(19) 대한민국특허청(KR)
(12) 등록특허공보(B1)

(51) Int. Cl.[6]
G06F 9/22

(45) 공고일자 1999년11월15일
(11) 등록번호 10-0230552
(24) 등록일자 1999년08월23일

(21) 출원번호 10-1996-0010200
(65) 공개번호 특1996-0038600
(22) 출원일자 1996년04월04일
(43) 공개일자 1996년11월21일
(30) 우선권주장 8/421,272 1995년04월13일 미국(US)

(73) 특허권자 인터내셔널 비지네스 머신즈 코포레이션 포만 제프리 엘
미국 10504 뉴욕주 아몬크
(72) 발명자 마틴 에드워드 홉킨스
미합중국 뉴욕 10514 차파쿠아 더글라스 로드 300
라빈드라 케이. 네어
미합중국 뉴욕 10510 브리어클리프 마노 쉬레이드 로드 52
(74) 대리인 박해천, 원석희

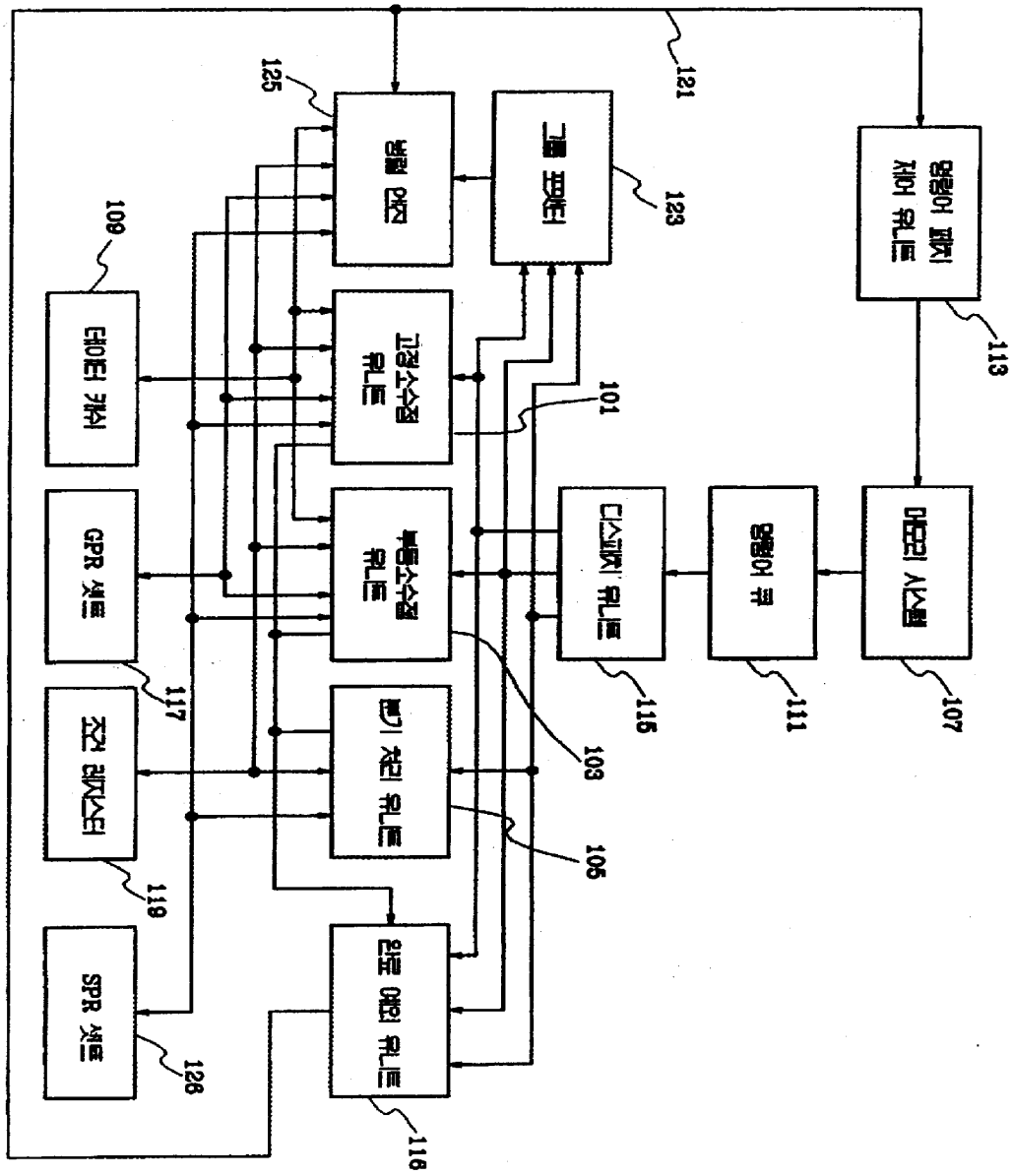
심사관 : 하유정

(54) 동적 명령어 포맷팅을 이용한 컴퓨터 처리 시스템

요약

컴퓨터 처리 장치는 병렬로 실행될 수 있는 명령어를 표현하는 명령어 그룹을 저장하기 위해 사용되는, 디코드된 명령어 버퍼(DIB)로 불리는 버퍼를 포함한다. DIB 그룹 내의 각각의 패턴은 긴 디코드된 명령어(LDI)로 불리는 명령어의 엔코딩이 될 수 있다. DIB는 메모리 시스템, 명령어 큐, 및 실행 유니트의 셋트로 명령어를 전달하는 명령어 디스패치 유니트로 구성된 통상적인 컴퓨터 처리 장치와 함께 동작한다. 어떤 명령어가 상기 DIB에서 활용가능하지 않으면, 그 명령어 및 그 후속 명령어가 상기 메모리 시스템으로부터 상기 명령어 큐로 폐치되어, 통상적인 방법으로 실행된다. 이때 통상적인 장치에 의한 명령어의 실행과 동시에, 그룹 포맷터는 LDI 셋트론 생성하는데, 이 각각의 LDI는 병렬로 실행될 수 있는 원래의 명령어의 셋트의 대체 엔코딩이 될 수 있다. LDI를 구성하는데 있어, 그룹 포맷터는 명령어들 사이의 종속성 및 명령어 대기시간을 분석한다. 그룹 포맷터에 의해 구성된 각각의 LDI 셋트는 DIB에 저장되어, 동일한 명령어 셋트의 다음 실행이 기능 유니트의 풀 컴플리먼트(full complement) 상에서 DIB로부터 직접 실행될 수 있으며, 따라서, 종속성 및 대기시간 분석을 위한 노력을 필요로 하지 않는다.

대표도



명세서

도면의 간단한 설명

도1은 본 발명에 따른 컴퓨터 처리 장치의 구성을 도시한 블록도.

도 2는 도1의 병렬 엔진의 구성을 도시한 블록도.

도3은 도2의 디코드된 명령어 버퍼(DIB)의 엔트리 포맷을 도시한 도면.

도4는 도2의 현재 그룹 버퍼(CGB)의 엔트리 포맷을 도시한 도면.

도5는 도2의 DIB의 기능 블록도.

도6은 도2의 CGB의 기능 블록도.

도7은 도1의 실행 유니트에 의해 실행되는 명령어 시퀀스의 프로그램 그래프.

도8은 도7의 프로그램 시퀀스를 위한 명령어들 사이의 종속성을 도시한 테이블.

도9a 내지 도9c는 도7의 프로그램 시퀀스를 통한 실행가능한 경로를 도시한 도면.

도10a 내지 도10c는 각각 도9a 내지 도9c의 프로그램 시퀀스를 처리하는데 있어서의 도1의 그룹 포맷터의 연산을 도시한 테이블.

도11은 도1의 그룹 포맷터의 기능 블록도.

도12a 및 도12b는 LDI 그룹을 구축하는데 있어서의 도11의 할당기(allocator)의 연산을 도시한 흐름도.

도13은 LDI 그룹을 폐쇄(closing)하는데 있어서의 도11의 할당기의 연산을 도시한 흐름도.

도14a 및 도 14b는 그룹 버퍼에서 할당기가 LDI 그룹을 구축할 때 도11의 그룹 버퍼와 타임 스탬프 테이블의 상태를 도시한 테이블.

도 15는 레지스터 재명명 기술을 이용하여 LDI의 그룹을 구축하는데 있어서의 도11의 할당기의연산을 도시한 흐름도.

도16a 및 도16b는 레지스터 재명명 기술을 이용하는 그룹 버퍼에서 할당기가 LDI 그룹을 구축할 때 도11의 그룹 버퍼와 타임 스탬프 테이블의 상태를 도시한 테이블.

도17은 LDI 그룹으로의 병합을 위해 명령어를 처리하는데 있어서의 할당기의 추가적인 연산을 도시한 흐름도.

도18a 내지 도18d는 로드 연산 및 저장 연산이 순서를 유지하도록 도17에 도시된 추가 연산을 이용하여 그룹 버퍼에서 할당기가 LDI 그룹을 구축할 때, 도11의 그룹 버퍼와 타임 스탬프 테이블의 상태를 도시한 테이블.

도19는 도1의 병렬 엔진에 의해 생성된 그룹의 실행 결과에 따라 그룹을 수정하는데 있어서의 도1의 그룹 포맷터의 연산을 도시한 테이블.

도20은 병렬 엔진의 실행 유니트에 의한 실행을 위해 압축된 LDI를 확장할 수 있는 도2의 CGI의 기능 블록도.

* 도면의 주요부분에 대한 부호의 설명

101 : 고정소수점 실행 유니트　　103 : 부동소수점 실행 유니트

105 : 분기 처리 유니트　　107 : 메모리 시스템

109 : 데이터 캐쉬　　111 : 명령어 캐쉬

113 : 명령어 페치 제어 유니트　　115 : 디스패치 유니트

116 : 완료/예외 유니트　　117 : 범용 레지스터 셋트

119 : 조건 레지스터　　121 : 버스

123 : 그룹 포맷터　　125 : 병렬 엔진

126 : 특수 목적 레지스터 셋트

발명의 상세한 설명

발명의 목적

발명이 속하는 기술분야 및 그 분야의 종래기술

본 발명은 일반적으로 데이터 처리 시스템(Data Processing System)에 관한 것으로서, 특히, 외부 메모리에 저장되어 있는 명령어들의 데이터 처리 시스템에 의한 실행에 관한 것이다.

컴퓨터 아키텍처는 일반적으로 순차적인(Sequential) 프로그램 실행 모델에 기반을 두고 있으며, 이러한 모델에서 프로그램 내의 각각의 명령어는 메모리로부터 페치(fetch)되어 완전히 실행된 다음, 그 모든 결과는 다음 명령어가 그와 비슷하게 처리되기 전에 메모리로 되돌려진다. 이러한 모델에 있어서, 실행된 다음 명령어는 통상적으로 직전에 실행된 명령어의 바로 다음에 오고, 직전에 실행된 명령어보다 그 다음으로 높은 주소를 가진 메모리내의 한 장소(location)에 기억된다. 통상적인 컴퓨터 프로그램 애플리케이션에서 자주 발생하는 분기 명령어(branch instructions)에 있어서는, 다음의 연속되는 장소가 아닌 다른 장소가 지정될 수도 있다.

이러한 순차 모델의 간단한 구현에 있어서, 각각의 명령어는 동일하고 예측 가능하게 처리된다. 따라서, 일련의 명령어의 실행에 필요한 시간은 실행되는 명령어들의 수에 정비례한다. 하나의 명령어가 그 이전의 명령어가 여전히 실행되고 있는 동안 페치도는 파이프라인 방식(pipelined) 또는 오버랩 방식(overlapped)의 구현에 있어서는, 현재 실행되고 있는 명령어들 사이에 종속성(dependencies)이 없는 한 지연은 발생하지 않는다. 종속성은 하나의 명령어가 다른 명령어의 수행결과가 얻어질 때까지 실행을 완료할 수 없을 때 일어난다. 이러한 상황에서, 두번째 명령어는 첫 번째 명령어가 실행을 완료하여 그 결과를 발생할 때까지 대기해야 하며, 따라서 실행에 지연을 유발한다.

파이프라인 방식의 구현에 있어서, 종속성으로 인한 지연을 줄이기 위해 다양한 방법이 사용된다. 첫 번째 방법에 있어서는, 종속적인 명령어(dependent instructions)들이 정확히 같은 시간의 실행에 활용되지 않게 되도록 명령어를 스케쥴링(schedule)하는 컴파일러에 이러한 구현에 대한 세부사항이 입력된다. 이러한 방법에 있어서는, 컴파일러가 모든 명령어들간의 종속성을 관리하며, 구현이 간단하다는 장점이 있다. 결과들은 항상 제때에 활용가능하도록 스케쥴링되기 때문에, 시스템 내부 클럭의 사이클 타임으로 측정되는 시스템 속도에 대한 유일한 제한은 명령어 셋트에서의 단일의 프리미티브 산술 또는 논리 연산(single primitive arithmetic or logical operation)의 복잡도(complexity)에 있다. 이러한 방법의 단점은 컴파일된 애플레이케이션이 단지 특정 구현에 대해서만 호환성이 있으며, 따라서, 이미 존재하는 모든 컴퓨터 프로그램 애플리케이션이 이러한 특정 구현 상에서 실행되도록 하기 위해서는 재컴파일 하여야 하고, 또한, 최종 사용자(end-user)는 재구입해야 한다는 것이다.

두 번째 방법은, 명령어들간의 종속성을 검사하고 이들 명령어를 신중하게 스케줄링하여, 데이터를 생성하는 명령어들이 실행된 이후에만, 이들 명령어가 그 데이터에 억세스하도록 하는 하드웨어를 프로세서에 포함시키는 것이다. 상기 두 번째 방법은 프리미티브 연산의 복잡도 뿐만 아니라, 명령어들 사이의 종속성을 발견하는데 있어서의 복잡도에 의해서도 제한되며, 이러한 복잡도는 실행될 수 있는 오버랩핑(overlapping) 연산의 수가 증가함에 따라 증가한다. 이것은 시스템 내부클럭의 사이클 타임을 증가시키거나, 또는 프로그램 실행을 완료하는데 요구되는 클럭 사이클의 수를 증가시키는 결과를 초래하며, 이들은 둘 다 시스템의 성능을 떨어뜨린다. 그러나, 이러한 방법은 단 한번 컴파일된 프로그램을 서로 다른 구현에서 실행시킬 수 있는 중요한 장점을 갖고 있다.

발명이 이루고자 하는 기술적 과제

본 발명의 한 목적은 데이터처리 시스템의 전체 시스템 성능을 개선하는 것이다.

본 발명의 다른 목적은 명령어들이 프로세서에 의해 실행되는 속도를 향상시키는 것이다.

본 발명의 또다른 목적은 명령어 스트림에서 종속성을 찾는데 요구되는 작업을 최소화하는 것이다.

본 발명의 또 다른 목적은 명령어들이 병렬로 실행될 수 있도록 명령어 스트림의 연산에 관해 프로세서가 동적으로 학습(learn)할 수 있도록 하는 것이다.

본 발명의 또 다른 목적은 명령어들의 병렬로 실행될 수 있도록 복수의 명령어 스트림에 관해 프로세서가 학습할 수 있도록 하는 것이다.

본 발명에 있어서는, 병렬로 실행될 수 있는 명령어를 나타내는 명령어 그룹을 저장하기 위해, 디코드된 명령어 버퍼(Decoded Instruction Buffer; 이하 "DIB"라 칭함)라 불리는 버퍼가 사용된다. DIB 그룹 안의 각각의 패턴은 긴 디코드된 명령어(Long Decoded Instruction; 이하 "LDI"라 칭함)라 불리는 긴 명령어의 엔코딩(encoding)이 될 수 있다. 각 LDI는 예를 들어, 명령어를 실행하는데 사용되는 기능 유니트(functional unit) 및 레지스터 자원(resources)을 포함하여, 명령어를 실행하는데 필요한 모든 정보를 포함할 수 있다. 또한, LDI는 현재 명령어의 결과에 의존하여 실행될 다음 그룹 및/또는 다음 LDI에 대한 포인터 셋트를 포함할 수 있다.

DIB는 메모리 시스템과 명령어 큐(instruction queue), 및 실행 유니트 셋트로 명령어를 공급하는 명령어 디스패치 유니트(dispatch unit)로 구성되어 있는 통상적인 컴퓨터 처리 장치와 함께 동작한다. 어떤 명령어가 DIB에서 활용가능하지 않을 때에는, 그 명령어 및 그 다음 명령어가 메모리 시스템으로부터 명령어 큐로 페치(fetch)되어, 활용가능한 기능 유니트의 서브셋트 또는 그러한 목적에 전용되는 기능 유니트를 필요로 하는 통상적인 방식으로 실행된다. 통상적인 장치에 의한 명령어의 실행과 동시에, 그룹 포멧터(group formatter)는 LDI 셋트를 생성하는데, 각각의 LDI는 병렬로 실행될 수 있는 원래의 명령어 셋트(set of original instruction)의 대체 엔코딩(alternate encoding)이 된다. 이러한 LDI를 구축하는 데 있어, 그룹 포맷터는 명령어들간의 종속성과 명령어 대기시간(latency)을 분석한다. 그룹 포맷터에 의해 구축된 각각의 LDI 셋트는 DIB에 저장되며, 따라서, 동일한 명령어 셋트의 그 다음 실행이 기능 유니트의 풀 컴플리먼트(full complement)상에서 DIB로부터 직접 실행될 수 있으며, 종속성이나 대기시간을 분석하는 노력을 필요로 하지 않게 된다.

DIB는 예를 들어, 원래의 명령어 스트림에서의 명령어 주소와 같은, 명령어 식별자(instruction identifier)에 의해 인덱스되는 캐쉬처럼 구성될 수 있다. 가장 가단한 형태로, 상기 DIB는 메모리처럼 구성되어, 주소로부터의 적절한 비트 셋트를 이용하여 억세스될 수 있다. 이런 경우에, 전체 주소가 내용과 함께 저장되거나 또는 병렬 메모리에 저장되고, 그 그룹을 확인하기 위해 요구되는 주소와 비교된다. 또한, 이러한 억세스와 병행하여 메모리 시스템도 억세스되며, 따라서, 만일 DIB에서 부정합(mismatch)이 발생하면, 메모리 시스템으로부터의 지연 없이 실행이 재개시될 수 있다.

발명의 구성 및 작용

이제, 도1을 참조하면, 본 발명의 컴퓨터 처리 장치는 하나 또는 그 이상의 고정소수점(fixed point) 실행 유니트(101)(1개만 도시됨), 하나 또는 그 이상의 부동소수점(floating point) 실행 유니트(103)(1개만 도시됨), 및 하나 또는 그 이상의 분기 처리 유니트(105)(1개만 도시됨)를 구비한 종래의 시퀀셜 머신을 포함한다. 메모리 시스템(107)은 실행 유니트(101,103,105)에 의해 실행될 명령어를 저장한다. 데이터 캐쉬(109)는 고정소수점 및 부동소수점 유니트(103,105)에 의해 실행되는 명령어와 관련된 데이터를 저장한다. 종래처럼, 메모리 시스템(107)은 메인 메모리와 캐쉬 메모리 서브시스템을 포함할 수 있다. 통상적으로 데이터 캐쉬(109)와 캐쉬 메모리 서브시스템은 셋트 어소시에이티드(set associated) 구조이다. 데이터 캐쉬(109)는 메모리 시스템(107)의 캐쉬 메모리 서브시스템과 분리된 구조(일반적으로 분할 캐쉬(split cache)로 불림)이거나, 또는 대안으로, 메모리 시스템(107)의 캐쉬 메모리 서브시스템의 일부가 될 수 있다(일반적으로 결합 캐쉬(combined cache)로 불림).

명령어 큐(111)는 메모리 시스템(107)으로부터 페치(fetch)된 명령어를 수신하도록 접속된다. 명령어는 메모리 시스템(107)으로부터 개별적으로 또는 블록이라 불리는 그룹으로 페치될 수 있다. 명령어 블록의 페칭은 캐쉬 시스템에 바람직한데, 그 이유는 그것이 피연산자에 대한 더 큰 캐쉬 억세스를 제공하기 때문이다. 현재의 관례상, 하나의 블록은 통상적으로 쿼드워드(QW : 16byte) 또는 8개의 워드(32byte)이지만, 미래의 머신에서 블록 크기가 더 커질 수도 있다. 또한, 명령어는 블록 경계에 정렬되거나 또는 그렇지 않을 수도 있다.

메모리 시스템(107)으로부터 명령어 큐(111)로의 명령어들의 페치는 명령어 페치 제어 유니트(113)에 의해 조정된다. 명령어 페치 제어 유니트(113)의 기능은 주소(address) 생성 및 명령어 페치 주소의 변환을 포함할 수 있다. 명령어 페치 제어 유니트(113)의 주 기능은 메모리 시스템(107)으로부터 명령어 큐(111)로 페치될 명령어의 블록을 식별하는 주소를 I-FETCH 주소 레지스터(도시되지 않음)에 로드하는 것이다.

명령어 페치 제어 유니트(113)는 또한, 메모리 시스템(107)으로부터 명령어 큐(111)로의 명령어의 추론적 프리페치(speculative prefetch)를 제어하는 프리페치 로직(prefetch logic)을 포함할 수 있다. 통상적으로, 이러한 프리페치 로직은 분기 히스토리 테이블(branch history table) 및 관련 제어 로직을 포함한다. 이와 같은 프리페치 로직의 예에 대한 더 자세한 설명은 본 발명의 양수인에게 양도된, Pomerene 등의 미국 특허 제 4,679,141 호에서 찾아볼 수 있으며, 이 특허는 그 전부가 참조로써 본 명세서에 병합되어 있다.

디스패치 유니트(115)는 명령어 큐(111)로부터 명령어를 로드하고, 이들 명령어를 디코드하고 적절한 실행 유니트들(101,103,105)에 의한 실행을 위해 디코드된 명령어를 스케줄링한다. 고정소수점 및 부동소수점 실행 유니트(101,103)에 의해 실행된 명령어의 결과는 갱신을 위해서 범용 레지스터 셋트(117)에 공급된다. 분기 처리 유니트(105)에 의해 실행된 분기 명령어의 결과는 갱신을 위해 조건(condition) 레지스터(119)로 공급된다. 고정소수점 및 부동소수점 실행 유니트(101,103)와 분기 처리 유니트(105)는 종래처럼 특수 목적 레지스터(Special Purpose Register ; 이하 "SPR"이라 칭함) 셋트(126)에 억세스한다. SPR 셋트(126)는 예를 들어, 링크 레지스터(link register), 카운터 레지스터(counter register), 및 고정소수점 예외 레지스터(fixed point exception register)를 포함할 수 있다.

완료/예외 유니트(completion/exception unit; 116)는 실행을 통해 디스패치로부터 명령어를 추적하고, 이들 명령어를 프로그램 순서로 회수한다. 또한, 완료/예외 유니트(116)는 명령어 스트림에서 다음 명령어의 주소를 저장하고, 그 주소를 NEXT-IADDR 버스(121)를 거쳐 명령어 페치 제어 유니트(113)로 출력하는 프로그램 카운터(도시되지 않음)를 포함한다.

종래의 시퀀셜 머신을 운영하는 로직은 Grohoski, G.F.의 "IBM RISC 시스템/6000 프로세서의 머신 구성"-IBM 연구 개발 저널, Vol 34, No 1, Jan 1990, pp37~58-에 기술된 로직과 매우 비슷하다. 일반적으로, 명령어 페치 제어 유니트(113)에 의해 메모리 시스템(107)으로 제공되는 주소는 연속적인 명령어의 셋트를 억세스하고, 그 명령어 셋트를 명령어 큐(111)로 페치하기 위해 사용된다. 다음 사이클에서, 디스패치 유니트(115)는 하나 또는 그 이상의 명령어를 명령어 큐(111)의 하부(bottom)로부터 로드하고, 이들 명령어를 디코딩하고, 적절한 실행 유니트(101,103,105)에 의한 실행을 위해 디코드된 명령어를 스케줄링한다. 예를 들어, 디스패치 유니트(115)는 고정소수점 유니트(101)로 고정소수점 명령어를, 부동소수점 유니트(103)로 부동소수점 명령어를, 그리고 분기 처리 유니트(105)로 분기 명령어를 스케줄링할 수 있다. 또한, 만약 디코드된 명령어 중에 분기 명령어가 없으면, 명령어 큐(111)에 있는 나머지 명령어들이 분기 명령어인지 조사될 수 있으며, 만약 분기 명령어가 발견되면, 현재의 사이클에서 이 분기 명령어는 분기 처리 유니트(105)에 의한 실행을 위해 스케줄링된다.

