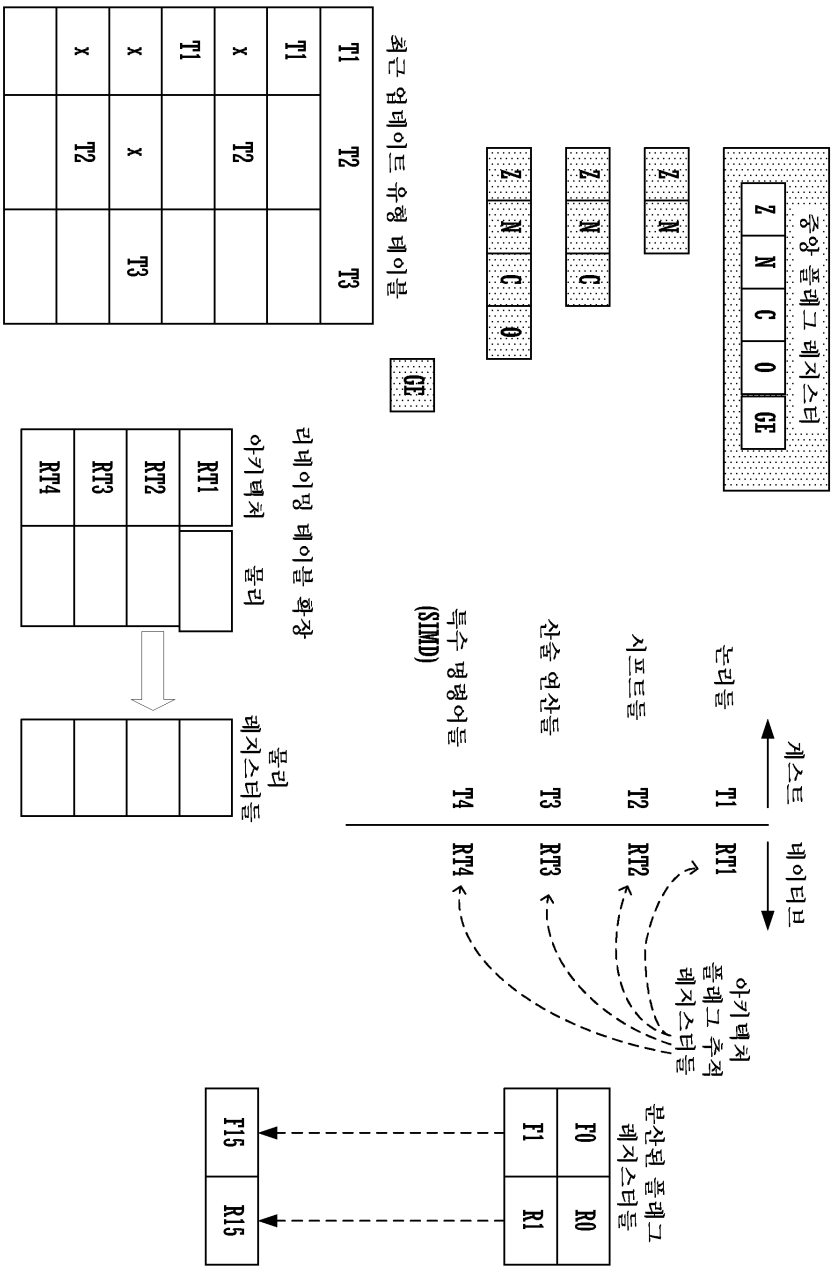
게스트 플래그 아키텍처 에뮬레이션



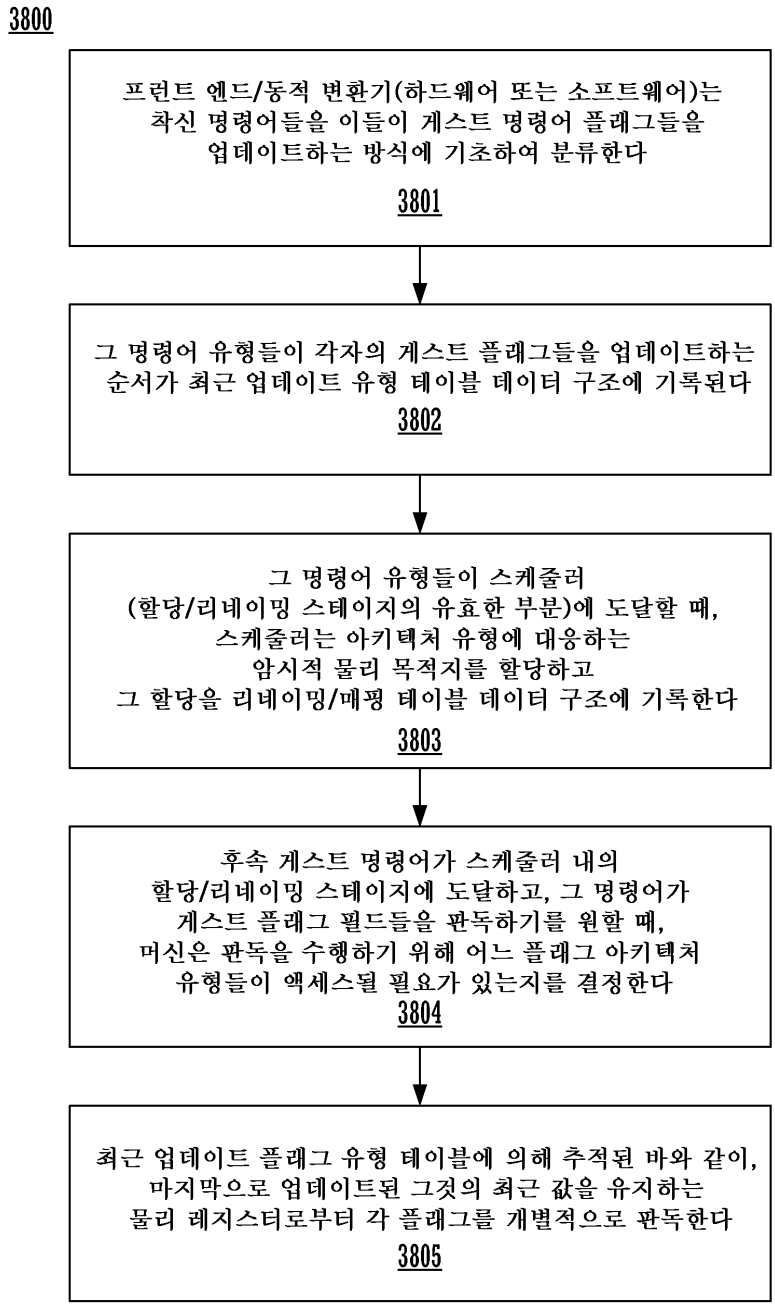
도면36



도면37


중앙 플래그 레지스터
Z
N
C
O
CE
Z
N
Z
N
C
Z
N
C
O
CE
최근 업데이트 유형 테이블
T1
T2
T3
T1
x
T2
T1
x
T3
x
T2
캐스트
네이티브
논리들
시프트들
산술 연산들
특수 명령어들
(SIMD)
T1
T2
T3
T4
RT1
RT2
RT3
RT4
아키텍처
플래그 추적
레지스터들
분산된 플래그
레지스터들
F0
F1
R0
R1
F15
R15
리네이밍 테이블 확장
아키텍처
물리
RT1
RT2
RT3
RT4
물리
레지스터들

도면38


3800
프런트 엔드/동적 변환기(하드웨어 또는 소프트웨어)는 착신 명령어들을 이들이 게스트 명령어 플래그들을 업데이트하는 방식에 기초하여 분류한다
3801
그 명령어 유형들이 각자의 게스트 플래그들을 업데이트하는 순서가 최근 업데이트 유형 테이블 데이터 구조에 기록된다
3802
그 명령어 유형들이 스케줄러 (할당/리네이밍 스테이지의 유효한 부분)에 도달할 때, 스케줄러는 아키텍처 유형에 대응하는 암시적 물리 목적지를 할당하고 그 할당을 리네이밍/매핑 테이블 데이터 구조에 기록한다
3803
후속 게스트 명령어가 스케줄러 내의 할당/리네이밍 스테이지에 도달하고, 그 명령어가 게스트 플래그 필드들을 판독하기를 원할 때, 머신은 판독을 수행하기 위해 어느 플래그 아키텍처 유형들이 액세스될 필요가 있는지를 결정한다
3804
최근 업데이트 플래그 유형 테이블에 의해 추적된 바와 같이, 마지막으로 업데이트된 그것의 최근 값을 유지하는 물리 레지스터로부터 각 플래그를 개별적으로 판독한다
3805