다음 사이클에서는, 디스패치 유니트(115)로부터 전송된 스케줄링된 명령어가 실행된다. 이 동일한 사이클에서, 완료/실행 유니트(116)는 완료된 명령어의 결과를 리오더링(reorder)하고, 다음 명령어의 주소를 NEXT-IADDR 버스(121)를 경유하여 명령어 페치 유니트(113)에 제공한다.

또한, 시퀀셜 머신의 디스패치 유니트(115)는 소정의 엔코딩된 명령어들을 실행 유니트(101,103,105)에 의한 실행을 위해 변환(translated)된 명령어들을 스케줄링하기 전에 원래의 명령어와 다른 포맷을 가진 하나 또는 그 이상의 명령어로 변환할 수 있다. 예를 들어, 디스패치 유니트(115)는 어떤 복잡한 명령어(예를 들어, CISC-타입)를 하나 또는 그 이상의 덜 복잡한 명령어(예를 들어, RISC-타입)로 변환하고, 실행 유니트(101,103,105)에 의한 실행을 위해 이들 명령어를 스케줄링할 수 있다.

상기한 바와 같이, 원하는 명령어 및 관련 데이터를 포함하는 명령어의 블록이 메모리 시스템(107)의 캐쉬 서브시스템에 존재하고, 분기 처리 유니트(105)가 고정소수점/부동소수점 실행 유니트(101,103)에 의해 생성될 결과를 대기하는데 사이클을 낭비하지 않는 한, 종래의 시퀀셜 머신은 최대로 한 사이클 당 하나의 고정소수점 연산, 하나의 부동소수점 연산, 및 하나의 분기 연산을 실행할 수 있다.

본 발명에 따르면, 실행 유니트(101,103,105)로의 하나 또는 그 이상의 명령어들의 디스패치와 동시에, 상기 디스패치된 명령어들이 분석을 위해 그룹 포맷터(123)에 공급된다. 명령어 실행과 동시에 및/또는 명령어 실행 이후에, 상기 그룹 포맷터(123)는 종속성 및 명령어 대기시간에 대해 명령어들을 분석하고, 상기 분석에 따라 긴 디코드된 명령어(long decoded instructions : LDI)의 그룹을 생성하고, 병렬 엔진(125)의 디코드 명령어 버퍼(DIB)내에 엔트리로서 LDI의 그룹을 저장한다. 각 그룹은 메모리 시스템(107)에 저장된 원래의 명령어 스트림의 시퀀스 S_G를 나타낸다. 특정 그룹의 각각의 LDI는 단일 머신 사이클에서 병렬로 실행될 수 있는 서브시퀀스 S_L(원래의 명령어 스트림의 시퀀스 S_G의 서브시퀀스)의 대체 엔코딩(alternate encoding)을 나타낸다. 그룹 포맷터(123)에 의해 구축된 LDI의 그룹들은 병렬 엔진(125)의 DIB에 저장되어, 명령어들의 동일 시퀀스 S_G의 그 다음 실행이 병렬 엔진(125)의 기능 유니트에 의해 병렬로 이루어질 수 있게 된다. 후술되는 바와 같은 그룹 포맷터와 병렬 엔진의 구현은 향상된 처리량(throughput)을 제공하기 위해 파이프라인 방식(pipelined)이 될 수 있다.

특히, 그룹 포맷터(123)는 그룹 내의 명령어들의 비-순서적인(out-of-order)실행에 대비하도록 그룹 내의 LDI들을 정렬시키며, 따라서, 병렬 엔진(125)의 효율을 극대화시킨다. 더욱이, 주목할 것은 그룹 포맷터(123)는 DIB에 저장된 명령어들과 대체 엔코딩들 사이에는 1대1 대응이 존재하도록 메모리 시스템(107)에 저장된 명령어드를 단순하게 변환하지 않는다는 것이다. 사실상, 그룹 포맷터(123)는 선행의 디스패치된 명령어에 대하여 종속성 및 명령어 대기시간에 관해 디스패치된 명령어들을 분석하고, 이러한 분석에 따라 대체 엔코딩을 이끌어낸다. 그룹 포맷터에 의해 수행되는 분석은 "문맥에 종속적(context dependent)"이기 때문에(즉, 제어 흐름과 선행 명령어들의 연산 제한 조건에 종속됨), 포맷터는 메모리 시스템(107)에 저장되어 있는 특정 명령어에 관계된 복수의 대체 엔코딩(multiple alternate encodings)을 이끌어낼 수 있다. 예를들어, 메모리 시스템(107)에 저장된 두 개의 명령어 시퀀스들을 고려해 보자. 첫 번째 명령어 시퀀스는 메모리 시스템(107)의 장소들 A,B,X,Y에 포함한다. 두 번째 명령어 시퀀스는 메모리 시스템(107)의 장소들 A,C,Y에 명령어들을 포함한다. 본 예에서, 그룹 포맷터(123)는 각각의 시퀀스에 관계된 대체 엔코딩을 생성할 수 있다. 그러나, 그룹 포맷터(123)에 의해 생성된 이들 대체 엔코딩은 둘다 명령어 A와 Y에 관계되어 있다. 이하, 그룹 포맷터(123)에 대해 상세히 설명한다.

도2에 도시된 바와 같이, 병렬 엔진(125)은 현재 그룹 버퍼(Current Group Buffer 이하 "CGB"라 칭함)(204) 및 그 CGB(204)에 저장된 대체 엔코딩된 명령어들을 실행할 수 있는 다수의 실행 유니트를 포함한다. 병렬 엔진(125)의 실행 유니트들은 대칭(symmetric)을 이룰 수있으며, 따라서, 고정소수점, 부동소수점, 및 분기 연산 중 어느 하나를 실행할 수 있다. 다른 실시예에 있는, 병렬 엔진(125)의 실행 유니트들이 비대칭(asymmetric)이 될 수도 있다. 예를 들어, 도2에 도시된 바와 같이, 병렬 엔진(125)은 네 개의 기능 유니트 FU0...FU3과 두 개의 분기 처리 유니트 BP0, BP1을 포함한다. 기능 유니트들은 고정 소수점 실행 유니트 및/또는 부동소수점 실행유니트가 될 수 있다. 이 경우에 있어, 각각의 기능 유니트는 GPR_Bus를 통해 범용 레지스터 셋트(117)에 연결되고, data_cache 버스를 통해 데이터 캐쉬(109)에 연결되고, SPR_bus를통해 SPR 셋트(126)에 연결되는 경로를 갖고 있다. 또한, 고정소수점 연산을 실행하는 병렬 엔진(125)의 실행 유니트들은 주소를 계산 및 변환할 수 있으며, 데이터 캐쉬와 통신할 수 있다. 병렬 엔진(125)의 실행 유니트들의 예에 관한 상세한 설명은 본 발명의 양수인에게 양도되었으며, Ebcioglu와 Groves의 이름으로 출원된 미합중국 특허출원 제 .... 호에서 찾아볼 수 있으며, 이 특허출원은 그 전부가 참조로써 본 병세서에 병합되어 있다. 또한 ,만일 CGB(204)에 저장된 LDI들에 의해 엔코딩된 명령어들의 포맷이 시퀀셜 머신의 실행유니트들 중 하나 또는 그 이상의 실행 유니트와 호환성이 있다면, 시퀀셜 머신의 실행 유니트 상에서 호환성있는 명령어들이 실행될 수 있으며, 따라서 상기 병렬 엔진(125)과 시퀀셜 머신이 실행 유니트를 공유할 수 있도록 허용한다.

일반적으로, DIB(200)는 캐쉬처럼 엔트리의 어레이로서 배열될 수 있으며, 여기서, 각각의 엔트리는 도3에 도시된 바와 같이, 그룹 식별자(Group Identifier:GID)와 그룹 명령어 데이터(Group instruction data ; GRP)를 포함한다. 그룹 식별자(GID)는 엔트리에 의해 엔코딩된 명령어의 그룹 내의 첫 번째 명령어의 주소를 식별한다. 바람직하게는, 그룹 식별자(GID)는 그룹 내의 첫 번째 명령어의 주소의 일부가 된다. 그룹 명령어 데이터(GRP)는 그룹 내의 엔코딩된 LDI 셋트를 표현한다. 부가적으로, 각각의 엔트리는 또한, 예를 들어, DIB(200)가 완전히 채워져 있는 상태(full)일 때, 엔트리의 교체(replacement) 및/또는 겹쳐쓰기(overwriting)를 제어하기 위해 사용될 수 있는 사용 필드 U(도시되지 않음)를 포함할 수 있다. 도5에 도시된 바와 같이, DIB(200)는 종래의 캐쉬와 같이 엔트리의 어레이(300)로서 구성될 수 있다. 예를 들어, DIB(200)는 128 바이트를 각각 포함하는 256개의 엔트리를 포함할 수 있다. 또한, DIB(200)는 풀 어소시어티브(fully associative), 셋트 어소시어티브(set associative), 또는 직접 맵핑(direct mapped) 구조를 가질 수 있다. 도시된 바와 같이, DIB(200)는 각각 128개의 엔트리가 있는 두 개의 셋트로 구성되어 있는 셋트 어소시어티브 방식이다. 이러한 DIB(200)의 구조 및 동작의 상세한 예는 STONE의 "고성능 컴퓨터 아키텍쳐(HIGH-PEREFORMANCE COMPUTER ARCHITECTURE)", pp. 29-69, Addison-Wesley Pub. Co., 1987에서 찾아 볼 수 있으며, 이것은 그 전부가 본 명세서에 참조로써 병합되어 있다.

도4 및 도6에 도시된 바와 같이, CGB(204)는 엔트리의 어레이로서 구성될 수있으며, 여기서, 각각의 엔트리는 전술한 바와 같이, 단일 머신 사이클 내에서 실행될 수 있는 원래의 명령어 셋트의 대체 엔코딩을 나타내는 LDI를 저장한다. 예를 들어, CGB(204)는 도면에 도시된 것처럼 네 개의 엔트리 LDI_0, LDI_1, LDI_2, LDI_3를 포함할 수 있다. 각각의 LDI는 단일 머신 사이클 동안에 도2의 병렬 엔진(125)의 실행 유니트들에 대한 제어를 기술하는 op-필드를 포함한다. 예를 들어, LDI_0은 기능 유니트 FU0, FU1, FU2, FU3를 각각 제어하는 op-필드 $FU0_0$, $FU1_0$, $FU2_0$, $FU3_0$을 포함하고, 분기 처리 유니트 BP0, BP1을 각각 제어하는 op-필드 $BP0_0$, $B01_0$을 포함한다. 본 예에서, LDI_0의 op-필드는 제1머신 사이클에서 병렬 엔진(125)의 실행 유니트를 제어하고, LDI_1의 op-필드는 제1머신 사이클에 연속되는 제2머신 사이클에서 실행 유니트를 제어하고, LDI_2의 op-필드는 제1 및 제2사이클에 연속되는 제3머신 사이클에서 실행 유니트를 제어하고, LDI_3의 op-필드는 제1, 제2, 및 제3사이클에 연속되는 제4머신 사이클에서 실행 유니트를 제어할 수 있다.

특히, LDI_0, LDI_1, LDI_2, 및 LDI_3 중 하나는 그룹 밖으로 분기하도록 분기 처리 유니트, BP0 또는 BP1 중의 하나를 제어함으로써 그 그룹을 종결시키는 BP op-필드를 포함한다. 그룹 밖으로의 분기는 DIB(200)에 저장된 다른 그룹의 주소로 향하거나, 메모리 시스템(107)에 저장된 원래의 시퀀스 내의 명령어의 주소로 향할 수 있다. 예를 들어, 만약 어떤 그룹이 풀 상태(full)로 되면(즉, 각각의 LDI 안에 명령어를 포함하면), LDI_3은 그 그룹 밖으로 분기하도록 분기 처리 유니트 BP0을 제어함으로써 그 그룹을 종결시키는 op-필드 $BP0_3$를 포함할 수 있다. 다른 실시예에 있어서는, LDI_1이 그룹 밖으로 분기하도록 분기 처리 유니트 BP1을 제어함으로써 그 그룹을 종결시키는 op-필드 BP1을 포함할 수 있다. 바람직하게는, 분기 처리 유니트는 NEXT_IADDR 버스(121)상에 그룹 외부의 주소를 배치함으로써 그 그룹 밖으로 분기된다.

또한, 병렬 엔진(125)의 복수의 분기 처리 유니트들이 주어진 머신 사이크 내에서 하나 이상의 분기 명령어를 완료할 수 있기 때문에, 분기 처리 유니트들 중 하나에는 다른 분기 처리 유니트보다 높은 우선순위가 할당되어, 만일 양 분기 처리 유니트들이 각각의 op-필드에 의해 엔코딩된 분기 명령어가 해결(resolved)된 것으로 판단하는 경우에, 가장 높은 우선순위를 가진 분기 처리 유니트가 그 해결된 분기 명령어의 목표 주소를 NEXT_IADDR 버스(121)에 배치하도록 하는것이 바람직하다. 예를 들어, BP0가 BP1보다 높은 우선 순위를 할당받을 수 있다. 이 경우에, BP1에 관련된 LDI들이 op-필드들은 BP0과 관련된 LDI들의 op-필드들에 대응하는 분기 명령어보다 원래의 프로그램 시퀀스에서 나중에 발생되는 분기 명령어에 대응한다. 그러므로, 주어진 LDI에서, 만일 분기 처리 유니트 BP0과 BP1이 둘다 각각의 op-필드들에 의해 엔코딩된 분기 명령어가 해결되었다고 판단하면, 가장 높은 우선순위의 분기 처리 유니트 BP0가 그 해결된 분기 명령어의 목표 주소를 NEXT_IADDR 버스(121)에 배치시키게 되며, 따라서, 원래의 프로그램 시퀀스의 제어흐름을 유지한다.

병렬 엔진(125)의 기능 유니트는 순차적인 순서로 CGB(204)의 LDI들을 실행시킨다. 보다 특정하여 말하면, 제1머신 사이클에서는, LDI_0의 op-필드들이 실행을위해 병렬 엔진(125)의 기능 유니트로 디스패치되고, LDI_1, LDI_2, LDI_3의 op-필드들은 LDI_0, LDI_1, LDI_2로 각각 시프트된다. 이 제1사이클에서, 기능 유니트 FU0...FU3은 LDI_0 내의 각각의 op-필드들에 의해 엔코딩된 명령어들을 실행한다. 또한, 제1사이클에서, 분기 처리 유니트 BP0과 BP1은 조건 레지스터(119)를 검사하고, 그것을 원하는 조건과 비교함으로써, LDI_0

내의 각각의 op-필드들에 의해 엔코딩된 분기 명령어를 실행한다.

만약 분기 처리 유니트(BPO, BP1) 중 단지 하나가 LDI_0의 각각의 op-필드에 의해 엔코딩된 분기 명령어가 해결되었다고 판단하면, 상기 하나의 분기 처리 유니트는 목표 주소가 CGB(204)에 현재 저장된 그룹 내의 LDI를 지시하고 있는지, 또는 그룹 밖의 명령어를 지시하고 있는지를 판단하기 위해, 상기해결된 분기 명령어의 목표 주소를 검사한다. 만일 하나의 분기 처리 유니트가 목표 주소가 그룹 내의 LDI를 지시하는 것으로 판단하면, 상기 하나의 분기 처리 유니트는 그에 따라 LDI들의 op-필드를 시프트하게 되고, 따라서, 다음 머신 사이클에서 병렬 엔진(125)의 기능 유니트 및 분기 처리 유니트들에 의한 실행을 위해 특정 LDI가 디스패치되게 된다. 그러나, 만일 하나의 분기 처리 유니트가 목표 주소가 그룹 밖의 것을 지시하는 것으로 판단하면, 상기 하나의 분기처리 유니트는 목표 주소를 NEXT_IADDR 버스(121)에 제공함으로써, 그룹 밖으로 분기한다.

만일 분기 처리 유니트 BPO과 BP1이 둘다 LDI_0의 각각의 op-필드들에 의해 엔코딩된 분기 명려어가 해결되었다고 판단하면, 가장 높은 우선순위의 분기 처리 유니트, 예를 들어, BPO이 목표 주소가 CGB(204)에 현재 저장된 그룹 내의 LDI를 지시하고 있는지, 또는 그룹 밖의 명령어를 지시하고 있는지를 판단하기 위해 그 해결된 분기 명령어의 목표 주소를 검사한다. 만일 가장 높은 우선순위의 분기 처리 유니트가 상기 목표 주소가 그룹 내의 LDI를 지시한다고 판단하면, 가장 높은 우선순위의 분기 처리 유니트는 그에 따라 LDI들의 op-필드를 시프트하게 되고, 따라서, 다음 머신 사이클에서 병렬 엔진(125)의 기능 유니트 및 분기 처리 유니트에 의한 실행을 위해 특정 LDI가 디스패치되게 된다. 그러나, 만일 가장 높은 우선순위의 분기 처리 유니트가 목표 주소가 그룹 밖을 지시한다고 판단하면, 가장 높은 우선순위의 분기 처리 유니트는목표 주소를 NEXT_IADDR 버스(121)에 배치함으로써, 그 그룹 밖으로 분기한다.

끝으로, 만약 분기 처리 유니트 BPO과 BP1이 둘다 LDI_0의 각각의 op-필드들에 의해 엔코딩된 분기 명령어가 해결되지 않았다고 판단하면, 현재 LDI가 상기한 바와 같이 그룹 밖의 주소를 NEXT_IADDR 버스(121)에 배치해야 하는 그룹 밖으로의 종결 분기(terminating branch)를 포함하지 않는한, 병렬 엔진(125)의 연산은 그 시퀀스의 다음 LDI의 처리로 계속된다.

지금까지는 DIB(200)와 CGB(204)의 구조에 대해 설명하였지만, 이제, 도2 내지 도6을 참조하여 DIB(200)와 CGB(204)의 동작에 대해 설명한다. 설명을 위해, 제1머신 사이클 다음에, 즉시 제2머신 사이클이 오고, 제2머신 사이클 다음에 즉시 제3머신 사이클이 오는 세 개의 머신 사이클을 고려하자. 또한, 제1머신 사이클에서, 종래의 시퀀셜 머신은 명령어를 완료하고, 다음 명령어의 주소를 NEXT_IADDR 버스(121)에 제공한다. 상기한 바와 같이, DIB(200)는 원래의 명령어들의 동일한 셋트들의 다음 실행이 병렬 엔진(125)의 실행유니트들에 의해 병렬로 이루어질 수 있도록, 원래의 명령어들의 셋트를 표현하는 데이터를 저장한다. 한편, CGB(202)는 병렬 엔진(125)의 실행 유니트들에 의해 현재 병렬로 실행되고 있는 원래의 명령어들의 단일 셋트 또는 그룹을 표현하는 데이터를 저장한다.

다음 명령어가 DIB(200)에서 엔코딩된 명령어들의 셋트의 일부인지를 판단하기 위해, NID 생성기(206)는 NEXT_IADDR 버스(121)에 제공된 다음 명령어의 주소에 따라 다음 명령어 식별자(Next instruction Identifier : NID)를 생성한다. 다음 명령어 식별자 NID는 DIB(200)의 엔트리들의 GID와 대응되어야 한다. 예를 들어, 각 엔트리의 GID가 그룹내의 첫 번째 명령어 주소의 일부인 경우에, 다음 명령어 식별자(NID)는 NEXT_IADDR 버스(121)에 제공된 주소의 대응되는 부분이 될 수 있다.

제1머신 사이클에서, 제어 유니트(202)는 도5에 도시된 것처럼 NID 생성기(206)에 의해 생성된 NID를 입력 래치(input latch)(304)에 로드시키기 위해 DIB제어 버스(208)를 통해 DIB(200)을 제어한다. 제1머신 사이클에서, NID 또는 DIB(200)의 입력 래치(304)에 저장된 완전 NID(full NID)의 일부분이 어레이(302)에 대응하는 로우(row)를 활성화시키는 기능을 갖고 있는 디코더(306)에 제공된다. 활성화된 로우의 각각의 엔트리(도시된 바와 같이 두 개의 엔트리임)는 어레이(302)로부터 판독되어, 입력 래치(304)에 저장된 NID와 함께, 히트 인식 및 검출로직(hit recognition and detection logic)(308)에 제공된다. 이 히트 인식 및 검출 로직은 어레이(302)로부터 판독된 엔트리들 중 한 엔트리의 GID가 입력 래치(304)를 경유하여 제공된 NID와 정합(match)하는지를 판단한다. 만일 이러한 조건이 만족되면, 히트 인식 및 검출로직(308)은 DIB 히트 신호를 제어 유니트(202)로 출력하고, 정합하는 엔트리의 GRP 데이터를 출력한다. 만일 그렇지 않다면, 히트 인식 및 검출 로직(308)은 DIB 미스(miss) 신호를 제어유니트(202)로 출력한다.

제1사이클에서, 만일 제어 유니트(202)가 DIB(200)의 히트 인식 및 검출 로직(308)으로 부터 DIB 히트 신호를 수신하면 (즉, 명령어가 DIB(200)에 저장된 그룹의 일부이면), 제어 유니트(202)는 DIB(200)에 의해 출력된 그룹 데이터를 로드하도록 CGB(204)를 제어한다. 그러나, 만일 제어 유니트(202)가 DIB의 히트 인식 및 검출 로직(308)으로부터 DIB 미스 신호를 수신하면(즉, 명령어가 DIB(200)에 저장된 그룹의 일부가 아니면), 제어 유니트(202)는 제2사이클에서 NEXT_IADDR 버스(121) 상에 제공되는 주소에 의해 식별되는 다음 명령어를 실행하도록 시퀀셜 머신을 제어한다.

제2사이클에서, 병렬 엔진(125)의 기능 유니트들과 분기 처리 유니트들은 제1 사이클 동안에 CGB(204)에 로드된 LDI들의 그룹의 LDI_0을 실행한다. 상기한 바와 같이, 병렬엔진(125)의 분기 처리 유니트는 조건 레지스터(119)를 검사하고 그것을 원하는 조건을 비교함으로써, LDI_0의 각각의 op-필드들에 의해 엔코딩된 분기 명령어들을 실행한다.

만일 분기 처리 유니트들 BPO, BP1 중 단지 하나가 LDI_0의 각각의 op-필드에 의해 엔코딩된 분기 명령어가 해결되었다고 판단하면, 상기 하나의 분기 처리 유니트는 목표 주소가 CGB(204)에 현재 저장된 그룹내의 LDI를 지시하고 있는지 또는 상기 그룹 밖의 명령어를 지시하고 있는지를 판단하기 위해 그 해결된 분기 명령어의 목표 주소를 검사한다. 만일 하나의 분기 처리 유니트가 목표 주소가 상기 그룹 내의 LDI를 지시한다고 판단하면, 상기 하나의 분기 처리 유니트는 그에 따라 LDI의 op-필드들을 시프트하게 되고, 따라서, 제3 머신 사이클에서 병렬 엔진(125)의 기능 유니트들과 분기 처리 유니트들에 의한 실행을 위해 특정 LDI가 디스패치되게 된다. 그러나, 만일 하나의 분기 처리 유니트가 목표 주소가 상기 그룹 밖의 것을 지시한다고 판단하면, 상기 하나의 분기 처리 유니트는 목표 주소를 NEXT_IADDR 버스(201)에

제공한다.

만일 분기 처리 유니트 BP0과 BP1이 둘다 LDI_0의 각각의 op-필드에 의해 엔코딩된 분기 명령어가 해결되었다고 판단하면, 가장 높은 우선순위의 분기 처리 유니트, 예를 들어, BP0이 목표 주소가 CGB(204)에 현재 저장된 그룹내의 LDI를 지시하고 있는지 또는 상기 그룹 밖의 명령어를 지시하고 있는지를 판단하기 위해 그 해결된 분기 명령어의 목표 주소를 검사한다. 가장 높은 우선순위의 분기 처리 유니트가 상기 목표 주소가 상기 그룹내의 LDI를 지시한다고 판단하면, 가장 높은 우선순위의 분기 처리 유니트는 그에 따라 LDI들의 op-필드를 시프트하게 되고, 따라서, 제3 머신 사이클에서 병렬 엔진(125)의 기능 유니트들과 분기 처리 유니트들에 의한 실행을 위해 특정 LDI가 디스패치되게 된다. 그러나, 만일 가장 높은 우선순위의 분기 처리 유니트가 목표 주소가 상기 그룹 밖의 것을 지시한다고 판단하면, 가장 높은 우선순위의 분기 처리 유니트는 목표 주소를 NEXT_IADDR 버스(121)에 제공한다.

또한, 제2사이클에서, 만일 LDI_0이 상기와 같이 그룹 밖으로의 종결 분기 명령어를 포함하면, 적절한 분기 처리 유니트가 상기 그룹 밖의 다음 명령어의 주소를 NEXT_IADDR 버스(121)에 제공한다.

제2사이클에서, 병렬 엔진(125)의 분기 처리 유니트들 중 하나가 하나의 주소를 NEXT_IADDR 버스(121)에 제공하면, NID 생성기(206)는 NEXT__IADDR 버스(121)상에 제공된 주소에 따라 다음 명령어 식별자(NID)를 생성하고, 제어 유니트는 DIB(200)에 정합하는 엔트리가 있는지를 판단하기 위해 상기한 바와 같이 생성된 NID를 이용하여 DIB(200)에 억세스한다. 만일 정합하는 엔트리가 발견되면, 제어 유니트(202)는 DIB(200)의 히트 인식 및 검출 로직(308)으로부터 DIB 히트 신호를 수신하고, 제어 유니트(202)는 제3 머신 사이클에서의 실행을 위해 DIB(200)에 의해 출력된 그룹 데이터를 로드하도록 CGB(204)를 제어한다. 그러나, 만일 정합하는 엔트리가 발견되지 않으면, 제어 유니트(202)는 DIB(200)의 히트 인식 및 검출 로직(308)으로 부터 DIB 미스 신호를 수신하고, 제어 유니트(202)는 제3 머신 사이클에서 ㅁ 상에 제공된 주소에 의해 식별된 다음 명령어를 실행하도록 시퀀셜 머신을 제어한다.

끝으로, 제2 사이클에서, 만일 분기 처리 유니트 BP0과 BP1이 둘다 LDI_0의 각각의 op-필드들에 의해 엔코딩된 분기 명령어가 해결되지 않았다고 판단하면, 병렬 엔진(125)의 연산은 제3 사이클로 계속되고, 여기서는, 시퀀스 내의 다음 LDI, 즉 LDI_1이 제 2 머신 사이클에서의 LDI_0의 처리에 관하여 전술한 바와 같이 처리된다.

그러므로, 다음 주소가 CGB(204)내의 LDI를 지시하거나, 또는 DIB(200)의 엔트리를 지시하는 한, 병렬 엔진(125)은 매 사이클마다 하나의 LDI를 회수한다.

상기한 바와 같이, 병렬 엔진(125)의 분기 처리 유니트들 중 단지 하나의 유니트만이 상기 시퀀스에서 다음 명령어가 DIB(200)에 존재하는지를 판단하기 위해 DIB(200)에 억세스한다. 그러나, 본 발명은 이러한 관점에 제한되지 않는다. 더욱 복잡한 실시예에 있어서, DIB(200)은 멀티포트(multi-ported)될 수 있으며, 따라서, 예를 들어, 병렬 엔진(125)의 분기 처리 유니트 BP0, BP1이 모두 DIB(200)에 억세스할 수 있도록 허용한다. 이 경우에, DIB(200)의 구조는 다수의 정합하는 엔트리들이 DIB로부터 판독될 수 있도록 변경될 수 있으며, 가장 높은 우선순위를 가진 분기 처리 유니트에 대응하는 정합하는 엔트리가 선택되어, 처리를 위해 CGB(204)에 로드된다. 이러한 접근방법은 시퀀스에서 다음 명령어가 DIB(200)에 존재하는지를 판단하기 위해 요구되는 시간을 최소화하기 위해 사용될 수 있다.

시퀀셜 머신의 실행 유니트들에 의한 명령어들의 실행과 동시에 및/또는 그 이후에, 그룹 포맷터(123)는 종속성 및 명령어 대기시간에 관해 이들 명령어를 분석하고, 상기 분석에 따라 긴 디코드된 명령어들(LDIs)의 그룹을 생성하고, 이들 명령어들의 다음 실행이 병렬 엔진(125)에 의해 병렬로 이루어질 수 있도록, 이들 LDI들의 그룹을 병렬 엔진(125)의 DIB에 엔트리로서 저장한다.

이하, 도7 내지 도10에 도시된 예를 이용하여 원래의 명령어 시퀀스의 분석에 따라 LDI의 그룹을 생성하는데 있어서의 그룹 포맷터(123)의 동작에 대해 설명한다.

도7은 시퀀셜 머신의 실행 유니트들(101,103,105)에 의해 실행되기 위해 메모리 시스템(107)에 저장된 명령어 시퀀스의 예를 설명하는 프로그램 그래프이다. 명령어 a, b 및 c를 뒤이어 분기 명령어 bcx가 뒤따르는데, 이 분기 명령어는 조건 x를 검사하고, 만일 x=0 이면 명령어 d로 분기하고, x=1 이면 명령어 k로 분기한다. 명령어 k로부터의 경로 상에는 명령어 l, m, n 및 o가 있다. 명령어 d로부터의 경로 상에는 명령어 e, f, g 및 다른 분기 명령어 bcy가 있으며, y=1이면 명령어 h로 분기한다. 이들 두개의 경로는 다시 명령어 i, j로 병합된다.

도8은 상기 시퀀스에서의 각각의 명령어의 종속성을 도시하고 있다. 이 시퀀스에서 명령어 a, b는 다른 명령어들에 종속되지 않는다. 명령어 c는 양 명령어 a와 b에 종속되며, 이것은 이 명령어 c가 명령어 a와 b가 실행완료된 이후에만 실행될 수 있다는 것을 의미한다. 다른 명령어들도 비슷하다. 끝으로, 분기 명령어 bcx는 명령어 a의 결과에 종속되며, 분기 명령어 bcy는 명령어 e의 결과에 종속된다.

도9는 시퀀셜 머신의 실행 유니트(101,103,105)에 의한 명령어 시퀀스의 3가지 실행예를 예시하고 있다. 도9a에 도시된 바와 같이, 제1 예에서는, 분기 명령어 bcx가 x=0에 대응하는 경로를 갖고 있고, 분기 명령어 bcy는 y=1에 대응하는 경로를 갖고 있다. 도9b에 도시된 바와 같이, 제2 예에 있어서는, 분기 명령어 bcx가 다른 경로, 즉, x=1에 대응하는 경로를 갖고 있다. 끝으로, 도9c에 도시된 바와 같이, 제3 에에 있어서는, 분기 명령어 bcx가 x=0에 대응되는 경로를 갖고 있고, 분기 명령어 bcy는 y=0에 대응하는 경로를 갖고 있다.

도10a는 도9a의 명령어 시퀀스의 분석에 따라 LDI(LDI_0..LDI_3)의 그룹을 생성하는데 있어서의 그룹 포맷터(123)의 동작을 예시하고 있다. 이 명령어 시퀀스는 명령어 a와 b로 시작된다. 명령어 a 또는 b 어느 것도 다른 명령들에 종속되지 않기 때문에, 그룹 포맷터(123)는 명령어 a와 b에 각각 대응되는 op-필드 FU0과 FU1을 생성하고, 도시된 바와 같이, 이들 op-필드를 LDI_0에 배치한다. 명령어 a와 b 다음에는 명령어 c가 뒤따른다. 명령어 c는 명령어 a와 b 둘다에 종속되기 때문에, 그룹 포맷터(123)는 명령어 c에 대응하

는 op-필드 FU0을 생성하고, 도시된 바와 같이, 이 op-필드를 LDI_1에 배치한다.

다음에는, 분기 명령어 bcx가 온다. 만약 이 분기 명령어가 순서대로 (in-order)(도면에서 보이는 것처럼) 배치될 예정이면, 이 분기 명령어는 그 분기 명령어에 선행하는 명령어의 LDI보다 앞서지 않게 LDI에 배치되어야 한다. 만일 그룹의 LDI들을 실행할 때, 병렬 엔진(125)의 엔코딩된 분기 명령어의 분기 조건이 만족되었다고 판단하면, 그 분기에 선행하는 명령어들이 완료되어야 하기 때문에 이러한 조건이 요구된다. 만일 분기 명령어가 비-순서적으로 배치될 예정이면, 각각의 분기 명령어에 대해, 특정 분기 명령어에 선행하는 명령어에 대응하는 최종 LDI를 나타내는 추가적인 정보가 저장되어야 한다. 이와 같은 경우에 있어서, 그룹의 LDI를 실행할 때, 만약 병렬 엔진(125)이 엔코딩된 분기 명령어의 분기 조건이 만족되었다고 판단하면, 이 병렬 엔진은 상기 분기 명령어에 선행하는 명령어들이 완료되도록, 상기 저장된 정보가 나타내는 바와 같은 최근의(latest) LDI까지의 LDI들을 실행해야한다.

그러므로, 분기 명령어의 순서적인 실행을 위해(도면에서 보이는 것처럼), 그룹 포맷터(123)는 분기 명령어 bcx에 대응하는 op-필드 BP0을 생성하고, 상기 op-필드 BP0을 LDI_1에 배치한다. 또한, 그룹 포맷터(123)에 의해 op-필드 BP0에 엔코딩된 제어흐름은 시퀀셜 머신에 의해 판단히 이루어지는 바와 같이 분기 명령어 bcx의 결과에 대응하는 것이 바람직하다. 그러므로, 시퀀셜 머신이 분기 명령어 bcx를 만날 때 조건 x=0이 만족되는 것으로 판단했기 때문에, 도시된 바와 같이, LDI_1의 BP0은 만약 x=1이면, 병렬 엔진(125)이 그룹으로부터 라벨 LK(즉, 명령어 k)로 분기해야 한다는 것을 나타낸다.

이 시퀀스에서 분기 명령어 bcx 다음에는 명령어 d가 뒤따른다. 명령어 d는 명령어 a와 c에 종속되기 때문에, 그룹 포맷터(123)는 명령어 d에 대응하는 op-필드를 생성하고, 이 op-필드를 LDI_2에 배치한다. 그룹 포맷터(123)는 또한, 상기 bcx의 조건이 x=0을 만족하는 경우에만, 명령어 d가 병렬 엔진(125)에 의해 실행될 것이라는 것을 나타내는, 명령어 d에 관련된 플래그를 셋팅할 수 있다. 명령어 d 다음에는 명령어 e가 뒤따른다. 명령어 e는 다른 명령어들에 종속되지 않기 때문에, 그룹 포맷터(123)는 명령어 e에 대응하는 op-필드 FU2를 생성하고, 이 op-필드를 LDI_0에 배치한다. 그룹 포맷터(123)는 또한, 상기 bcx의 조건이 x=0을 만족하는 경우에만, 명령어 e가 병렬 엔진(125)에 의해 실행될 것이라는 것을 나타내는, 명령어 e에 관련된 플래그를 셋팅할 수있다. 이 시퀀스에서 다음은 명령어 f가 온다. 이 명령어 f는 명령어 e에 종속되기 때문에, 그룹 포맷터(125)는 명령어 f에 대응하는 op-필드 FU1을 생성하고, 이 상기 op-필드를 LDI_1에 배치한다. 그룹 포맷터(123)는 또한, 상기 bcx의 조건이 x=0을 만족하는 경우에만, 명령어 f가 병렬 엔진(125)에 의해 실행될 것이라는 것을 나타내는, 명령어 f에 관련된 플래그를 셋팅할 수 있다. 이어서, 명령어 g가 뒤따른다. 명령어 g는 명령어 e와 f에 종속되기 때문에, 그룹 포맷터(123)는 명령어 g에 대응하는 op-필드 FU1을 생성하고, 이 op-필드를 LDI_2에 배치한다. 그룹 포맷터(123)는 또한, 상기 bcx의 조건이 x=0을 만족하는 경우에만, 명령어 g가 병렬 엔진(125)에 의해 실행될 것이라는 것을 나타내는, 명령어 g에 관련된 플래그를 셋팅할 수 있다.

명령어 g 다음에는 분기 명령어 bcy가 뒤따른다. 이 분기 명령어 bcy는 그 분기 명령어 bcy에 선행하는 명령어들의 LDI보다 앞서지 않게 LDI에 배치되어야 하기 때문에, 그룹 포맷터(123)는 분기 명령어 bcy에 대응하는 op-필드 BP0을 생성하고, 이 op-필드를 LDI_2에 배치한다. 또한, 그룹 포맷터(123)에 의해 op-필드 BP0에 엔코딩된 제어 흐름은 시퀀셜 머신에 의해 판단이 이루어지는 바와 같이 분기 명령어 bcy의 결과에 대응하는 것이 바람직하다. 그러므로, 상기 시퀀셜 머신이 상기 분기 명령어 bcy를 만날 때 조건 y=1이 만족된 것으로 판단했기 때문에, 도면에 도시된 바와 같이, 만약 LDI_2의 op-필드 BP0은 y=0이면 병렬엔진(125)이 그룹으로부터 라벨 LP(즉, 명령어 p)로 분기해야 한다는 것을 나타낸다.

이 분기 명령어 bcy 다음에는 명령어 h가 뒤따른다. 이 명령어 h는 명령어 b와 f에 종속되기 때문에, 그룹 포맷터(123)는 명령어 d에 대응하는 op-필드 FU2를 생성하고, 이 op-필드를 LDI_2에 배치한다. 그룹 포맷터(123)는 또한, 상기 bcx의 조건이 x=0을 만족하고 bcy의 조건이 y=1을 만족하는 경우에만, 명령어 h가 병렬 엔진(125)에 의해 실행될 것이라는 것을 나타내는, 명령어 h에 관련된 플래그를 셋팅할 수 있다. 명령어 h 다음에는 명령어 i가 뒤따른다. 이 명령어 i는 명령어 d와 g에 종속되기 때문에, 그룹 포맷터(123)는 명령어 i에 대응하는 op-필드 FU0을 생성하고, 이 op-필드를 LDI_3에 배치한다. 그룹 포맷터(123)는 또한, 상기 bcx의 조건이 x=0을 만족하고 bcy의 조건이 y=1을 만족하는 경우에만, 명령어 i가 병렬 엔진(125)에 의해 실행될 것이라는 것을 나타내는, 명령어 i에 관련된 플래그를 셋팅할 수 있다.

이 시퀀에서 다음은 명령어 j가 이어진다. 이 명령어 j는 명령어 i에 종속되고 더 이상의 활용가능한 LDI가 없기 때문에(즉, 명령어 i는 그룹의 마지막 LDI, 즉, LDI_3에 엔코딩됨), 명령어 j는 현재 그룹에 배치될 수 없다. 이 경우에, 그룹 포맷터(123)는 상기 bcx의 조건이 x=0을 만족하고 bcy의 조건이 y=1을 만족하는 경우에만, 명령어 j의 주소(즉, 도시된 바와 같이 라벨 LJ)로의 분기를 나타내는 op-필드를 생성함으로써 그룹을 폐쇄한다.

그룹을 폐쇄한 후에, 그룹 포맷터(123)는 이 폐쇄된 그룹을 DIB에 기록하고, 그 다음에 명령어 j에 대응하는 op-필드를 FU0을 생성하고 이 op-필드를 새로운 그룹의 LDI_0에 배치함으로써, 새로운 그룹을 시작한다. 다음에, 그룹 포맷터(123)의 동작은 새로운 그룹의 명령어 시퀀스에 대해 전술한 바와 같이 계속된다.

이와 유사하게, 도 10b와 도10c는 각각 도9b와 도9c의 명령어 시퀀스에 따라 그룹 포맷터(123)에 의해 실행되는 그룹의 편성을 예시하고 있다.

지금까지 그룹 포맷터(123)의 동작을 설명하였지만, 이제는 그룹 포맷터(123)의 하드웨어적 구현의 특정 실시예에 관해 설명한다. 도11에 도시된 바와 같이, 그룹 포맷터(123)는 할당기(allocator)(501), op-필드 엔코더(OP-field encoder)(503), 타임 스탬프 테이블(Time Stamp Table : 이하 "TST"라 칭함)(505) 및 그룹 버퍼(group buffer)(507)를 포함할 수 있다.

여기서, op-필드 엔코더(503)는 시퀀셜 머신의 실행 유니트들에 의해 실행을 위해 디스패치 유니트

(115)에 의해 디스패치된 명령어를 수신하고, 시퀀셜 머신의 분기 처리 유니트(105)에 의해 생성된 분기 결과(Branch outcome : 이하 "BR"라 칭함) 데이터도 수신한다. 이들 명령어 각각에 대해, op-필드 엔코더(503)는 그 명령어에 대응하는 적어도 하나의 op-필드를 생성한다. op-필드 엔코더(503)에 의해 수행되는 엔코딩은 병렬 엔진(125)의 기능 유니트 및 분기 처리 유니트들과 호환성 있는 포맷으로 명령어를 포맷팅하는 것과 병렬 엔진(125)의 기능 유니트 및 분기처리 유니트들과 호환성 있는 하나 또는 그 이상의 op-필드로 명령어를 변환하는 것을 포함할 수 있다. 명령어들을 변환하는데 있어서의 op-필드 엔코더(503)의 동작에 대한 자세한 설명은 일례로 본 발명의 양수인에게 양도되고 P. Emma의 이름으로 본 발명과 동시에 출원된 "Method and Apparatus for the Transparent Emulation of an Existing Instruction-Set Architecture by an Arbitrary Underlying Instruction-Set Architecture)"라는 제목의 미국 특허출원에서 찾아볼 수 있는데, 이 특허출원은 그 전부가 본 명세서에 참조로써 병합되어 있다. 또한, 분기 명령어들에 대해, op-필드 엔코더(503)에 의해 수행되는 엔코딩은 시퀀셜 머신의 분기 처리 유니트(105)로부터 제공되는 BR 데이터에 의해 명시되는 바와 같이 분기 명령어의 결과에 대응하는 것이 바람직하다.

TST(505)는 명령어가 이용하거나 또는 정의할 수도 있는 병렬 엔진(125)의 각각의 레지스터 자원과 관련된 타이밍 정보를 포함한다. 일반적으로, 할당기(501)는 LDI들의 현재 그룹을 구축하기 위해 op-필드 엔코더(503)에 의해 생성된 op-필드와 TST(505)에 저장된 타이밍 정보를 이용한다. 그룹 버퍼(507)는 그룹이 할당기(501)에 의해 구축되고 있을 때 LDI의 현재 그룹을 저장하기 위해 사용된다.

특히, TST(505)는 병렬 엔진(125)의 관련 레지스터 자원이 활용가능한(즉, 상기 레지스터 자원을 정의한 그룹의 이전의 명령어들이 완료되게 됨) 현재 그룹의 LDI를 식별하는 타임 스탬프 식별자를 저장한다. 예를 들어, 병렬 엔진(125)은 PowePC 마이크로프로세서에서 통상적인 바와 같이 76개의 레지스터를 포함할 수 있으며, 여기서, 76개의 레지스터는 32개의 범용 레지스터와, 32개의 부동소수점 레지스터와, 1개의 링크(LR) 레지스터와, 1개의 카운트(CTR) 레지스터와, 1개의 고정소수점 예외 레지스터(XER), 및 1개의 부동소수점 상태 및 제어(FPSCR) 레지스터를 포함한다. 이러한 경우에 있어서, TST(505)는 관련 레지스터가 활용가능한 현재 그룹의 LDI를 식별하는 식별자를 각각 저정하는 76개의 셀(cell)을 포함할 수 있다.

시퀀셜 머신의 실행 유니트들에 의한 실행을 위해 디스패치된 각각의 명령어에 대해, 할당기(501)는 이들 명령어에 대응하는 LDI의 그룹들을 편성하고 저장하기 위해 도12 및 도13의 흐름도에 도시된 바와 같이, TST(505), op-필드 엔코더(503) 및 그룹 버퍼(507)와 대화한다. 할당기(501)는 예를 들어, 하나의 시퀀서(sequencer) 또는 조합 로직(combination logic)을 포함시켜 구현될 수 있다. 단계(600)에서 시작하여, 할당기(501)는 특정 명령어가 조건부 분기 명령어인지를 검사한다. 만일 그렇다면, 단계(602)에서, 할당기(501)는 op-필드 엔코더(503)가 그룹 버퍼(507)내의 current_max 플래그에 대응하는 LDI의 활용가능한 슬롯에 상기 명령어에 대응하는 op-필드를 기록하도록 제어하고, 상기 명령어에 의해 정의된 각각의 자원에 대해, 상기 자원과 타임 스탬프 식별자를{current_max 플래그+1}로 갱신하고, 갱신된 타임 스탬프 식별자를 TST(505)에 기록한다. current_max 플래그는 조건부 분기 명령어들이 순서적으로 유지될 수 있도록 현재의 최대 LDI를 추적한다. 만일 조건부 분기 명령어가 비-순서적으로 배치될 것이라면, 각각의 분기 명령어들에 대해, 특정 분기 명령어에 선행하는 명령어들에 대응하는 최종 LDI를 나타내는 추가 정보가 저장되어야 한다. 이러한 경우에 있어서, 상기 그룹의 LDI들을 실행할 때, 만일 병렬 엔진(125)이 엔코딩된 조건부 분기 명령어의 분기 조건들이 만족되었다고 판단하면, 병렬 엔진은 조건부 분기 명령어에 선행하는 명령어들이 완료되도록, 저장된 정보에 의해 지시된 바와 같이 최근의 LDI까지 LDI들을 실행한다. 단계(602) 이후에, 연산은 단계(600)로 복귀하여, 그 시퀀스의 다음 명령어를 현재 그룹으로의 통합을 위해 처리하기 시작한다.

만일 단계(600)에서, 할당기가 상기 명령어가 조건부 분기 명령어가 아니라고 판단하면, 연산은 단계(601)로 계속된다. 단계(601)에서는, 특정 명령어에 의해 사용된 각각의 자원에 대해, 할당기(501)는 그 자원이 활용가능하게 될 LDI를 식별하는, 그 자원에 대응하는 타임 스탬프 식별자를 TST(505)로부터 검색한다. 단계(603)에서, 할당기(501)는 단계(601)에서 검색된 타임 스탬프 식별자 중에서 최대의 것을 판단하는데, 이것은 설명의 편의상 TS_{use}(MAX)로 레이블링된다. 다음에, 단계(605)에서는, 특정 명령어에 의해 정의된 각각의 자원에 대해, 할당기(501)는 상기 자원이 활용가능하게 될 LDI를 식별하는, 자원에 대응하는 타임 스탬프 식별자를 TST(505)로부터 검색한다. 단계(607)에서, 할당기(501)는 단계(605)에서 검색된 타임 스탬프 식별자 중에서 최대의 것을 판단하는데, 이것은 설명의 편의상 TS_{def}(MAX)로 레이블링된다.

단계(609)에서, 할당기(501)는 특정 명령어가 그룹 내의 명령어들에 의해 이미 정의되어 있는 자원을 정의하는 경우에(즉, 어떤 자원을 재정의하는 경우에), 그 특정 명령어가 이러한 명령어들에 선행하도록 TS_{use}(MAX)와 TS_{def}(MAX) 중에서 더 큰 것으로 earliest_LSI 플래그를 셋팅한다. 단계(613)에서, 할당기(501)는 earliest_LDI 플래그가 3보다 큰지 검사함으로써 현재 그룹 내에 상기 명령어를 위한 빈공간이 있는지(즉, 그룹이 풀(full) 상태인지)를 판단한다. 만일 그렇다면, 그룹은 풀(full) 상태이고, 연산은 후술되게 되는 바와 같이 도13에 예시된 단계로 계속되어, 그 그룹을 폐쇄하고, 그 폐쇄된 그룹을 그룹 버퍼(507)로부터 병렬 엔진(125)의 DIB로 기록한다. 만일 그룹이 풀(full) 상태가 아니면, 연산은 단계(615)로 계속되어, earliest_LDI 플래그에 대응하는 LDI 내의 슬롯이 활용가능한지를 판단한다. 만일 슬롯이 활용가능하지 않다면, 단계(617)에서, earliest_LDI 플래그가 증가되고, 연산은 단계(613)로 복귀한다. 그러나, 만일 단계(615)에서 earliest_LDI 플래그에 대응하는 LDI 내의 슬롯이 활용가능하다면, 연산은단계(619) 내지 단계(621)로 계속된다. 단계(619)에서, 할당기(501)는 op-필드 엔코더(503)가 그 명령어에 대응하는 op-필드를 그룹 버퍼(507)에 저장된 LDI의 활용가능한 슬롯으로 기록하도록 제어한다. 단계(621)에서는, 명령어에 의해 정의된 각각의 자원에 대해, 할당기(501)는 상기 자원에 관련된 타임 스탬프 식별자를 {earliest_LDI 플래그+1}로 갱신하고, 그 갱신된 타임 스탬프 식별자를 TST(505)에 기록한다.

다음에, 단계(622)에서, 할당기(501)는 current_max 플래그가 earliest_LDI 플래그보다 작은지를 판단하고, 만일 그렇다면, current_max 플래그를 earliest_LDI 플래그로 셋팅하여, currnet_max 플래그를 갱신한다.

본 발명의 바람직한 실시예에서, 특정 명령어들은 정지 명령어(stopping instruction)(즉, 그룹이 풀(full) 상태가 아니더라도, 그룹이 자동적으로 폐쇄되도록 하는 명령어)로 지정될 수도 있다. 예를 들어,

레지스터 분기 명령어가 정지 명령어로 지정될 수 있다. 이러한 경우에, 단계(623)에서, 할당기(501)는 그 명령어가 정지 명령어인지를 검사한다. 만약 상기 명령어가 정지 명령어이면, 연산은 후술되는 바와 같은 도13에 예시된 단계로 계속되어, 그룹을 폐쇄하고, 그 폐쇄된 그룹을 그룹 버퍼(505)로부터 병렬 엔진(125)의 DIB에 기록한다. 그러나, 만일 단계(623)에서, 할당기가 상기 명령어가 정지 명령어가 아니라고 판단하면, 할당기(501)의 연산은 단계(601)로 복귀하여, 그 시퀀스의 다음 명령어를 처리하게 된다.

그룹을 폐쇄하고, 그 폐쇄된 그룹을 그룹 버퍼(507)로부터 병렬 엔진(125)의 DIB에 기록하는데 있어서의 할당기(501)의 연산에 대해서는 도13에 상세하게 예시되어 있다. 단계(605)에서, 할당기(501)는 레지스터 분기 명령어로 인해 상기 그룹이 정지되고 있는지를 검사함으로써 시작된다. 이것은 단계(623)에서 플래그 셋트를 검사함으로써 수행될 수 있다. 만일 단계(650)에서, 할당기(501)가 상기 명령어가 레지스터 분기 명령어 이라고 판단하면, 단계(652)에서 할당기는 op-필드 엔코더(503)가 상기 명령어에 대응하는 op-필드를 그룹 버퍼(507)의 current_max 플래그에 대응하는 LDI의 활용가능한 슬롯에 기록하도록 제어하고, 상기 명령어에 의해 정의된 각각의 자원에 대해, 상기 자원에 관련된 타임 스탬프 식별자를 [current_max flag+1]로 갱신하고, 그 갱신된 타임 스탬프 식별자를 TST(505)에 기록한다.

만약 단계(650)에서, 할당기(501)가 상기 명령어가 분기 명령어가 아니라고 판단하면, 상기 그룹은 풀(full) 상태로 되어야 한다. 그러므로, 단계(651)에서, 할당기는 current_max 플래그에 대응하는 LDI의 BP0 슬롯이 활용가능한지를 판단한다. 만일 그렇다면, 단계(653)에서, 할당기(501)는 종결 분기 명령어(terminating branch instruction)를 나타내는 op-필드를 그룹 버퍼(505)의 활용가능한 BP0 슬롯에 배치하고, 연산은 단계(655)로 계속된다. 종결 분기 명령어의 목표 주소(Target address)는 그 시퀀스에서 실행될 다음 명령어의 주소를 나타낸다.

그러나, 만일 단계(651)에서 BP0 슬롯이 활용가능하지 않으면, 연산은 단계(657)로 계속되며, 여기서, 할당기(501)는 종결 분기 명령어(terminating branch instruction)를 나타내는 op-필드를 그룹 버퍼(507)의 current_max 플래그에 대응하는 LDI의 활용가능한 BP1 슬롯에 배치하고, 연산은 단계(655)로 계속된다. 종결 분기 명령어의 목표 주소는 그 시퀀스에서 실행될 다음 명령어의 주소를 나타낸다.

단계(655)에서, 할당기(501)는 그룹 버퍼(507)를 병렬 엔진(125)의 DIB에 기록한다. 단계(659)에서, 할당기(501)는 그룹 버퍼(507)를 클리어(clear)시킨다. 끝으로, 단계(661)에서, 할당기(501)는 새로운 그룹으로의 통합을 위해 시퀀스의 다음 명령어 처리를 시작하기 위해 도12에 대하여 전술한 단계들로 복귀하기 전에, earliest_LDI 플래그와 TST(505)를 리셋한다.

전술한 바와 같은 그룹 포맷터(123)의 연산은 간단한 예를 이용하면 양호하게 설명된다. 다음의 명령어 시퀀스를 고려해 보자.

1. a r1=r2, r3 (R1=R2+R3)

2. a r4=r1, r5 (R4=R1+R5)

3. mul r3=r1, r4 (R3=R1*R4)

4. a r4=r2, r5 (R4=R2+R5)

5. inc r4 (R4=R4+1)

6. mul r2=r3, r4 (R2=R3*R4)

도14a 및 도14b는 그룹 포맷터(123)가 도12 및 도13에 대하여 전술한 바와 같이 상기 명령어 시퀀스를 그룹으로 포맷팅하기 위해 동작할 때 TST(505)와 그룹 버퍼(507)의 상태를 예시하고 있다.

상기 그룹의 개시부분에서, 첫 번째 명령어를 처리하기 전에, TST(505)는 각각의 셀에 제로(zero)를 포함하도록 초기화된다. 첫 번째 명령어에 있어서는, 그 명령어에 의해 사용되는 자원, r2와 r3가 0이라는 대응하는 타임 스탬프 식별자를 갖고 있기 때문에, 할당기는 단계(603)에서 TS_{use}(MAX)를 0으로서 정의한다. 또한, 명령어에 의해 정의된 자원 r1이 0이라는 대응하는 타임 스탬프 식별자를 갖고 있기 때문에, 할당기는 단계(607)에서 TS_{def}(MAX)를 0으로서 정의한다. 단계(609)에서, 할당기(501)는 earliest_LDI 플래그를 0으로 셋팅하고, 흐름은 단계(613)로 계속된다. 단계(613)에서, 이제 제로로 세팅된 earliest_LDI 플래그가 3보다 크지 않기 때문에, 단계(615)에서 할당기(501)는 earliest_LDI 플래그에 대응하는 LDI(이 경우에는 LDI_0)에서 슬롯 FU0...FU3이 활용가능한지를 검사한다. 슬롯 FU0가 LDI_0에서 활용가능하기 때문에, 연산은 단계(619)로 계속되고, 여기서, 할당기(501)는 도14b에 도시된 바와 같이, LDI_0의 FU0 슬롯에 첫 번째 명령어에 대응하는 op-필드를 배치하도록 op-필드 엔코더(503)를 제어한다. 다음에, 단계(621)에서, 할당기(501)는 첫 번째 명령어(r1)에 의해 정의된 자원, 이 경우에는 r1에 대응하는 타임 스탬프 식별자를 도14a에 도시된 바와 같이, earliest_LDI 플래그(즉, earliest_LDI +1)의 증분(increment)(이 경우에서는 1)으로 갱신한다. 단계(623)에서, 상기 첫 번째 명령어가 정지 명령어가 아니기 때문에, 그룹 포맷터(123)의 연산은 상기 시퀀스의 두 번째 명령어로 계속된다.

두 번째 명령어에 있어서, 이 명령어에 의해 사용되는 자원, r1과 r5가 1과 0이라는 대응하는 타임 스탬프 식별자를 각각 갖고 있기 때문에, 할당기는 단계(603)에서 TS_{use}(MAX)를 1로서 정의한다. 또한, 이 명령어에 의해 정의된 자원 r4가 0이라는 대응하는 타임 스탬프 식별자를 갖고 있기 때문에, 할당기는 단계(607)에서 TS_{def} (MAX)를 0으로서 정의한다. 단계(609)에서, 할당기(501)는 earliest_LDI 플래그를 TS_{use} (MAX), 즉, 1로 셋팅하고, 흐름은 단계(613)로 계속된다. 단계(613)에서, 이제 1로 셋팅된 earliest_LDI 플래그가 3보다 크지 않기 때문에, 단계(615)에서 할당기(501)는 슬롯 FU0...FU3이 earliest_LDI 플래그에 대응하는 LDI(이 경우에는 LDI_1)에서 활용가능한지를 검사한다. 슬롯 FU0는 LDI_1에서 활용가능하기 때문에, 연산은 단계(619)로 계속되며, 여기서, 할당기(501)는 도14b에 도시된 바와 같이, LDI_1의 FU0 슬롯에 두 번째 명령어에 대응하는 op-필드를 배치하도록 op-필드 엔코더(503)를 제어한다. 다음에, 단계(621)에서, 할당기(501)는 두 번째 명령어(r4)에 의해 정의된 자원, 이 경우에는

r4에 대응하는 타임 스탬프 식별자를 도14a에 도시된 바와 같이, earliest_LDI 플래그(즉, earliest_LDI +1)의 증분(이 경우에서는 2)으로 갱신한다. 단계(623)에서는, 상기 두 번째 명령어가 정지 명령어가 아니기 때문에, 그룹 포맷터(123)의 연산은 이 시퀀스의 세 번째 명령어로 계속된다.

세 번째 명령어에 있어서는, 이 명령어에 의해 사용된 자원, r1과 r4이 1과 2라는 대응하는 타임 스탬프 식별자를 각각 갖고 있기 때문에, 할당기는 단계(603)에서 TS_{use}(MAX)를 2로서 정의한다. 또한, 명령어에 의해 정의된 자원, r3가 0이라는 대응하는 타임 스탬프 식별자를 갖고 있기 때문에, 할당기는 단계(607)에서 TS_{def}(MAX)를 0으로서 정으한다. 단계(609)에서, 할당기(501)는 earliest_LDI 플래그를 TS_{use}(MAX)로 셋팅하는데, 즉, 2로 세팅하고, 연산 흐름은 단계(613)로 계속된다. 단계(613)에서, 지금 2로 세팅된 earliest_LDI 플래그가 3보다 크지 않기 때문에, 단계(615)에서 할당기(501)는 슬롯 FU0...FU3이 earliest_LDI 플래그에 대응하는 LDI(여기서는 LDI_2)에서 활용가능한지를 검사한다. 슬롯 FU0은 LDI_2에서 활용가능하기 때문에, 연산은 단계(619)로 계속되고, 여기서, 할당기(501)는 도14b에 도시된 바와 같이, LDI_2의 FU0 슬롯에 세 번째 명령어에 대응하는 op-필드를 배치하도록 op-필드 엔코더(503)를 제어한다. 다음에, 단계(621)에서, 할당기(501)는 세 번째 명령어에 의해 정의된 자원, 이 경우에는 r3에 대응하는 타임 스탬프 식별자를 earliest_LDI 플래그(즉, earliest_LDI +1)의 증분으로 갱신하는데, 이 경우에 이 증분은 도14a에 도시된 바와 같이 3이된다. 단계(623)에서, 상기 세 번째 명령어는 정지 명령어가 아니기 때문에, 그룹 포맷터(123)의 연산은 이 시퀀스의 네 번째 명령어로 계속된다.

네 번째 명려어에 대해서는, 이 명령어에 의해 사용되는 자원, r2와 r5가 0이라는 대응하는 타임 스탬프 식별자를 갖고 있기 때문에, 할당기는 단계(603)에서 TS_{use}(MAX)를 0으로서 정의한다. 또한, 명령어에 의해 정의된 자원 r4가 2라는 대응하는 타임 스탬프 식별자를 갖고 있기 때문에, 할당기는 단계(607)에서 TS_{def}(MAX)를 2로서 정의한다. 단계(609)에서, 할당기(501)는 earliest_LDI 플래그를 2로 셋팅하고, 연산 흐름은 단계(613)로 계속된다. 단계(613)에서, 이제 2로 셋팅된 earliest_LDI 플래그가 3보다 크지 않기 때문에, 단계(615)에서 할당기(501)는 슬롯 FU0...FU3이 earliest_LDI 플래그에 대응하는 LDI(이 경우에는 LDI_2)에서 활용가능한지를 검사한다. 슬롯 FU1은 LDI_2에서 활용가능하기 때문에, 연산은 단계(619)로 계속되며, 여기서, 할당기(501)는 도14b에 도시된 바와 같이, LDI_2의 FU1 슬롯에 네 번째 명령어에 대응하는 op-필드를 배치하도록 op-필드 엔코더(503)를 제어한다. 다음에, 단계(621)에서, 할당기(501)는 네 번째 명령어에 의해 정의된 자원, 이 경우에는 r4에 대응하는 타임 스탬프 식별자를 도14a에 도시된 바와 같이, earliest_LDI 플래그(즉, earliest_LDI +1)의 증분(이 경우에서는 3)으로 갱신한다.
단계(623)에서, 상기 네 번째 명령어가 정지 명령어가 아니기 때문에, 그룹 포맷터(123)의 연산은 이 시퀀스의 다섯 번째 명령어로 계속된다.

다섯 번째 명령어에 대해서는, 이 명령어에 의해 사용되는 자원, r4가 3이라는 대응하는 타임 스탬프 식별자를 갖고 있기 때문에, 할당기는 단계(603)에서 TS_{use}(MAX)를 3으로 정의한다. 또한, 명령어에 의해 정의된 자원 r4가 3이라는 대응하는 타임 스탬프 식별자를 갖고 있기 때문에, 할당기는 단계(607)에서 TS_{def}(MAX)를 3으로서 정의한다. 이 경우에, 할당기(501)는 earliest_LDI 플래그를 3으로 셋팅하고, 연산 흐름은 단계(613)로 계속된다. 단계(613)에서, 이제 3으로 셋팅된 earliest_LDI 플래그가 3보다 크지 않기 때문에, 단계(615)에서 할당기(501)는 슬롯 FU0...FU이 earliest_LDI 플래그에 대응하는 LDI(이 경우에는 LDI_3)에서 활용가능한지를 체크한다. 슬롯 FU0가 LDI_3에서 활용가능하기 때문에, 연산은 단계(619)로 계속되고, 여기서, 할당기(501)는 도14b에 도시된 바와 같이, LDI_3의 FU0 슬롯에 다섯 번째 명령어에 대응하는 op-필드를 배치하도록 op-필드 엔코더(503)를 제어한다. 다음에, 단계(621)에서, 할당기(501)는 다섯 번째 명령어에 의해 정의된 자원, 이 경우에는 r4에 대응하는 타임 스탬프 식별자를 도14a에 도시된 바와 같이, earliest_LDI 플래그(즉, earliest_LDI+1)의 증분(이 경우에서는 4)으로 갱신한다. 단계(623)에서, 상기 다섯 번째 명령어가 정지 명령어가 아니기 때문에, 그룹 포맷터(123)의 연산은 이 시퀀스의 여섯 번째 명령어로 계속된다.

여섯 번째 명령어에 대해서는, 이 명령어에 의해 사용되는 자원, r3과 r4가 3과 4라는 대응하는 타임 스탬프 식별자를 갖고 있기 때문에, 할당기는 단계(603)에서 TS_{use}(MAX)를 4로서 정의한다. 또한, 명령어에 의해 정의된 자원 r2가 0이라는 대응하는 타임 스탬프 식별자를 갖고 있기 때문에, 할당기는 단계(607)에서 TS_{def}(MAX)를 0으로서 정의한다. 단계(609)에서, 할당기(501)는 earliest_LDI 플래그를 TS_{use}(MAX) 즉, 4로 셋팅하고, 연산 흐름은 단계(613)로 계속된다. 단계(613)에서, 이제 4으로 셋팅된 earliest_LDI 플래그가 3보다 크기 때문에, 할당기(501)는 도13의 단계들로 분기하여, 이 그룹을 폐쇄하고, 그룹 버퍼(507)에 저장된 폐쇄된 그룹을 병렬 엔진(125)의 DIB에 기록한다. 다음에, 할당기(501)는 첫 번째 명령어로부터 다섯 번째 명령어에 대해 전술한 바와 같이, 여섯 번째 명령어 및 그 후속 명령어들을 새로운 그룹으로 포맷팅하는 동작을 계속한다.

단계(609)에서 자원 재정의 조건(resource redefinition condition)을 만나면, 비록 특정 명령어에 의해 사용되는 자원이 상기 그룹의 earlier-LDI에서 활용가능할 수도 있지만, 전술한 바와 같이 할당기(501)는 공통 자원을 정의하는 그룹의 하나 또는 그 이상의 이전 명령어에 이어지는 LDI에 상기 명령어를 기록한다. 이렇게 하는 것은 자원의 가장 효율적인 이용은 아니다. 다른 실시예에서, 그룹 포맷터(123)는 본 발명의 양수인에게 공히 양도된, J.Cocke 등의 미국 특허 제 4,992,938호에 설명된 종래의 레지스터 재명명 기술을 이용할 수 있으며, 이 특허는 본 명세서에 참조로써 병합되어 있다. 레지스터 재명명 기술은 병렬 엔진(125)의 자원의 보다 효율적인 이용을 제공한다.

특히, 도15도에 도시된 바와 같이, 도12 및 도13과 관련하여 전술한 바와 같이, 할당기(501)의 연산은 레지스터 재명명 기술을 이용하기 위해 수정될 수 있다. 이러한 경우에 있어서는, 단계(700)에서, 할당기(501)는 특정 명령어가 조건부 분기 명령어인지를 검사함으로써 시작된다. 만일 그렇다면,
단계(702)에서, 할당기(501)는 상기 명령어에 대응하는 op-필드를 그룹 버퍼(507)에서 current_max 플래그에 대응하는 LDI의 활용가능한 슬롯으로 기록하도록 op-필드 엔코더(503)를 제어하고, 상기 명령어에 의해 정의된 각각의 자원에 대해, 그 자원에 관련된 타임 스탬프 식별자를 {current_max 플래그+1}로 갱

신하고, 그 갱신된 타임 스탬프 식별자를 TST(505)에 기록한다. 전술한 바와 같이, current_max 플래그는 상기 조건부 분기 명령어가 순서적으로 유지될 수 있도록 현재 최대 LDI를 추적한다. 단계(702)이후에, 연산은 단계(700)로 되돌아가서, 현재 그룹으로의 통합을 위해 그 시퀀스의 다음 명령어를 처리하기 시작한다.

만일 단계(700)에서, 할당기(501)가 상기 명령어가 조건부 분기 명령어가 아니라고 판단하면, 연산은 단계(701)로 계속된다. 단계(701)에서는, 특정 명령어에 의해 사용되는 각각의 자원에 대해, 할당기(501)는 그 자원에 관련된 재명명 플래그를 검사하고(상세하게 후술됨), 그에 따라 자원을 재명명한다. 다음에는, 단계(704)에서, 특정 명령어에 의해 사용되는 각각의 자원에 대해, 할당기(501)는 그 자원이 활용가능하게 될 LDI를 식별하는, 상기 자원에 대응하는 타임 스탬프 식별자를 TST(505)로부터 검색한다. 단계(706)에서, 할당기(501)는 단계(704)에서 검색된 타임 스탬프 식별자 중 최대의 것을 판단하게 되는데, 이것은 설명의 편의상 TS_{use}(MAX)로 레이블링된다.

단계(708)에서, 특정 명령어에 의해 정의된 각각의 자원에 대해, 할당기(501)는 관련된 재명명 플래그가 셋팅되었는지를 검사한다. 만약 재명명 플래그가 셋팅되어 있지 않으면, 단계(712)에서 할당기(501)는 재명명 플래그를 세팅하여, 특정 자원이 이미 이전에 정의되었다는 것을 상기 그룹의 후속 명령어들에 표시하고, 또한, 이 새로운 자원 명칭을 그 자원의 원래의 명칭으로서 초기화한 다음에, 연산은 단계(718)로 계속된다.

그러나 , 만약 단게(708)에서, 특정 명령어에 의해 정의된 각각의 자원에 대해, 할당기(501)가 재명명 플래그가 셋팅되었다고 판단하면, 연산은 단계(710) 내지 단계(716)로 계속된다. 단계(710)에서, 할당기(501)는 자유로운 자원(free resource)을 로케이트하고, 단계(714)에서 특정 자원을 자유로운 자원으로 재명명한다. 단계(716)에서, 할당기(501)는 특정 자원이 재명명되었다는 것을 나타내는 재명명 플래그를 셋팅하고, 이 지원의 새로운 명칭을 자유 자원 명칭으로 셋팅한 다음, 연산은 단계(718)로 계속된다.

단계(718)에서, 할당기(501)는 earliest_LDI 플래그를 단계(706)에서 정의된 바와 같이, TS_{use}(MAX)로 셋팅하고, 연산은 도12에 관해 전술한 바와 같이, 단계(613)로 계속되어, 상기 명령어를 나타내는 op-필드를 그룹 버퍼(507)의 적절한 LDI에 배치하게 된다.

전술한 바와 같은 자원 재명명 기술을 이용하는 그룹 포맷터(123)의 동작을 설명하기 위해, 전술한 6개의 명령어의 동일한 시퀀스를 생각하자. 도16a 및 도16b는 그룹 포맷터(123)가 도15에 관하여 전술한 바와 같이, 상기 명령어 시퀀스를 그룹으로 포맷팅하기 위해 동작할 때의 TST(505) 및 그룹 버퍼(507)의 상태를 예시하고 있다.

네 번째 명령어가 재명명을 초래하는 시퀀스 상의 첫 번째 명령어이기 때문에, 첫 번째 명령어부터 세 번째 명령어를 처리하는데 있어서의 그룹 포맷터(123)의 연산 결과는 변경되지 않는다. 그러나, 네 번째 명령어에 대해서는, 단계(708)에서, r4와 관련된 재명명 플래그가 셋팅되었으며, 연산은 자유 자원, 예를 들어, 도16a에 도시된 바와 같이, 자원 rN을 로케이트하기 위해 단계(710)로 계속된다. 단계(712)에서, 할당기(501)는 재정의 조건을 유발한 특정 자원(이 경우에는 r4)을 자유 자원 rN으로 재명명하고, 단계(716)에서, 할당기(501)는 자원 r4가 명시적으로 rN으로 재명명되었다는 것을 나타내기 위해 특정 자원 r4가 관련된 재명명 플래그 및 자원 명칭을 셋팅한다. 단계(718)에서, 할당기(501)는 earliest_LDI 플래그를 TS_{use}(MAX), 여기서는 0으로 셋팅하고, 연산 흐름은 단계(613)로 계속된다. 단계(123)에서는, 이제 0 으로 셋팅된 earliest_LDI 플래그가 3보다 크지 않기 때문에, 단계(615)에서 할당기(501)는 슬롯 FU0...FU3이 earliest_LDI 플래그에 대응하는 LDI(이 경우에는 LDI_0)에서 활용가능한지를 검사한다. 슬롯 FU1이 LDI_0에서 활용가능하기 때문에, 연산은 단계(619)로 계속되며, 여기서, 할당기(501)는 도16b에 도시된 바와 같이 LDI_0의 FU1 슬롯에 네 번째 명령어에 대응하는 op-필드를 배치하도록 op-필드 엔코더(503)를 제어한다. 그리고 나서, 단계(621)에서, 할당기(501)는 네 번째 명령어에 의해 정의된 자원(이 경우에는 rN)에 대응하는 타임 스탬프 식별자를 도 16a에 도시된 바와 같이, 이 경우에는 1이되는 earliest_LDI 플래그(즉, earliest_LDI +1)의 증분으로 갱신한다. 단계(623)에서, 네 번째 명령어가 정지 명령어가 아니기 때문에, 그룹 포맷터(123)의 연산은 시퀀스의 다섯 번째 명령어로 계속된다.

다섯 번째 명령어에 대해서는, 이 명령어에 의해 사용되는 자원과 관련된 재명명 플래그가 자원 r4가 재명명되었다는 것을 나타내기 때문에, 할당기(501)는 도16a에 도시된 바와 같이, 단계(701)에서 자원 r4를 rN으로 재명명한다. 또한, 이 명령어에 의해 사용되는 자원, rN이 1이라는 대응하는 타임 스탬프 식별자를 갖고 있기 때문에, 할당기는 단계(706)에서 TS_{use}(MAX)를 1로서 정의한다. 단계(708)에서는, 상기 명령어에 의해 정의된 자원, 이 경우에는 r4와 관련된 재명명 플래그가 셋팅되었기 때문에, 연산은 단계(710)로 계속되어, 자유 자원, 예를 들어, 도16a에 도시된 바와 같이, 자원 rN을 로케이트한다. 제16(a)도에서 보이는 바와 같이 자원 rN로 예를 들었다. 단계(712)에서, 할당기(501)는 재정의 조건을 유발한 특정 자원, 이 경우에는 r4를 자유 자원 rN으로 재명명하고, 단계(716)에서, 할당기(501)는 자원 r4가 명시적으로 rN으로 재명명되었다는 것을 나타내기 위해 특정 자원 r4와 관련된 재명명 플래그 및 자원 명칭을 셋팅한다. 단계(718)에서, 할당기(501)는 earliest_LDI 플래그를 TS_{use}(MAX), 즉, 이 경우에는 1로 셋팅하고, 연산은 단계(613)로 계속된다. 단계(613)에서, 이제 1로 셋팅된 earliest_LDI 플래그가 3보다 크지 않기 때문에, 단계(615)에서 할당기(501)는 슬롯 FU0...FU3이 earliest_LDI 플래그에 대응하는 LDI(여기서는 LDI_1)에서 활용가능한지를 검사한다. 슬롯 FU1이 LDI_1에서 활용가능하기 때문에, 연산은 단계(619)로 계속되며, 여기서, 할당기(501)는 도14b에 도시한 바와 같이, LDI_1의 FU1 슬롯에 다섯 번째 명령어에 대응하는 op-필드를 배치하도록 op-필드 엔코더(503)를 제어한다. 그리고 나서, 단계(621)에서, 할당기(501)는 다섯 번째 명령어에 의해 정의된 자원, 이 경우에는 rN에 대응하는 타임 스탬프 식별자를 도16a에 도시한 바와 같이, 이 경우에는 2가 되는, earliest_LDI 플래그(즉, earliest_LDI+1)의 증분으로 갱신한다. 단계(623)에서는, 다섯 번째 명령어가 정지 명령어가 아니기 때문에, 그룹 포맷터(123)의 연산은 시퀀스의 여섯 번째 명령어로 계속된다.

여섯 번째 명령어에 대해서는, 이 명령어에 의해 사용되는 자원, 즉, 이 경우에는 r4와 관련된 재명명

플래그가 그 자원 r4가 재명명되었다는 것을 나타내기 때문에, 할당기(501)는 단계(701)에서, 도16a에 도시된 바와 같이, 자원 r4를 rN으로 재명명한다. 또한, 이 명령어에 의해 사용되는 자원, r3와 rN이 각각 3과 2라는 대응하는 타임 스탬프 식별자를 갖고 있기 때문에, 할당기는 단계(603)에서 TS_{use}(MAX)를 3으로서 정의한다. 단계(708)에서, 할당기(501)는 상기 명령어에 의해 정의된 자원, 즉, 이 경우에는 r2와 관련된 재명명 플래그가 리셋되어 있지 않다는 것을 판단하고, 연산은 단계(712)로 계속되어, r2와 관련된 자원 명칭 플래그를 그 원래의 명칭으로 셋팅한다. 단계(718)에서, 할당기(501)는 earliest_LDI 플래그를 TS_{use}(MAX), 즉, 이 경우에는 3으로 셋팅하고, 연산은 단계(613)로 계속된다. 이 단계(613)에서는, 이제 3으로 셋팅된 earliest_LDI 플래그가 3보다 크지 않기 때문에, 단계(615)에서 할당기(501)는 슬롯 FU0...FU3이 earliest_LDI 플래그에 대응하는 LDI(여기서는 LDI_3)에서 활용가능한지를 검사한다. 슬롯 FU1이 LDI_3에서 사용 가능하기 때문에, 연산은 단계(619)로 계속되며, 여기서, 할당기(501)는 도14b에 도시한 바와 같이, LDI_3의 FU0 슬롯에 여섯 번째 명령어에 대응하는 op-필드를 배치하도록 op-필드 엔코더(503)를 제어한다. 그리고 나서, 단계(621)에서, 할당기(501)는 여섯 번째 명령어에 의해 정의된 자원, 즉, 이 경우에는 r3에 대응하는 타임 스탬프 식별자를 도16a에 도시한 바와 같이, earliest_LDI 플래그(즉, earliest_LDI+1)의 증분, 즉, 이 경우에는 4로 갱신한다. 단계(623)에서, 여섯 번째 명령어가 정지 명령어가 아니기 때문에, 첫 번째 명령어부터 여섯 번째 명령어까지에 관해 전술한 바와 같이, 그룹 포맷터(123)의 연산은 시퀀스의 그 다음 후속 명령어(도시 안됨)로 계속된다.

전술한 처리 단계뿐만 아니라, 단계(619)에서, 할당기(501)는 특정 op-필드에 의해 표현되는 기능이 실행되어질(또는 실행되지 않을) 조건을 나타내는 LDI의 op-필드와 관련된 플래그를 셋팅할 수도 있다. 이러한 경우에, 병렬 엔진(125)의 실행 유니트들은 관련된 플래그에 의해 표현된 조건이 만족된 경우에만, 대응하는 op-필드에 의해 표현되는 기능을 실행하도록 제어된다.

전술한 예에서, 시퀀셜 머신의 실행 유니트들로 실행을 위해 디스패치된 각각의 명령어에 대해, op-필드 엔코더(503)는 병렬 엔진(125)의 기능 유니트 및 분기 처리 유니트에 의해 단일 머신 사이클에서 실행될 수 있는 명령어에 대응하는 단일 op-필드를 생성한다. 그러나, 본 발명은 이러한 관점에 제한되지 않는다. 다른 실시예에 있어서는, 명령어에 대응하는 op-필드가 병렬 엔진(125)의 기능 유니트 및 분기 처리 유니트 상에서의 실행을 위해 하나 이상의 머신 사이클을 이용할 수도 있다. 단계(621)에서의 할당기(501)의 연산은 적절한 수의 대기시간 단위로 명령어에 의해 정의된 자원에 대응하는 타임 스탬프 식별자를 갱신하도록 수정된다. 예를 들어, 만일 op-필드가 실행을 위해 3개의 머신 사이클을 이용하면, 상기 명령어에 의해 정의된 자원에 대응하는 타임 스탬프 식별자는 3으로 갱신된다. 또한, 다른 실시예에서는, op-필드 엔코더(503)가 시퀀셜 머신의 실행 유니트에 의한 실행을 위해 디스패치된 단일 명령어에 대응하는 다수의 op-필드를 생성할 수도 있다. 이 경우에, 할당기(501)의 연산은 상기 명령어에 대응하는 다수의 op-필드가 단계(619)에서 적절한 LDI이 배치되도록 수정된다. 또한, 다른 실시예에서는, 병렬 엔진(125)의 하나 또는 그 이상의 기능 유니트가 단일 머신 사이클에서 하나 이상의 원래의 명령어들을 실행할 수도 있으며, 이에 대해서는, 예를 들어, Malik 등에 의한 "Interlock Collapsing ALU for Increased Instruction-Level Parallelism", Proceedings of Micro-25, 포틀랜드, 오레곤, 1992년 12월, pp. 149-157에 기술되어 있으며, 이 참조문헌은 그 전부가 여기에 참조로서 병합되어 있다.이 경우에, op-필드 엔코더(503)는 하나 이상의 원래의 명령어에 대응하는 단일 op-필드를 생성하고, 할당기(501)의 연산은 하나 이상의 원래의 명령어에 대응하는 단일 op-필드가 단계(619)에서 적절한 LDI에 배치되도록 수정된다.

본 발명의 제2 실시예에 따르면, 제1 실시예에 대하여 전술한 바와 같은 할당기(501)의 연산은 도17에 도시된 바와 같은 추가적인 처리 단계를 포함할 수 있다. 보다 특정하여 말하면, 시퀀셜 머신의 실행 유니트에 의한 실행을 위해 디스패치된 각각의 명령어에 의해, 할당기(501)의 연산은 그 할당기(501)가 특정 명령어가 정지 명령어인지를 검사하는 단계(801)를 포함할 수 있다. 여기서, 정지 명령어는 시퀀셜 머신 아키텍쳐의 명령어 셋트의 사전-정의된 서브셋트가 될 수도 있다. 이러한 사전-정의된 서브셋트는 예를 들어, 다중 레지스터 로드 명령어나 정수 분할 명령어와 같이, 높은 대기시간을 갖고, 따라서, 현재 그룹으로 포맷팅하기에는 비효율적이거나, 또는 병렬 엔진(125)에 의해 실행되지 못할 수도 있는 명령어들을 포함하는 것이 바람직하다. 만약 단계(801)에서, 할당기(501)가 상기 명령어가 정지 명려어라고 판단하면, 할당기(501)의 연산은 전술한 바와 같이 도13에 예시된 단계들로 계속되어, 상기 그룹을 폐쇄하고, 그룹 버퍼(507)로부터의 폐쇄된 그룹을 병렬 엔진(125)의 DIB에 기록한다.

또한, 그룹 포맷터(123)는 데이터가 손실되거나 변조되지 않도록, 로드 연산과 저장 연산의 순서를 유지할 수 있다. 이것은 TST(505)의 하나의 셀(ML로 명명됨)을 메모리 로드 연산에 할당하고, TST(505)의 하나의 셀(MS로 명명됨)을 메모리 저장 연산에 할당함으로써 이루어질 수 있다. 특히, 상기 셀 ML은 그룹의 최근의 메모리 로드 연산이 완료되게 될 현재 그룹의 LDI를 식별하는 타임 스탬프 식별자를 저장하고, 상기 셀 MS는 그룹의 최근의 메모리 저장 연산이 완료되게 될 현재 그룹의 LDI를 식별하는 타임 스탬프 식별자를 저장한다. 이 경우에, 그룹 포맷터(123)의 할당기(501)는 도17에 도시된 바와 같이, 로드 연산과 저장 연산의 순서를 유지하기 위해 이들 셀 ML 및 MS에 저장된 타임 스탬프 식별자를 이용한다. 단계(803)에서, 할당기(501)는 상기 명령어가 로드 연산을 포함하고 있는지를 검사한다. 만약 그렇다면, 연산은 단계(805)로 계속되며, 여기서, 할당기(501)는 특정 명령어에 의해 정의된 자원에 대응하는 TST(505)의 셀에 TST(505)의 셀 ML을 관련시키고, 또한 특정 명령어에 의해 사용되는 자원에 대응하는 TST(505)의 셀에 셀 MS를 관련시킨다. 단계(805) 이후에, 할당기(501)의 연산은 전술한 바와 같이, 단계(601)로 계속되어, 명령어를 처리하고, 그것을 현재 그룹의 적절한 LDI(활용가능한 경우)에 배치한다.

그러나, 만일 단계(803)에서 할당기(501)가 상기 명령어가 로드 연산을 포함하고 있지 않다고 판단하면, 연산은 단계(807)로 계속되어, 상기 명령어가 저장 연산을 포함하는지를 검사한다. 만일 그렇다면, 동작은 단계(809)로 계속되며, 여기서, 할당기(501)는 특정 명령어에 의해 정의된 자원에 대응하는 TST(505)의 셀에 TST(505)의 셀 MS를 관련시키고, 특정 명령어에 의해 사용되는 자원에 대응하는 TST(505)에 셀에 셀 MS 및 ML를 또한 관련시킨다. 단계(809) 이후에, 할당기(501)의 연산은 전술한 바와 같이, 단계(601)로 계속되어, 명령어를 처리하고, 그것을 현재 그룹의 적절한 LDI(활용가능한 경우)에 배치한다.

끝으로, 만일 단계(807)에서 할당기(501)가 상기 명령어가 저장 연산을 포함하고 있지 않다고 판단하면, 연산은 전술한 바와 같이 단계(601)로 계속되어, 상기 명령어를 처리하고, 그것을 현재 그룹의 적당한

LDI(활용가능한 경우)에 배치한다.

메모리 로드 연산 및 저장 연산을 포함하는 명령어들의 순서를 유지하는데 있어서의 할당기(501)의 연산을 설명하기 위해, 다음의 명령어 시퀀스를 고려하자 .

1. Store r7 -> u(r10)

2. Add r,3r4 -> r2

3. Load 5(r2) -> r3

4. Load 7(r6) -> r9

5. Store r9 -> 8(r2)

도18a 및 도18b는 예를 들어, 도17, 도12 및 도13과 관련하여 전술한 바와 같이, 그룹 포맷터(123)가 이 명령어 시퀀스를 그룹으로 포맷팅할 때, TST(505) 및 그룹 버퍼(507)의 상태를 예시하고 있다.

이 그룹의 시작부분에서, 첫 번째 명령어를 처리하기 전에, TST(505)는 각각의 셀에 0을 갖도록 초기화된다. 첫 번째 명령어에 대해서는, 할당기(501)는 단계(807)에서, 상기 명령어가 저장 연산을 포함하고 있는지를 판단하고, 연산은 단계(808)로 계속되며, 여기서, 할당기(501)는 첫 번째 명령어에 의해 정의된 자원에 대응하는 TST(505)의 셀에 TST(505)의 셀 MS를 관련시키고, 첫 번째 명령어에 의해 사용되는 자원에 대응하는 TST(505)의 셀에 셀 MS와 ML를 또한 관련시킨다. 그리고 나서, 첫 번째 명령어에 의해 사용되는 자원, r7, r10, MS, ML에 대응하는 TST(505)의 셀들이 0이라는 타임 스탬프 식별자를 갖고 있기 때문에, 할당기는 단계(603)에서 TS_{use}(MAX)를 0으로 정의한다. 또한, 명령어에 의해 정의된 자원 MS에 대응하는 TST(505)의 셀이 0이라는 타임 스탬프 식별자를 갖고 있기 때문에, 할당기는 단계(607)에서 TS_{def}(MAX)를 0으로서 정의한다. 단계(609)에서, 할당기(501)는 earliest_LDI 플래그를 0으로 셋팅하고, 흐름은 단계(613)로 계속된다. 단계(613)에서, 이제 0으로 세팅된 earliest_LDI 플래그가 3보다 크지 않기 때문에, 단계(615)에서 할당기(501)는 earliest_LDI 플래그에 대응하는 LDI(이 경우에는 LDI_0)에서 슬롯 FU0...FU3이 활용가능한지를 검사한다. 슬롯 FU0가 LDI_0에서 활용가능하기 때문에, 연산은 단계(619)로 계속되며, 여기서, 할당기(501)는 도18b에 도시된 바와 같이, LDI_0의 FU0 슬롯에 첫 번째 명령어에 대응하는 op-필드를 배치하도록 op-필드 엔코더(503)를 제어한다. 그리고 나서, 단계(621)에서, 할당기(501)는 도18a 에 도시된 바와 같이, 첫 번째 명령어에 의해 정의된 자원, 즉, 이 경우에는 MS에 대응하는 셀의 타임 스탬프 식별자를 earliest_LDI 플래그(즉, earliest_LDI +1)의 증분, 즉, 이 경우에는 1로 갱신한다. 단계(623)에서, 상기 첫 번째 명령어가 정지 명령어가 아니기 때문에, 그룹 포맷터(123)의 연산은 상기 시퀀스의 두 번째 명령어로 계속된다.

두 번째 명령어에 대해서는, 이 명령어가 로드 또는 저장 연산을 포함하고 있지 않기 때문에, 단계(805,808)는 바이패스되고, 연산은 단계(601)로 계속된다. 상기명령어에 의해 사용되는 자원, r3과 r4는 0이라는 대응하는 타임 스탬프 식별자를 각각 갖고 있기 때문에, 할당기는 단계(603)에서 TS_{use}(MAX)를 0으로서 정의한다. 또한, 이 명령어에 의해 정의된 자원 r2는 0이라는 대응하는 타임 스탬프 식별자를 갖고 있기 때문에, 할당기는 단계(607)에서 TS_{def}(MAX)를 0으로서 정의한다. 단계(609)에서, 할당기(501)는 earliest_LDI 플래그를 0으로 셋팅하고, 연산 흐름은 단계(613)로 계속된다. 단계(613)에서, 이젠 0으로 셋팅된 earliest_LDI 플래그가 3보다 크지 않기 때문에, 단계(615)에서 할당기(501)는 earliest_LDI 플래그에 대응하는 LDI(이 경우에는 LDI_0)에서 슬롯 FU0...FU3이 활용가능한지를 검사한다. 슬롯 FU1이 LDI_0에서 활용가능하기 때문에, 연산은 단계(619)로 계속되며, 여기서, 할당기(501)는 도18b에 도시된 바와 같이, LDI_0 의 FU1 슬롯에 두 번째 명령어에 대응하는 op-필드를 배치하도록 op-필드 엔코더(503)를 제어한다. 그리고 나서, 단계(621)에서, 할당기(501)는 두 번째 명령어에 의해 정의된 자원, 즉, 이 경우에는 r2에 대응하는 셀의 타임 스탬프 식별자를 도18a에 도시된 바와 같이, earliest_LDI 플래그(즉, earliest_LDI +1)의 증분, 즉, 이 경우에는 1로 갱신한다. 단계(623)에서, 상기 두 번째 명령어는 정지 명령어가 아니기 때문에, 그룹 포맷터(123)의 연산은 이 시퀀스의 세 번째 명령어로 계속된다.

세 번째 명령어에 대해서는, 단계(803)에서, 할당기(501)는 이 명령어가 로드 연산을 포함하는지를 판단하고, 연산은 단계(805)로 계속되며, 여기서, 할당기(501)는 세 번째 명령어에 의해 정의된 자원에 대응하는 TST(505)의 셀에 TST(505)의 셀 ML을 관련시키고, 또한, 세 번째 명령어에 의해 사용되는 자원에 대응하는 TST(505)의 셀에 셀 MS를 관련시킨다. 다음에는, 세 번째 명렁어에 의해 사용되는 자원 r2 및 MS에 대응하는 TST(505)의 셀이 1이라는 타임 스탬프 식별자를 갖고 있기 때문에, 할당기(501)는 단계(603)에서 TS_{use}(MAX)를 1로서 정의한다. 또한, 이 명령어에 의해 정의된 자원, r3과 ML에 대응하는 TST(505)의 셀이 0이라는 타임 스탬프 식별자를 갖고 있기 때문에, 할당기(501)는 단계(607)에서 TS_{def}(MAX)를 0으로서 정의한다. 단계(609)에서, 할당기(501)는 earliest_LDI 플래그를 TS_{use}(MAX), 즉, 1로 세팅하고, 연산 흐름은 단계(613)로 계속된다. 단계(613)에서, 이제 0으로 셋팅된 earliest_LDI 플래그가 3보다 크지 않기 때문에, 단계(615)에서 할당기(501)는 earliest_LDI 플래그에 대응하는 LDI(이 경우에는 LDI_1)에서 슬롯 FU0...FU3이 활용가능한지를 검사한다. 슬롯 FU0은 LDI_1에서 활용가능하기 때문에, 연산은 단계(619)로 계속되며, 여기서, 할당기(501)는 도18b에 도시된 바와 같이, LDI_1의 FU0 슬롯에 세 번째 명령어에 대응하는 op-필드를 배치하도록 op-필드 엔코더(503)를 제어한다. 다음에, 단계(621)에서, 할당기(501)는 세 번째 명령어에 의해 정의된 자원, 즉, 이 경우에는 r3과 ML에 대응하는 셀의 타임 스탬프 식별자를 도18a에 도시된 바와 같이, earliest_LDI 플래그(즉, earliest_LDI +1)의 증분, 즉 이 경우에는 2로 갱신한다. 단계(623)에서, 세 번째 명령어는 정지 명령어가 아니기 때문에, 그룹 포맷터(123)의 연산은 이 시퀀스의 네 번째 명령어로 계속된다.

네 번째 명령어에 대해서는, 단계(803)에서, 할당기(501)는 상기 명령어가 로등 연산을 포함하는지를 판단하고, 연산은 단계(805)로 계속되며, 여기서, 할당기(501)는 네 번째 명령어에 의해 정의된 자원에 대

응하는 TST(505)의 셀에 TST(505)의 셀 ML을 관련시키고, 또한 네 번째 명령어에 의해 사용되는 자원에 대응하는 TST(505)의 셀에 셀 MS를 관련시킨다. 다음에는, 상기 명령어에 의해 사용되는 자원, r6 및 MS에 대응하는 TST(505)의 셀이 각각 0과 1이라는 타임 스탬프 식별자를 갖고 있기 때문에, 할당기는 단계(603)에서 TS_{use}(MAX)를 1로서 정의한다. 또한, 상기 명령어에 의해 정의된 자원 r9와 ML에 대응하는 TST(505)의 셀이 각각 0과 2라는 타임 스탬프 식별자를 갖고 있기 때문에, 할당기는 단계(607)에서 TS_{def}(MAX)를 2로서 정의한다. 단계(609)에서, 할당기(501)는 earliest_LDI 플래그를 TS_{use}(MAX), 즉 2로 셋팅하고, 연산 흐름은 단계(613)로 계속된다. 단계(613)에서, 이제 2로 셋팅된 earliest_LDI 플래그가 3보다 크지 않기 때문에, 단계(615)에서 할당기(501)는 earliest_LDI 플래그에 대응하는 LDI(이 경우에는 LDI_2)에서 슬롯 FU0...FU3이 활용가능한지를 검사한다. 슬롯 FU0이 LDI_2에서 활용가능하기 때문에, 연산은 단계(619)로 계속되며, 여기서, 할당기(501)는 도18b에 도시된 바와 같이, LDI_2의 FU0 슬롯에 네 번째 명령어에 대응하는 op-필드를 배치하도록 op-필드 엔코더(503)를 제어한다. 다음에는, 단계(621)에서, 할당기(501)는 네 번째 명령어에 의해 정의된 자원, 즉, 이 경우에는 r9와 ML에 대응하는 셀의 타임 스탬프 식별자를 도18a에 도시된 바와 같이, earliest_LDI 플래그(즉, earliest_LDI +1)의 증분, 즉 이 경우에는 3으로 갱신한다. 단계(623)에서, 상기 네 번째 명령어가 정지 명령어가 아니기 때문에, 그룹 포맷터(123)의 연산은 이 시퀀스의 다섯 번째 명령어로 계속된다.

다섯 번째 명령어에 대해, 단계(807)에서, 할당기(501)는 상기 명령어가 저장 연산을 포함하는지를 판단하고, 연산은 단계(809)로 계속되며, 여기서, 할당기(501)는 다섯 번째 명령어에 의해 정의된 자원에 대응하는 TST(505)의 셀에 TST(505)의 셀 MS을 관련시키고, 또한 이 다섯 번째 명령어에 의해 사용되는 자원에 대응하는 TST(505)의 셀에 셀 ML과 MS를 관련시킨다. 그리고 나서, 다섯 번째 명령어에 의해 사용되는 자원, r6, r9, ML, MS에 대응하는 TST(505)의 셀이 각각 1, 3, 3, 1이라는 타임 스탬프 식별자를 갖고 있기 때문에, 할당기는 단계(603)에서 TS_{use}(MAX)를 3으로서 정의한다. 또한, 상기 명령어에 의해 정의되는 자원 MS에 대응하는 TST(505)의 셀이 1이라는 타임 스탬프 식별자를 갖고 있기 때문에, 할당기(501)는 단계(607)에서 TS_{def} (MAX)를 1로서 정의한다. 단계(609)에서, 할당기(501)는 earliest_LDI 플래그를 TS_{use} (MAX), 즉 3로 셋팅하고, 연산 흐름은 단계(613)로 계속된다. 단계(613)에서, 이제 3로 셋팅된 earliest_LDI 플래그가 3보다 크지 않기 때문에, 단계(615)에서 할당기(501)는 earliest_LDI 플래그에 대응하는 LDI(이 경우에는 LDI_3)에서 슬롯 FU0...FU3이 활용가능한지를 검사한다. 슬롯 FU0이 LDI_3에서 활용가능하기 때문에, 연산은 단계(619)로 계속되며, 여기서, 할당기(501)는 도18b에 도시된 바와 같이, LDI_3의 FU0 슬롯에 다섯 번째 명령어에 대응하는 op-필드를 배치하도록 op-필드 엔코더(503)를 제어한다. 그리고 나서, 단계(621)에서, 할당기(501)는 네 번째 명령어에 의해 정의된 자원, 즉, 이 경우에는 MS에 대응하는 셀의 타임 스탬프 식별자를 도18a에 도시된 바와 같이, earliest_LDI 플래그(즉, earliest_LDI +1)의 증분, 즉 이 경우에는 4로 갱신한다. 단계(623)에서, 상기 다섯 번째 명령어가 정지 명령어가 아니기 때문에, 그룹 포맷터(123)의 연산은 이 시퀀스의 다음 명령어로 계속된다.

또한, 그룹 포맷터(123)가 로드 명령어의 순서를 반드시 유지할 필요는 없다. 예를 들어, 다른 로드 명령어보다 앞선 LDI에 로드 명령어를 배치하는 것이 더욱 효율적일 수도 있다. 이렇게 하기 위해, 전술한 바와 같은 할당기(501)의 연산은 로드 명령에 대해서는 단계(605~609)가 바이패스되도록 수정되며, 따라서, 할당기(501)의 연산은 로드 명령어에 대해서는 직접 단계(603)로부터 단계(611)로 계속된다. 이러한 경우에, 그룹 포맷터(123)가 이 명령어의 시퀀스를 그룹으로 포맷팅하도록 동작할 때의 TST(505) 및 그룹 버퍼(507)의 상태가 도 18c 및 도18d에 예시되어 있다. 또한, 로드 연산 및 저장 연산의 순서를 유지하기 위해 전술한 바와 같이 TST(505)의 ML 및 MS 셀을 이용하는 기술은 다른 연산, 예를 들면, 특수목적 레지스터 이동 연산(move special purpose register operation)으로 확장될 수 있다.

본 발명의 제3 실시예에 따르면, 그룹 포맷터(123)의 연산은 병렬 엔진(125)의 기능 유니트 및 분기 처리 유니트에 의해 생성된 그룹의 실행 결과에 연결되도록 수정될 수 있다. 보다 특정하여 말하면, CGB(204)에 저장된 현재 그룹 밖으로의 분기를 경험할 때, 그룹 포맷터(123)는 새로운 그룹을 편성하는 것을 시작하는 대신에, CGB(204)에 저장된 현재 그룹을 수정하도록 제어될 수 있다.

이러한 특징을 설명하기 위해, 도10a와 관련하여 전술한 바와 같이 그룹 포맷터(123)에 의해 포맷팅되어, 계속해서 병렬 엔진(125)의 기능 유니트 및 분기 처리 유니트에 의해 실행되는 그룹을 고려하자. 예를 들어, LDI_1이 실행될 때, 조건은 X=1을 가리키고, 그룹 밖으로의 분기가 라벨 LK로 발생한다고 가정하자. 이 시점에서, 새로운 그룹을 시작하는 것이 아니라, 도19에 도시된 바와 같이, 기존의 그룹이 그룹 포맷터(123)로 로드되어, 강화된다. 먼저, LDI_1에서 그룹 밖으로의 분기가 제거되고, 명령어 k는 가능한 가장 빠른 LDI, 이 경우에는 LDI_0의 FU3 슬롯에 배치된다. 다음 명령어 1이 명령어 k에 종속되기 때문에, 이 명령어 1은 활용가능한 슬롯, 이 경우에는 LDI_1의 FU2에 배치된다. 다음 명령어 m은 e와 k에 종속되기 때문에, 이 명령어 m은 활용가능한 슬롯, 이 경우에는 LDI_1의 FU3에 배치된다. 그러나, 다음 명령어 n은 어떤 명령어에도 종속되지 않으며, 따라서 LDI_0과 LDI_1의 모든 슬롯이 활용가능하며, 명령어 n은 활용가능한 슬롯, 이 경우에는 LDI_2의 FU3에 배치된다. 다음 명령어 o는 n에 종속되기 때문에, 명령어 o는 활용가능한 슬롯, 이 경우에는 LDI_3의 FU1에 배치된다. 그리고 나서, 그룹 포맷터(123)의 연산은 시퀀셜 머신의 실행 유니트에 의한 실행을 위해 메모리 시스템(107)으로부터 페치되어 디스패치 유니트(115)에 의해 디스패치된, 시퀀스의 다음 명령어로 계속된다.

이것을 수행하기 위해, 포맷팅하는 동안, 분기를 만났을 때, 그룹 포맷터(123)는 분기 명령어를 엔코딩하고, 이 엔코딩된 분기 명령어를 TST(505)의 현재 상태와 함께, 이 분기 명령어에 할당된 LDI의 슬롯, 예를 들어, BP0나 BP1에 저장한다. 그러므로, 각각의 BP 슬롯은 그것이 많은 정보를 포함하기 때문에 폭이 넓다. 그러고 나서, 상기 그룹이 다음에 병렬 엔진(125)의 기능 유니트 및/또는 분기 처리 유니트에 의해 실행될 때, 만일 분기 명령어가 그룹 포맷터(125)에 의해 엔코딩된 경로와는 다른 경로를 갖도록 해결되면, CGB(204)에 저장된 LDI의 현재 그룹과 분기 명령어에 의해 엔코딩된 TST의 상태는 상기한 바와 같이 후속 처리를 위해 그룹 포맷터(123)로 로드된다.

또한, 만약 레지스터 재명명 기술이 상기한 바와 같이 이용되면, 분기를 만났을 때, 그룹 포맷터(123)는 엔코딩된 분기 명령어 및 TST(505)의 현재 상태와 함께, 재명명 정보의 현재 상태를 그 분기 명령어에 할당된 LDI의 슬롯에 저장한다. 그리고 나서, 그룹이 다음에 병렬 엔진의 기능 유니트 및/또는 분기 처리 유니트에 의해 실행될 때, 만약 분기 명령어가 그룹 포맷터(125)에 의해 엔코딩된 경로와는 다른 경로를 갖도록 해결되면, CGB(204)에 저장된 LDI의 현재 그룹 및 분기 명령어에 의해 엔코딩된 재명명 정보의 상태와 TST의 상태가 처리를 위해 그룹 포맷터(123)로 로드된다.

본 발명의 제4 실시예에 따르면, 그룹 포맷터(123)에 의해 생성되어 DIB에 저장된 LDI는 압축될 수 있다. 이러한 압축 연산은 op-필드 엔코더(503)에 의해 실행될 수 있으며, 여기서, 각각의 op-필드는 개별적으로 압축된 다음에, 그룹의 LDI로 삽입되고, 그룹은 DIB에 기록된다. 대안으로, 압축은 각각의 LDI 상에서 수행되거나, 또는 예를 들어, 그룹이 DIB에 기록될 때, 전체적으로 그룹 상에서 수행될 수도 있다. 이 경우에, 도2의 제어 유니트(202)가 DIB 히트 신호를 수신하면(즉, 시퀀스에서 다음 명령어가 DIB에 저장된 그룹의 일부분이 됨), DIB로부터의 압축된 LDI 출력은 제1실시예에 관련하여 상기한 바와 같이 그룹의 LDI를 실행하기 전에 LDI로 확장되어야 한다. 이것을 수행하기 위해, 상기한 바와 같은 도7의 현재 그룹 버퍼가 도20에 도시된 바와 같이 변경될 수 있다.

보다 특정하여 말하면, CGB(204)는 DIB로부터의 압축된 LDI(CLDI) 출력을 수신하는 확장기(expander)(1001)를 포함할 수 있다. 예를 들면, 이 확장기(1001)는 DIB로부터 $CLDI_0$, $CLDI_1$, $CLDI_2$, $CLDI_3$을 수신할 수 있다. 병렬 엔진(125)의 제어 유니트(202)가 DIB 히트 신호를 수신할 때, 동일 머신 사이클에서 확장기(1001)는 DIB로부터 CLDI 출력을 로드하고, 상기 그룹의 첫 번째 CLDI(즉, $CLDI_0$)를 LDI로 확장하고, 이 LDI를 CGB(204)에 LDI_0으로서 기록한다. 다음 사이클에서, 병렬 엔진(125)의 기능 유니트 및 분기 처리 유니트는 제1 실시예에 관련하여 상기한 바와 같이 LDI_0의 op-필드에 명시된 기능을 실행한다. 이것이 진행되고 있는 동안, 확장기(1001)는 나머지 CLDI들($CLDI_1$, $CLDI_2$, $CLDI_3$)을 LDI로 확장하고, 이 LDI(LDI_1, LDI_2, LDI_3)들을 CGB(204)에 기록한다. 다음 사이클에서, 연산은 제1 실시예에 관련해 상기한 바와 같이 LDI_0의 op-필드들에 의해 명시된 기능을 수행하는데 있어서의 병렬 엔진의 결과에 따라, 상기 그룹의 다음 LDI로 계속되거나, 또는 상기 그룹의 다른 LDI로 계속되거나, 또는 그룹 밖의 명령어로 계속된다.

다른 실시예에서는, 그룹 실행 시간을 최소로 하기 위해, 상기한 바와 같이 그룹포맷터(123)에 의해 수행되는 압축 연산이 상기 그룹의 첫 번째 LDI(즉, LDI_0)에 대해서는 디스에이블(disable)될 수도 있다.

전술한 실시예에서, 병렬 엔진(125)의 실행 유니트가 CGB(204)에 저장된 LDI의 op-필드에 의해 엔코딩된 연산의 실행을 종료한 후, 그 실행 결과는 적절한 자원으로 전달되어야 한다. 병렬 엔진(125)이 레지스터 재명명을 이용하지 않는 경우에, 실행 유니트는 매 LDI 이후에 적절한 자원으로 실행 결과를 전달할 수도 있다. 그러나, 병렬 엔진(125)이 레지스터 재명명을 이용하는 경우에는, 실행 유니트는 병렬 엔진(123)이 그룹 밖으로의 분기를 경험하거나, 그룹의 끝을 경험할 때, 실행 결과를 전달해야 한다. 이 경우에, 전달 연산(put away operation)은 필요하다면, 재명명된 자원에 저장된 실행 결과를 재명명 플래그에 따라 원래의 할당된 자원으로 전달하는 기능을 포함한다.

또한, 실행의 흐름이 특수한 방식으로 처리될 소정의 이벤트를 위해 인터럽트될 필요가 있는 상황(예, 페이지 폴트)이 종종 발생한다. 이러한 상황은 인터럽트나 예외로서 불린다. 많은 경우에, 예외가 발생하면, 시스템은 머신의 상태를 정확하게 아는 것이 중요하다. 이것은 본 발명에서와 같이 명령어들이 비-순서적으로 실행될 수 있도록 허용될 때 곤란하게 된다.

이러한 요건을 다루기 위해, 본 발명의 병렬 엔진(125)은 연산이 레지스터 기록연산이든지 또는 메모리 저장 연산이든지 간에, 그 연산의 결과를 그룹의 실행이 완료됐을 경우에만, 그 구성된 레지스터로 위탁하는 것이 바람직하다. 그룹 내에서 예외가 발생하면, 실행되었지만 위탁되지는 않은 명령어의 결과는 폐기되고, 따라서, 머신의 상태는 정확히 그 그룹에 들어갔을 때의 상태가 된다. 이제, 현재 그룹의 주소는 다시 예외 지점에 도달할 때까지 시퀀셜 머신에서 한번에 하나씩 명령어를 페치하고 실행하기 위해 사용된다. 예외가 처리된 이후에, 병렬 엔진(125)의 실행은 DIB에서 히트가 있는 경우에는 그 예외를 유발한 명령어 이후의 명령어에서 다시 시작되거나, 또는 그렇지 않으면 순차적인 방식으로 다시 시작된다.

발명의 효과

전술한 설명으로부터 알 수 있는 바와 같이, 실행의 히스토리를 유지함으로써, 한 번 이상 만나는 명령어들을 실행하기 위해, 짧은 사이클 시간을 갖는 짧은 파이프라인이 사용될 수 있다. 또한, 만일 병렬로 실행될 수 있는 명령어들의 신분(identity)이 이러한 명령어들을 만난 첫 번째에서 유지된다면, 이러한 동일한 명령어들은 그 실행을 위해 적당한 기능 유니트가 활용가능한 경우에 모든 다음 번에서 병렬로 실행될 수 있다.

병렬로 실행될 수 있는 명령어를 디코딩하고 판단하기 위한 로직은 이들 명령어들이 언급된 것보다 많은 스테이지를 가진 파이프라인 상에서 복수의 사이클에서 실행될 수 있으므로 복잡할 필요가 없다. 기대하고 또한 실제적으로 관찰되는 것에 의하면, 복수의 사이클을 사용하여 이러한 연산을 수행하는 비용은 짧은 파이프라인 상에서 병렬로 이들 동일한 명령어를 반복 실행하는 것에 의해 그 이상으로 보상받을 수 있다.

프리-디코드된 명령어 캐쉬(predecoded instruction cache)와 같은 이전에 제안된 방식과 비교하면, 이러한 방식은 몇 가지 장점을 갖고 있다. 첫 번째로는, DIB가 프로그램의 실행 중에 실제적으로 만나는 명령어만을 포함하기 때문에 낭비되는 작업이 거의 없다. 그러나, 프리-디코드된 명령어 캐쉬는 명령어를 만나는 첫 번째 조차도 실행 경로를 예상해야할 필요가 있다. 두 번째로는, 사용된 분기를 통해 명령어를 병합하는데 있어 추가적인 비용이 들지 않는다. 세 번째로는, 복수의 분기 경로가 다음 메모리 레벨로부터 페치될 가능성이 있는, 이들 경로로부터의 명령어를 대기할 필요없이, 디코드된 명령어 그룹으로 동적

으로 병합될 수 있다. 네 번째로는, 엔코딩 메커니즘이 예상되는 대기시간이 실제적인 대기시간으로 동적으로 조정한다. 다섯 번째로는, 이러한 방식은 기존의 명령어 셋트 아키텍쳐와 잘 동작하며, 재컴파일에 종속되지 않는다.

종래의 수퍼스칼라 방식과 비교하여, 이 방식은 몇 가지 장점을 갖고 있다. 첫 번째로는, 이들 연산을 복수의 사이클에서 수행함으로써, 디스패치 및 디코드 로직의 복잡도가 크게 감소된다. 두 번째로는, 동일한 이유로 인해, 프로세서의 사이클 시간이 더욱 작아진다. 세 번째로는, 코드를 통한 단지 하나의 예상 경로가 아니라, 코드를 통한 여러 개의 경로로부터 연산을 병합함으로써 기능 유니트의 보다 양호한 이용이 이루어질 수 있다. 네 번째로는, 분기 예측 분기 히스토리 테이블의 필요성이 그러한 모든 정보를 명령어 스트림 히스토리에 포함시킴으로써 제거된다.

VLIW 머신과 비교하여, 이 방식은 다음의 장점을 갖고 있다. 첫 번째로, 이 방식은 어떠한 기존의 명령어 셋트 아키텍쳐와도 동작하고, 새로운 명령어 셋트가 정의될 필요가 없다. 두 번째로, 기능 유니트의 수와 DIB의 내부 엔코딩이 컴파일된 코드에 영향을 주지 않고, 기술적 능력/제한 및/또는 프로세서 비용 제한에 적합하도록 변경될 수 있다.

비록 본 발명이 그 특정 실시예들을 참조하여 도시되고 설명되었지만, 이 기술분야에 통상의 지식을 가진 자는, 본 발명의 사상 및 범위를 벗어나지 않고, 그 형태와 세부사항에 대한 상기 및 다른 변경이 이루어질 수 있다는 것을 이해할 것이다.

(57) 청구의 범위

청구항 1

적어도 하나의 제1 실행 유니트에 의해 실행될 명령어들을 저장하기 위한 제 1 메모리 : 페치 제어 신호에 따라 상기 제1 메모리로부터 명령어 큐로 상기 명령어들을 페치하기 위한 명령어 페치 유니트 : 상기 적어도 하나의 제1 실행 유니트에 의한 실행을 위해 상기 명령어 큐에 저장된 명령어들을 디스패치하기 위한 디스패치 유니트 : 및상기 적어도 하나의 제1 실행 유니트에 의해 수행되는 명령어들의 실행에 따라 상기 페치 제어 신호를 갱신하기 위한 수단을 구비한 제1 처리 엔진 : 다수의 제2 실행 유니트를 포함하는 대체 엔코딩 실행 수단 : 상기 제1 처리 엔진에 의한 상기 명령어 시퀀스들의 실행과 동시에, 상기 디스패치 유니트에 의해 실행을 위해 디스패치된 명령어 시퀀스들의 대체 엔코딩(alternate encoding)-여기서, 상기 대체 엔코딩은 긴 디코드된 명령어(LDI)들의 셋트를 포함하고, 각각의 긴 디코드된 명령어(LDI)는 상기 다수의 제2 실행 유니트 중 하나에 각각 대응하는 op-필드의 셋트를 포함하고, 각각의 op-필드는 대응하는 제2 실행 유니트에 의해 실행가능함 - 을 생성하기 위한 그룹 포맷터 : 및 상기 그룹 포맷터에 의해 생성된 상기 대체 엔코딩을 저장하기 위한 제2 메모리를 포함하고, 상기 페치 제어 신호가 상기 제2 메모리에 저장된 상기 대체 엔코딩에 대응한다는 것을 검출할 때, 상기 다수의 제2 실행 유니트는 상기 제2 메모리에 저장된 상기 대체 엔코딩을 실행하는 컴퓨터 처리 장치.

청구항 2

제1항에 있어서, 상기 그룹 포맷터는 상기 제1 처리 엔진의 상기 실행 유니트에 의한 상기 명령어 시퀀스들의 실행 다음에 상기 대체 엔코딩을 생성하는 컴퓨터 처리 장치.

청구항 3

제1항에 있어서, 상기 그룹 포맷터에 의해 생성된 상기 대체 엔코딩은 병렬 실행에 적합한 컴퓨터 처리 장치.

청구항 4

원고누락

청구항 5

제1항에 있어서, 상기 대체 엔코딩 실행 수단의 상기 제2 실행 유니트 중 적어도 하나는 상기 제1 처리 엔진의 상기 제1 실행 유니트와 다른 컴퓨터 처리 장치.

청구항 6

제1항에 있어서, 상기 대체 엔코딩 실행 수단의 상기 다수의 제2 실행 유니트 중 하나는 상기 제1 처리 엔진의 상기 제1 실행 유니트를 포함하는 컴퓨터 처리 장치.

청구항 7

원고누락

청구항 8

제1항에 있어서, 각각의 op-필드는 단일 머신 사이클에서 상기 대체 엔코딩 실행 수단의 대응하는 제2실행 유니트에 의해 실행될 수 있는 컴퓨터 처리 장치.

청구항 9

제1항에 있어서, 상기 대체 엔코딩 실행 수단은 상기 op-필드가 사용/ 및/또는 정의할 수 있는 다수의 레지스터 자원들을 포함하고, 상기 그룹 포맷터는, 상기 디스패치 유니트에 의해 실행을 위해 디스패치된 상기 명령어 시퀀스들에 따라 상기 op-필드들을 생성하는 op-필드 엔코더 ; 상기 각각의 레지스터 자원과 관련된 타임 스탬프 식별자들 - 여기서, 상기 타임 스탬프 식별자들은 상기 긴 디코드된 명령어들의 셋트의 어떤 LDI 상에서 상기 관련된 레지스터 자원이 활용가능하게 될 것인지를 식별함 - 을 저장하기 위한

타임 스탬프 테이블 ; 및 상기 명령어 시퀀스 내의 명령어들 사이의 종속성과, 상기 타임 스탬프 테이프블에 저장된 상기 타임 스탬프 식별자, 및 상기 op-필드 엔코더에 의해 생성된 상기 op-필드들에 따라, 상기 긴 디코드된 명령어들의 셋트를 구축하는 할당기를 포함하는 컴퓨터 처리 장치.

청구항 10

제9항에 있어서, 상기 할당기는 상기 명령어 시퀀스 내의 명령어들의 대기시간(latency)에 따라 상기 긴 디코드된 명령어들의 셋트를 구축하는 컴퓨터 처리 장치.

청구항 11

제9항에 있어서, 상기 할당기는 상기 명령어 시퀀스 내의 하나의 명령어가 사전-정의된 정지 명령어와 대응하는 것을 검출할 때, 상기 긴 디코드된 명령어들의 셋트를 폐쇄하는 컴퓨터 처리 장치.

청구항 12

제9항에 있어서, 상기 명령어 시퀀스 중 하나의 명령어가 현재 그룹의 op-필드에 의해 이전에 정의된 자원을 정의하는 것을 검출할 때, 상기 할당기는 상기 자원을 자유(free)자원으로 재명명하는 컴퓨터 처리 장치.

청구항 13

제9항에 있어서, 상기 타임 스탬프 테이블은 로드 연산 및 저장 연산과 각각 관련된 상기 타임 스탬프 식별자들을 포함하고 : 상기 할당기는 상기 로드 연산 및 저장 연산의 순서를 유지하기 위해, 상기 로드 연산 및 저장 연산과 관련된 상기 타임 스탬프 식별자들을 이용하는 컴퓨터 처리 장치.

청구항 14

제1항에 있어서, 상기 제2메모리는, 상기 긴 디코드된 명령어들의 셋트를 각각 포함하는 엔트리들의 어레이 ; 상기 엔트리들의 어레이 중 하나의 엔트리가 상기 페치 제어 신호에 대응하는지를 판단하기 위한 수단 ; 및 상기 하나의 엔트리가 상기 페치 제어 신호에 대응한다고 판단할 때, 상기 하나의 엔트리의 상기 긴 디코드된 명령어들의 셋트를 판독하기 위한 수단을 포함하고, 상기 대체 엔코딩 실행 수단은, 상기 제2 메모리로부터 판독된 상기 긴 디코드된 명령어들의 셋트를 저장하기 위한 버퍼를 포함하고, 상기 다수의 제2 실행 유니트들은 상기 버퍼에 저장된 상기 긴 디코드된 명령어들의 셋트의 상기 op-필드들을 실행하는 컴퓨터 처리 장치.

청구항 15

제14항에 있어서, 상기 대체 엔코딩 실행 수단은 상기 op-필드들의 실행 결과에 따라, 상기 페치 제어 신호를 갱신하기 위한 수단을 더 포함하는 컴퓨터 처리 장치.

청구항 16

제14항에 있어서, 상기 다수의 제2 실행 유니트들은 각각의 LDI의 실행을 종료한 이후에, 각각의 LDI의 상기 op-필드의 실행 결과를 적절한 레지스터 자원들로 전달하는(put away) 컴퓨터처리 장치.

청구항 17

제14항에 있어서, 상기 다수의 제2 실행 유니트들은 상기 긴 디코드된 명령어들의 셋트의 실행을 종료한 이후에, 상기 긴 디코드된 명령어들의 셋트의 상기 op-필드들의 실행 결과를 전달하는 컴퓨터 처리 장치.

청구항 18

제14항에 있어서, 소정의 예외를 경함할 때, 상기 버퍼에 저장된 상기 긴 디코드된 명령어들의 셋트 내의 엔코딩된 제1 명령어에 대응하도록 상기 페치 제어 신호를 갱신하고 : 상기 제1 처리 엔진과 관련된 레지스터 자원들을 상기 제1 명령어 이전의 그 원래의 상태로 갱신하고 ; 상기 제1 처리 엔진으로 하여금 상기 제1 명령어의 처리를 시작할 수 있도록 하는 예외 제어기를 더 포함하는 컴퓨터 처리 장치.

청구항 19

제14항에 있어서, 상기 제2 메모리에 저장된 하나 또는 그 이상의 상기 긴 디코드된 명령어들은 압축되고 : 상기 대체 엔코딩 실행 수단은 실행 이전에 압축된 각각의 긴 디코드된 명령어를 확장하기 위한 수단을 포함하는 컴퓨터 처리 장치.

청구항 20

제14항에 있어서, 상기 버퍼에 저장된 상기 긴 디코드된 명령어들의 셋트의 상기 op-필드들 중 하나가 상기 버퍼에 저장된 상기 긴 디코드된 명령어들의 셋트로부터의 분기 명령어로서 해결된 것을 검출할 때, 후속 처리를 위해 상기 버퍼에 현재 저장된 상기 긴 디코드된 명령어들의 셋트를 상기 그룹 포맷터로 로드하기 위한 수단을 더 포함하는 컴퓨터 처리 장치.

청구항 21

명령어들이 제1메모리에 저장되어 있는 컴퓨터 처리 시스템에서, 상기 명령어들을 실행하는 방법에 있어서, 페치 제어 신호에 따라 상기 제1메모리로부터 명령어 큐로 명령어들을 페치하는 단계 : 적어도 하나의 제1실행 유니트에 의한 실행을 위해 상기 명령어 큐에 저장된 명령어를 디스패치하는 단계 : 상기 적어도 하나의 제1실행 유니트에 의해 수행되는 명령어 실행에 따라, 상기 페치 제어 신호를 갱신하는 단계 : 및 상기 적어도 하나의 제1실행 유니트에 의한 상기 명령어 시퀀스들의 실행과 동시에, 실행을 위해 디스패치된 명령어 시퀀스들의 대체 엔코딩-여기서, 상기 대체 엔코딩은 긴 디코드된 명령어(LDI)들의 셋트

를 포함하고, 각각의 긴 디코드된 명령어(LDI)는 다수의 제2 실행 유니트 중 하나에 각각 대응하는 op-필드의 셋트를 포함하고, 각각의 op-필드는 대응하는 제2실행 유니트에 의해 실행가능함 -을 생성하는 단계 ; 그룹 포맷터에 의해 생성된 상기 대체 엔코딩을 제2메모리에 저장하는 단계 ; 및 상기 페치 제어 신호가 상기 제2메모리에 저장된 상기 대체 엔코딩에 대응한다는 것을 검출할 때, 상기 제2메모리에 저장된 상기 대체 엔코딩을 실행하도록 상기 다수의 제2실행 유니트들을 제어하는 단계를 포함하는 명령어 실행 방법.

청구항 22

제21항에 있어서, 상기 대체 엔코딩은 상기 실행 유니트에 의한 상기 명령어 시퀀스들의 실행 다음에 생성되는 명령어 실행 방법.

청구항 23

제21항에 있어서, 상기 대체 엔코딩은 상기 다수의 제2 실행 유니트들에 의한 병렬 실행에 적합한 명령어 실행 방법.

청구항 24

원고누락

청구항 25

원고누락

청구항 26

제21항에 있어서, 각각의 op-필드는 단일머신 사이클에서 상기 대체 엔코딩 실행 수단의 대응하는 제2실행 유니트에 의해 실행될 수 있는 명령어 실행 방법.

청구항 27

제21항에 있어서, 상기 컴퓨터 처리 시스템은, 상기 op-필드들이 사용 및/또는 정의할 수 있는 다수의 레지스터 자원 ; 실행을 위해 디스패치된 상기 명령어 시퀀스들에 따라 상기 op-필드를 생성하는 op-필드 엔코더 ; 및 상기 각각의 레지스터 자원과 관련된 타임 스탬프 식별자들 - 여기서, 상기 타임 스탬프 식별자들은 상기 긴 디코드된 명령어들의 셋트의 어떤 LDI 상에서 상기 관련된 레지스터 자원이 활용가능하게 될 것인지를 식별함 - 을 저장하기 위한 타임 스탬프 테이블을 포함하고, 상기 방법은, 상기 명령어 시퀀스 내의 명령어들 사이의 종속성과, 상기 타임 스탬프 테이블에 저장된 상기 타임 스탬프 식별자, 및 상기 op-필드 엔코더에 의해 생성된 상기 op-필드들에 따라, 상기 긴 디코드된 명령어들의 셋트를 구축하는 단계를 더 포함하는 명령어 실행 방법.

청구항 28

제27항에 있어서, 상기 긴 디코드된 명령어의 셋트는 상기 명령어 시퀀스 내의 명령어들의 대기시간에 따라 구축되는 명령어 실행 방법.

청구항 29

제27항에 있어서, 상기 명령어 시퀀스 내의 하나의 명령어가 사전-정의된 정지 명령에 대응한다는 것을 검출할 때, 상기 긴 디코드된 명령어들의 셋트를 폐쇄하는 단계를 더 포함하는 명령어 실행 방법.

청구항 30

(2회정정) 제27항에 있어서, 상기 명령어 시퀀스 중 하나의 명령어가 상기 긴 디코드된 명령어들의 셋트 내의 하나의 LDI의 op-필드에 의해 이전에 정의된 자원을 정의한다는 것을 검출할 때, 상기 자원을 자유 자원(free resource)으로 재명명하는 단계를 더 포함하는 명령어 실행 방법.

청구항 31

제27항에 있어서, 상기 타임 스탬프 테이블은 로드 연산 및 저장 연산과 각각 관련된 타임 스탬프 식별자들을 포함하고, 상기 방법은, 상기 로드 연산 및 저장 연산과 관련된 상기 타임 스탬프 식별자들에 따라 상기 로드 연산 및 저장 연산의 순서를 유지하는 단계를 더 포함하는 명령어 실행 방법.

청구항 32

제21항에 있어서, 상기 제2 메모리는 상기 긴 디코드된 명령어들의 셋트를 각각포함하는 엔트리들의 어레이를 포함하고, 상기 방법은, 상기 엔트리들의 어레이 중 하나의 엔트리가 상기 페치 제어 신호에 대응하는지를 판단하는 단계 ; 상기 하나의 엔트리가 상기 페치 제어 신호에 대응한다고 판단할 때, 상기 하나의 엔트리의 상기 긴 디코드된 명령어들의 셋트를 판독하는 단계 : 상기 제2 메모리로부터 판독된 상기 긴 디코드된 명령어들의 셋트를 저장하는 단계 : 및 상기 버퍼에 저장된 상기 긴 디코드된 명령어들의 셋트의 상기 op-필드들을 실행하는 단계를 더 포함하는 명령어 실행 방법.

청구항 33

제32항에 있어서, 상기 op-필드들의 실행 결과에 따라 상기 페치 제어 신호를 갱신하는 단계를 더 포함하는 명령어 실행 방법.

청구항 34

제32항에 있어서, 각각의 LDI의 실행을 종료한 후 상기 각각의 LDI의 상기 op-필드들의 실행 결과를 적절

한 레지스터 자원으로 전달하는 단계를 더 포함하는 명령어 실행 방법.

청구항 35

제32항에 있어서, 상기 긴 디코드된 명령어들의 셋트의 실행을 종료한 후 상기 긴 디코드된 명령어들의 셋트이 상기 op-필드들의 실행 결과를 전달하는 단계를 더 포함하는 명령어 실행 방법.

청구항 36

제32항에 있어서, 소정의 예외를 경험할 때, 상기 버퍼에 저장된 상기 긴 엔코딩된 명령어들의 셋트 내의 엔코딩된 제1 명령어에 대응하도록 상기 페치 제어 신호를 갱신하는 단계 : 상기 적어도 하나의 실행 유니트와 관련된 레지스터 자원들을 상기 제1 명령어 이전의 그 원래의 상태로 갱신하는 단계 : 및 상기 적어도 하나의 실행 유니트에 의한 실행을 위해 상기 제1 명령어의 처리를 가능하게 하는 단계를 더 포함하는 명령어 실행 방법.

청구항 37

제32항에 있어서, 상기 제2 메모리에 상기 하나 또는 그 이상의 긴 디코드된 명령어들을 저장하기 전에, 상기 하나 또는 그 이상의 긴 디코드된 명령어들을 압축하는 단계 ; 및 상기 하나 또는 그 이상의 긴 디코드된 명령어들을 실행하기 전에, 압축된 각각의 긴 디코드된 명령어를 확장하는 단계 더 포함하는 명령어 실행 방법.

청구항 38

제32항에 있어서, 상기 버퍼에 저장된 상기 긴 디코드된 명령어들의 셋트의 상기 op-필드들 중 하나가 상기 버퍼에 저장된 상기 긴 디코드된 명령어들의 셋트로부터의 분기 명령어로서 해결된 것을 검출할 때, 상기 버퍼에 현재 저장된 상기 긴 디코드된 명령어들의 셋트를 갱신하는 단계를 더 포함하는 명령어 실행 방법.

청구항 39

제32항에 있어서, 상기 버퍼에 저장된 상기 긴 디코드된 명령어들의 셋트의 상기 갱신 단계는, 상기 명령어 시퀀스들 및 그 명령어 시퀀스들 내의 명령어들 사이의 종속성에 따라, 적어도 하나의 긴 디코드된 명령어를 상기 긴 디코드된 명령어들의 셋트에 추가하는 단계를 포함하는 명령어 실행 방법.

청구항 40

제39항에 있어서, 상기 적어도 하나의 긴 디코드된 명령어는 상기 명령어 시퀀스들 내에서의 상기 명령어의 대기시간에 따라, 상기 버퍼에 저장된 상기 긴 디코드된 명령어들의 셋트에 추가되는 명령어 실행 방법.

청구항 41

제1항에 있어서, 상기 명령어 시퀀스들 중 적어도 하나의 시퀀스 S는 분기 명령어 B, 상기 분기 명령어 B에 선행하는 명령어 PB, 및 상기 분기 명령어 B 다음에 오는 명령어 FB를 포함하고, 상기 시퀀스 S에 대응하는 상기 긴 디코드된 명령어들의 셋트는 상기 분기 명령어 B, 상기 명령어 PB, 및 상기 명령어 FB에 대응하는 op-필드들을 포함하는 컴퓨터 처리 장치.

청구항 42

제41항에 있어서, 상기 명령어 시퀀스에서 상기 명령어 PB와 상기 명령어 FB 중 하나의 위치는 상기 분기 명령어 B에 인접해 있지 않은 컴퓨터 처리 장치.'

청구항 43

제41항에 있어서, 상기 명령어 FB는 상기 분기 명령어 B의 실행 결과에 근거하여 결정되는 컴퓨터 처리 장치.

청구항 44

제41항에 있어서, 상기 시퀀스 S는 복수의 분기 명령어를 포함하고, 상기 상기 시퀀스 S에 대응하는 상기 긴 디코드된 명령어들의 셋트는 상기 복수의 분기 명령어 각각에 대응하는 op-필드들을 포함하는 컴퓨터 처리 장치.

청구항 45

제44항에 있어서, 상기 시퀀스 S는 상기 복수의 분기 명령어의 실행 결과에 근거하여 결정되는 컴퓨터 처리 장치.

청구항 46

제21항에 있어서, 상기 명령어 시퀀스들 중 적어도 하나의 시퀀스 S는 분기 명령어 B, 상기 분기 명령어에 선행하는 명령어 PB, 및 상기 분기 명령어 다음에 오는 명령어 FB를 포함하고, 상기 시퀀스 S에 대응하는 상기 긴 디코드된 명령어들의 셋트는 상기 분기 명령어 B, 상기 명령어 PB, 및 상기 명령어 FB에 대응하는 op-필드들을 포함하는 명령어 실행 방법.

청구항 47

제46항에 있어서, 상기 명령어 시퀀스에서 상기 명령어 PB와 상기 명령어 FB 중 하나의 위치는 상기 분기

명령어 B에 인접해 있지 않은 명령어 실행 방법.

청구항 48

제46항에 있어서, 상기 명령어 FB는 상기 분기 명령어 B의 실행 결과에 근거하여 결정되는 명령어 실행 방법.

청구항 49

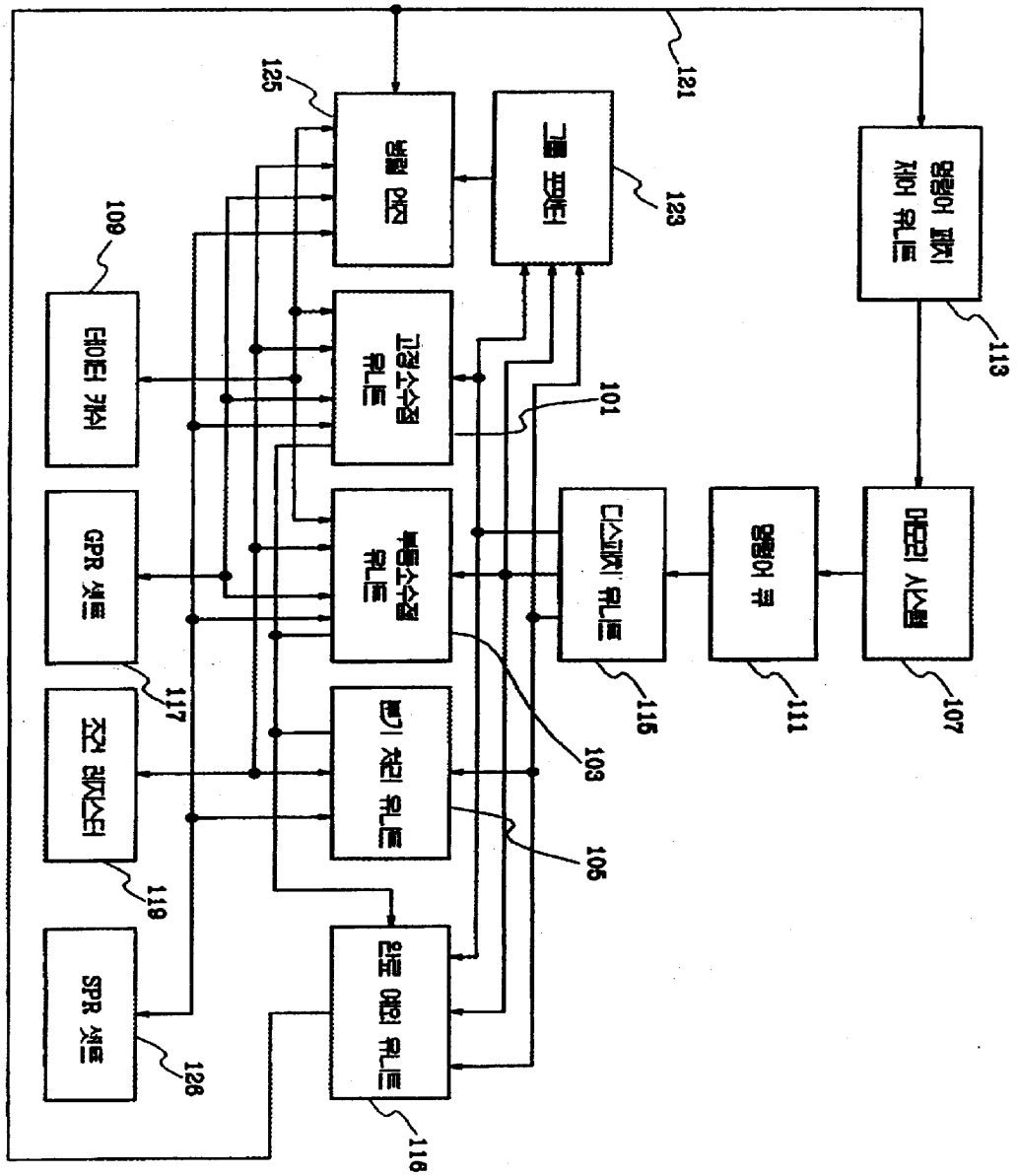
제46항에 있어서, 상기 시퀀스 S는 복수의 분기 명령어를 포함하고, 상기 상기 시퀀스 S에 대응하는 상기 긴 디코드된 명령어들의 셋트는 상기 복수의 명령어 각각에 대응하는 op-필드들을 포함하는 명령어 실행 방법.

청구항 50

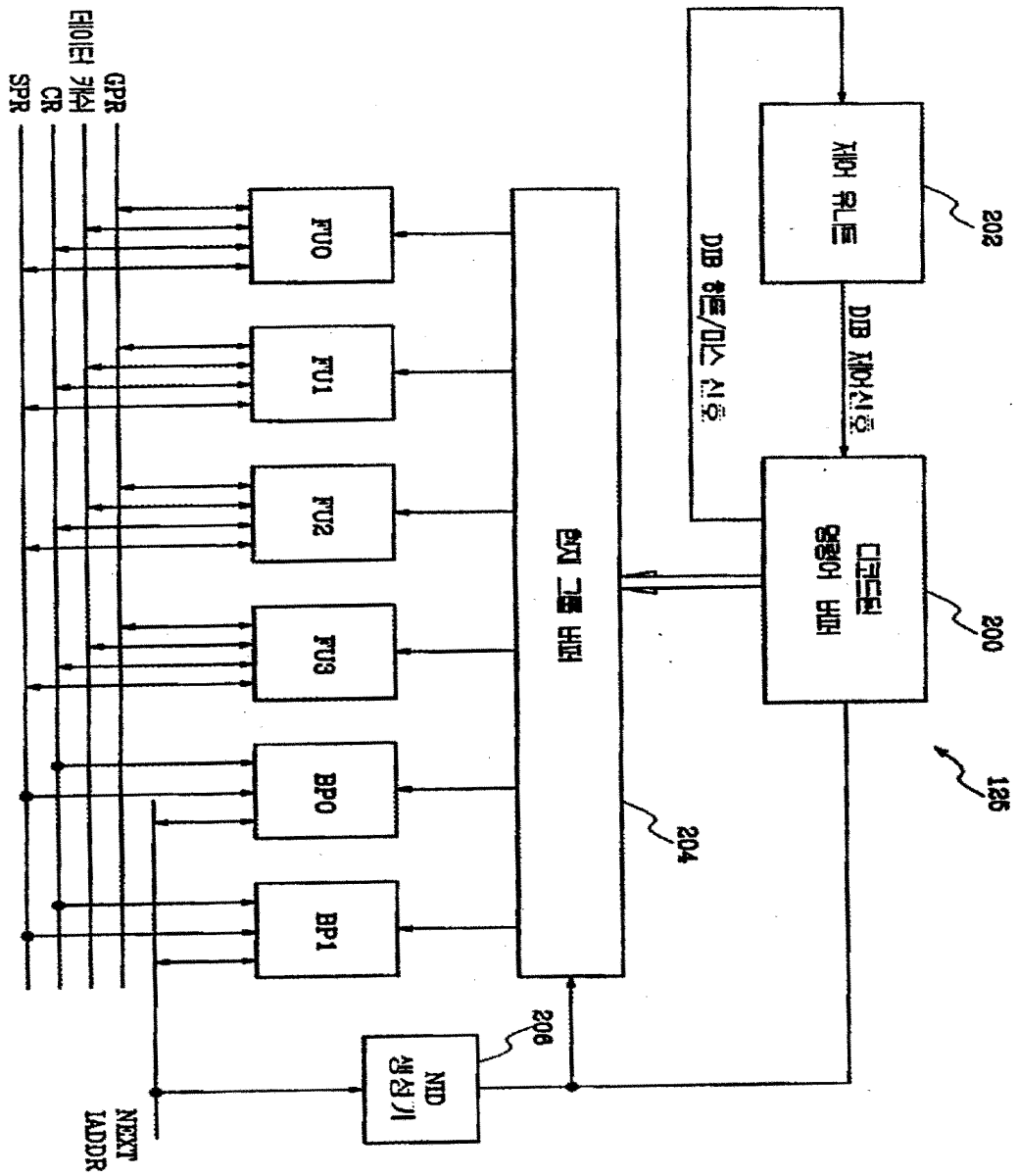
제49항에 있어서, 상기 시퀀스 S는 상기 복수의 분기 명령어의 실행 결과에 근거하여 결정되는 명령어 실행 방법.

도면

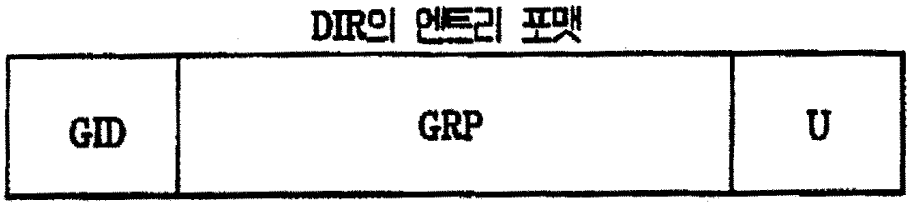
도면1



도면2


GPR
데이터 캐쉬
CR
SPR
FU0
FU1
FU2
FU3
BP0
BP1
NEXT IADDR
NID 생성기
206
레지스터 버퍼
204
디코드된 명령어 버퍼
200
125
제어 유닛
202
DIB 제어신호
DIB 히트/미스 신호

도면3

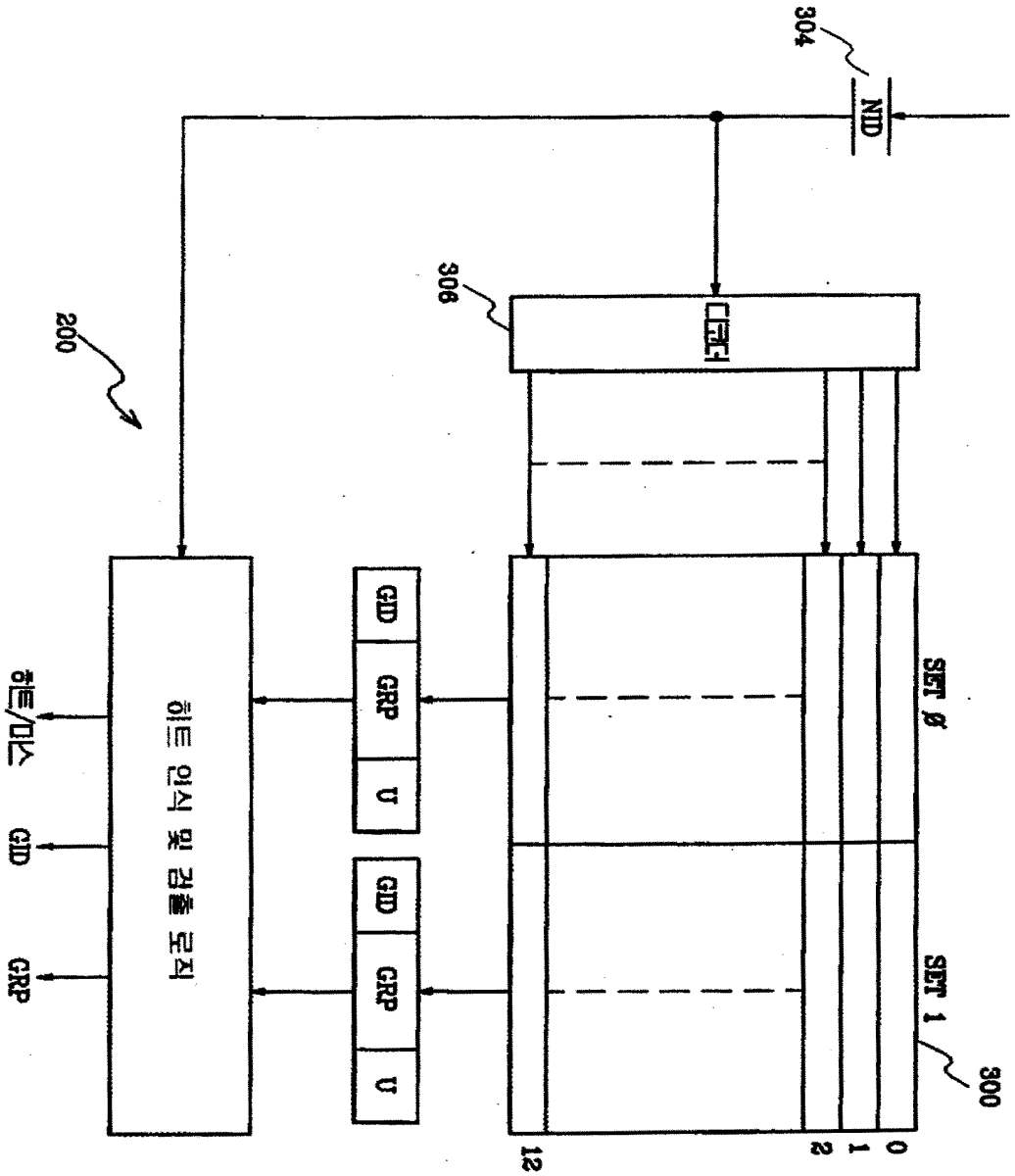

DIR의 엔트리 포맷
GID
GRP
U

도면4

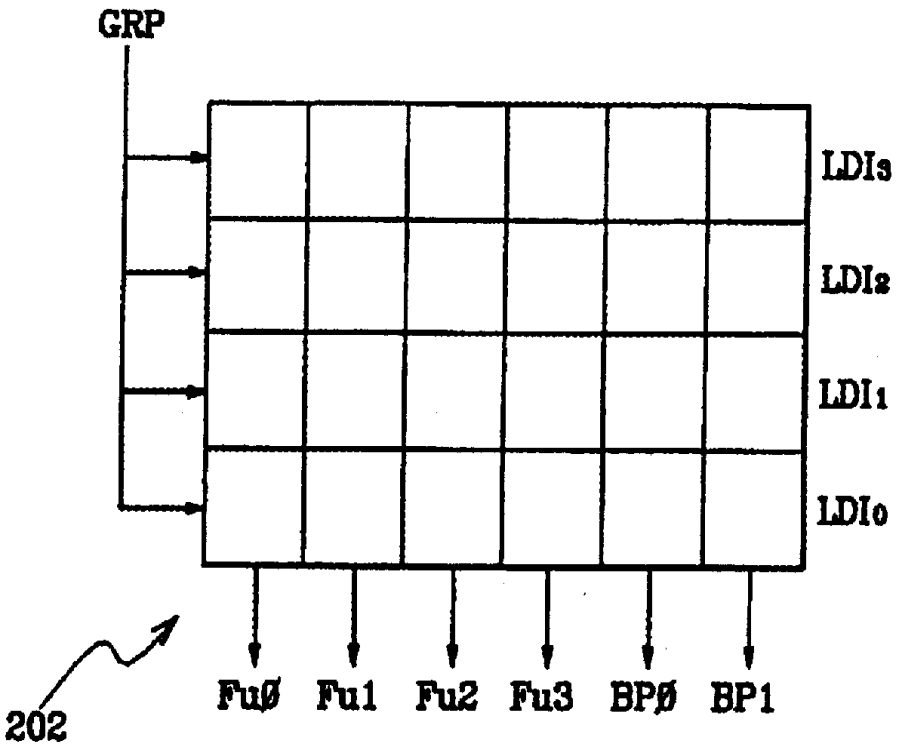
CGB의 팻

LDI_3	$FUØ_3$	$FU1_3$	$FU2_3$	$FU3_3$	$BPØ_3$	$BP1_3$
LDI_2	$FUØ_2$	$FU1_2$	$FU2_2$	$FU3_2$	$BPØ_2$	$BP1_2$
LDI_1	$FUØ_1$	$FU1_1$	$FU2_1$	$FU3_1$	$BPØ_1$	$BP1_1$
LDI_0	$FUØ_0$	$FU1_0$	$FU2_0$	$FU3_0$	$BPØ_0$	$BP1_0$

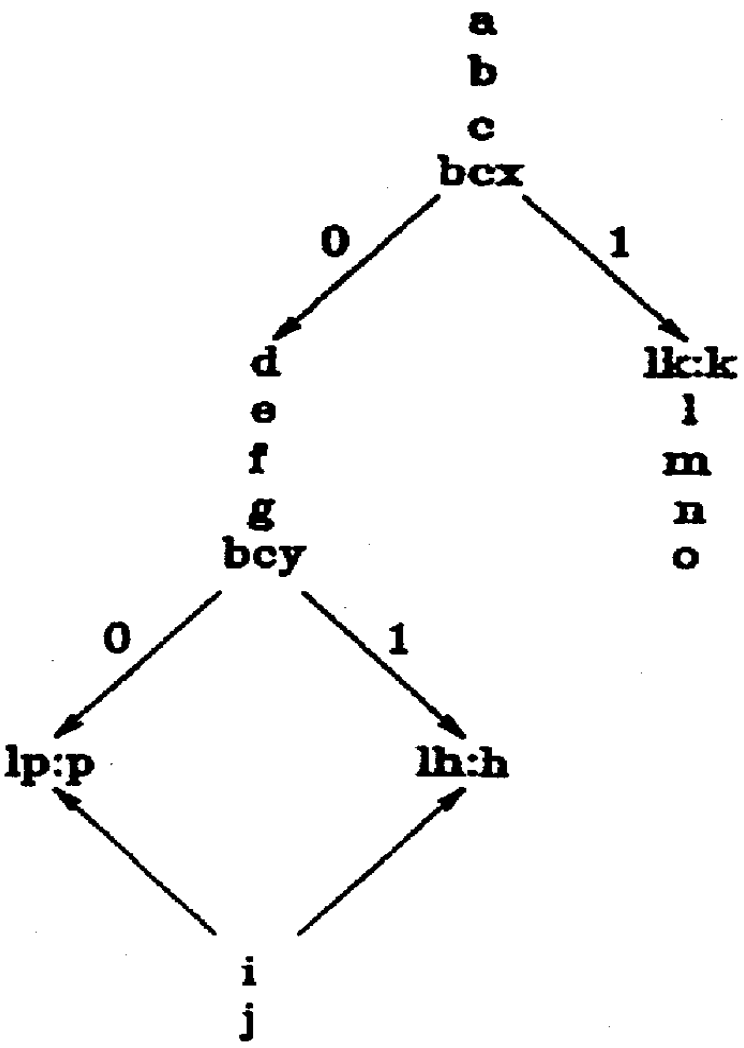
도면5


304
NID
306
디코더
200
워드 인식 및 검출 로직
히트/미스
GID
GRP
U
SET 0
SET 1
300
0
1
2
12

도면6



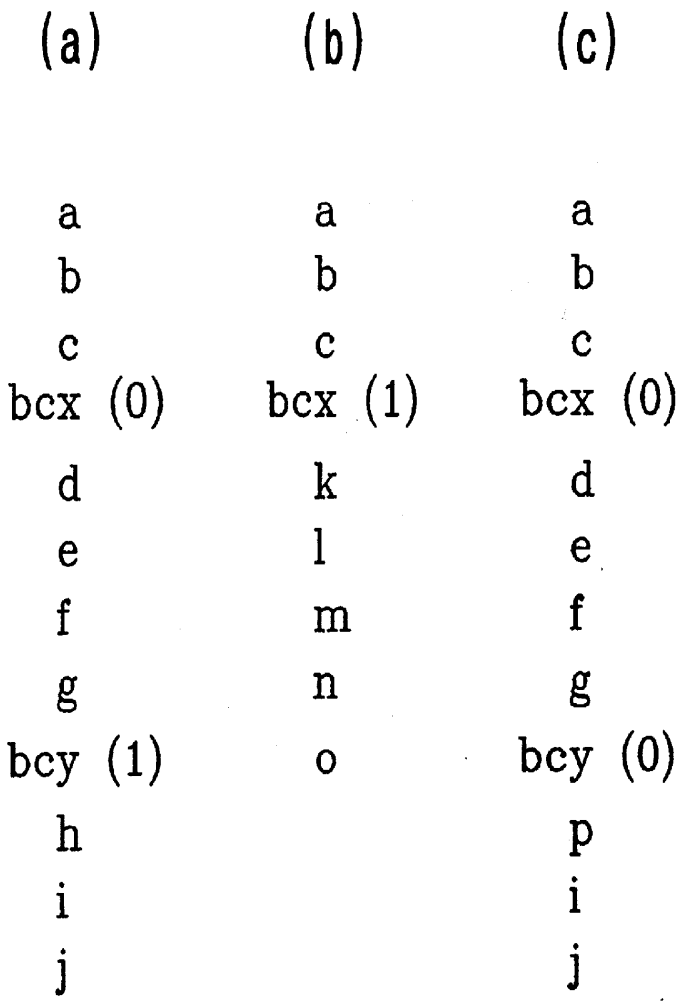
도면7



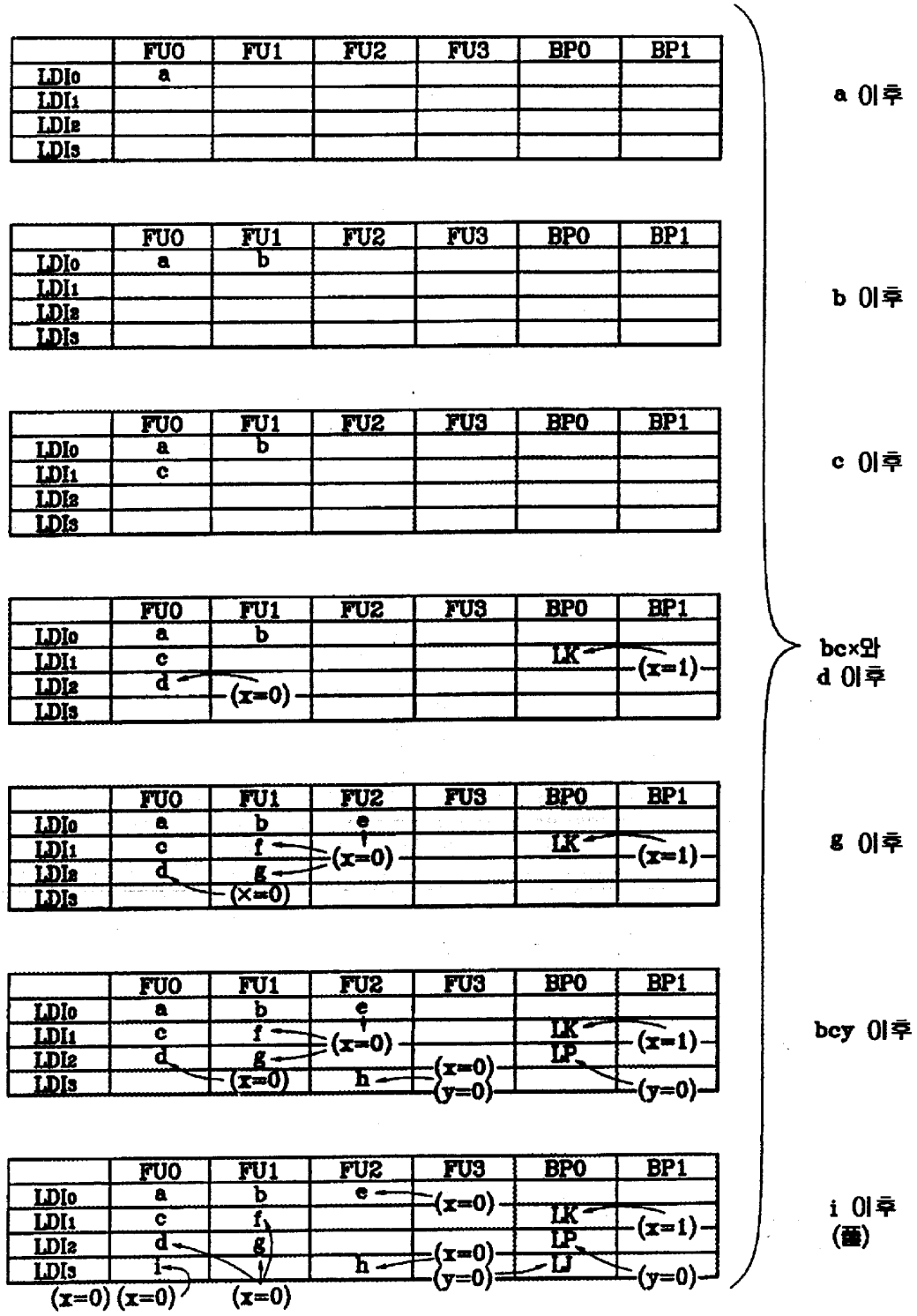
도면8

명령어	종속성
a	
b	
c	a,b
d	a,c
e	–
f	e
g	e,f
h	b,f
i	d,g
j	i
k	–
l	k
m	c,k
n	–
o	n
p	e
bcx	a
bcy	e

도면9


(a)
(b)
(c)
a
b
c
bcx (0)
d
e
f
g
bcy (1)
h
i
j
a
b
c
bcx (1)
k
l
m
n
o
a
b
c
bcx (0)
d
e
f
g
bcy (0)
p
i
j

도면10a


FU0 FU1 FU2 FU3 BP0 BP1
LDI0
LDI1
LDI2
LDI3
a
b
c
d
e
f
g
h
i
(x=0)
(y=0)
LK
LP
LJ
(x=1)
a 이후
b 이후
c 이후
bcx와
d 이후
g 이후
bcy 이후
i 이후
(끝)

도면10b

	FU0	FU1	FU2	FU3	BP0	BP1
LDI0	a	b	k (x=1)	n (x=1)		
LDI1	c	1 (x=1)	0 (x=1)		LD= (x=1)	
LDI2	m (x=1)					
LDI3						

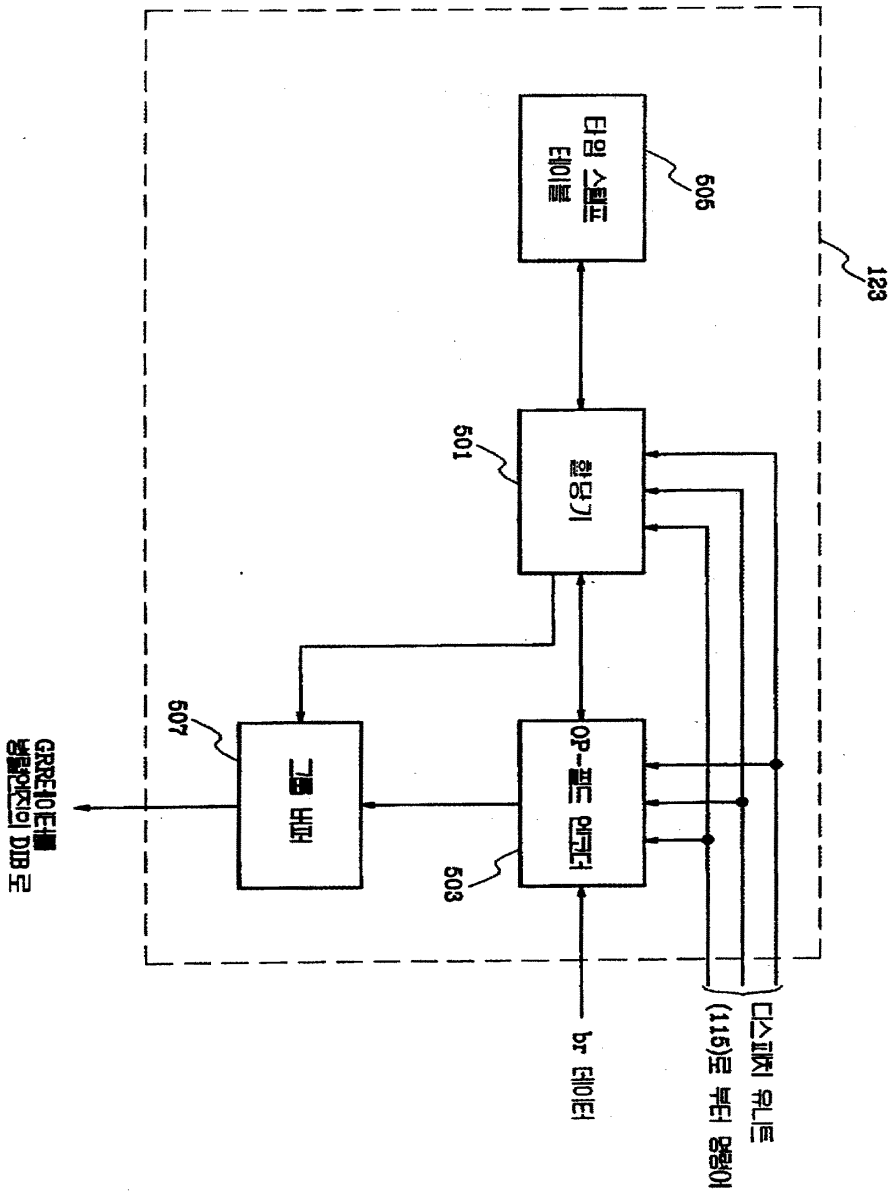
0 이후
(아직 끝이 아님)

도면10c

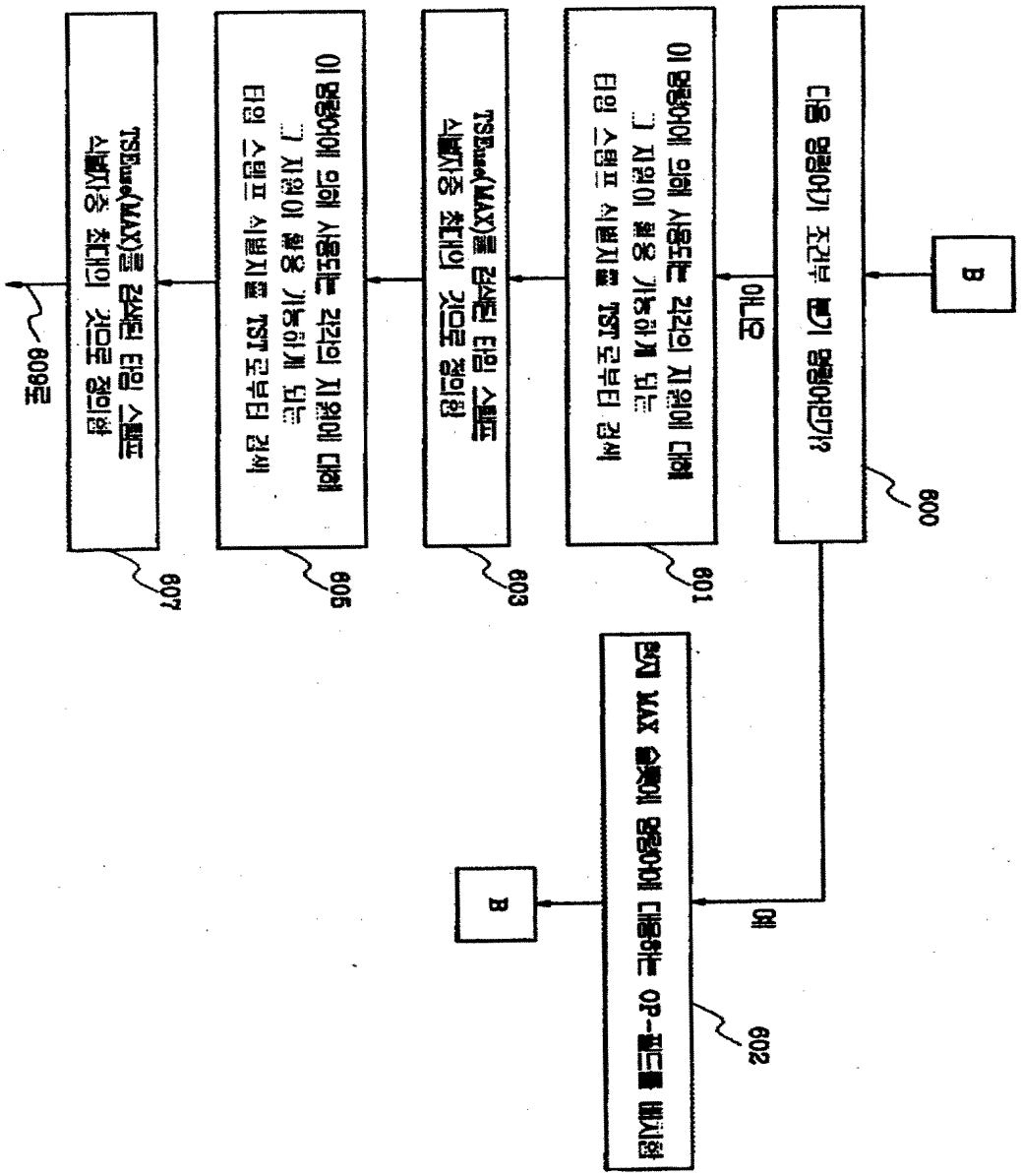
	FU0	FU1	FU2	FU3	BP0	BP1
LDI0	a	b	e (x=1)			
LDI1	c	f (x=1)	p (x=1) (y=1)		LD= (x=1)	
LDI2	d (x=0)	g (x=1)			LH= (Y=1)	
LDI3	i (x=0) (y=0)				LJ= (x=0) (y=0)	

1 이후
(끝)

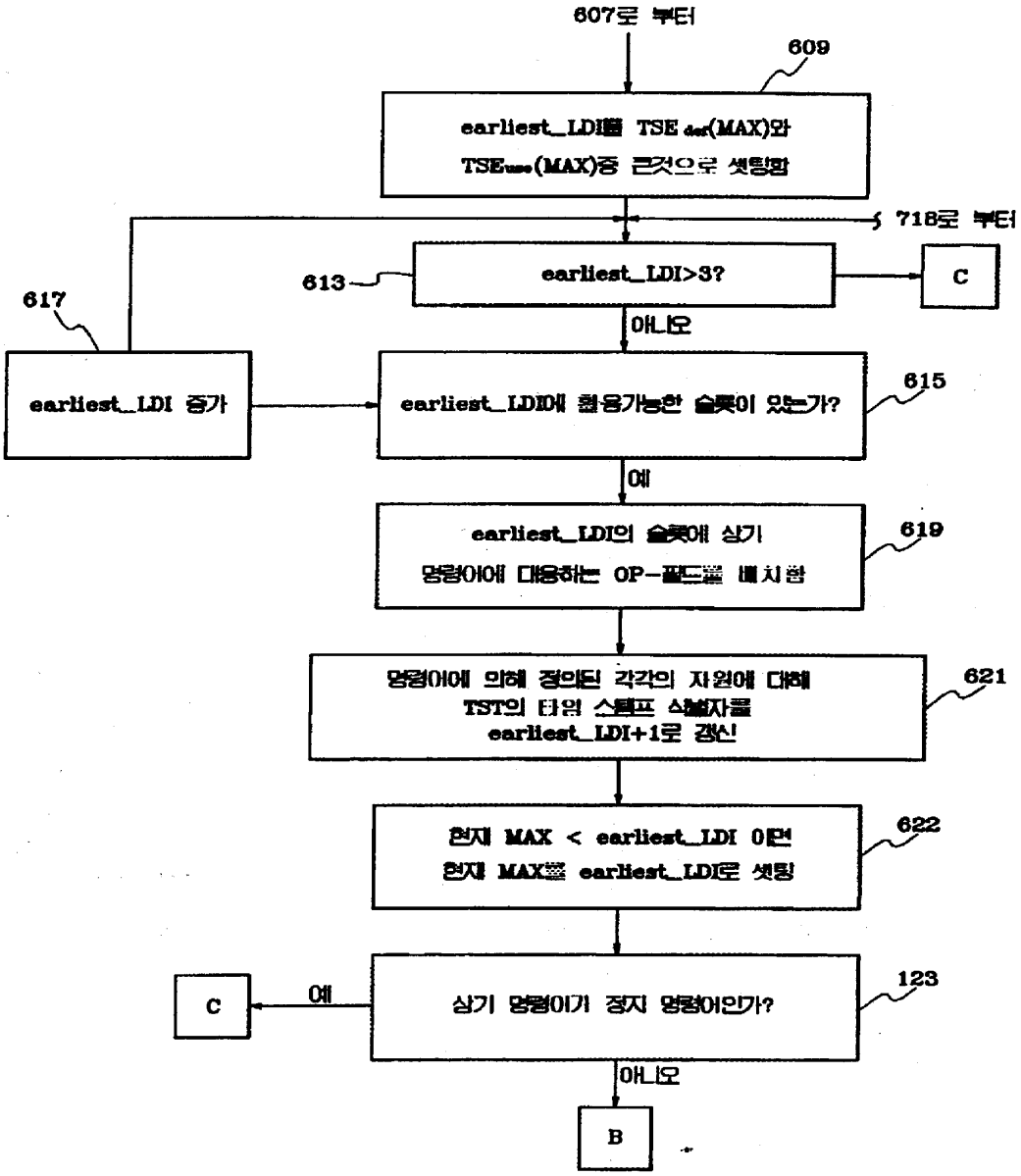
도면11


123
505
타임 스탬프 테이블
501
할당기
507
그룹 버퍼
503
OP-코드 인코더
GRE데이터를 발생한지의 DDB로
br 데이터
디스패치 유니트 (116)로 부터 명령어

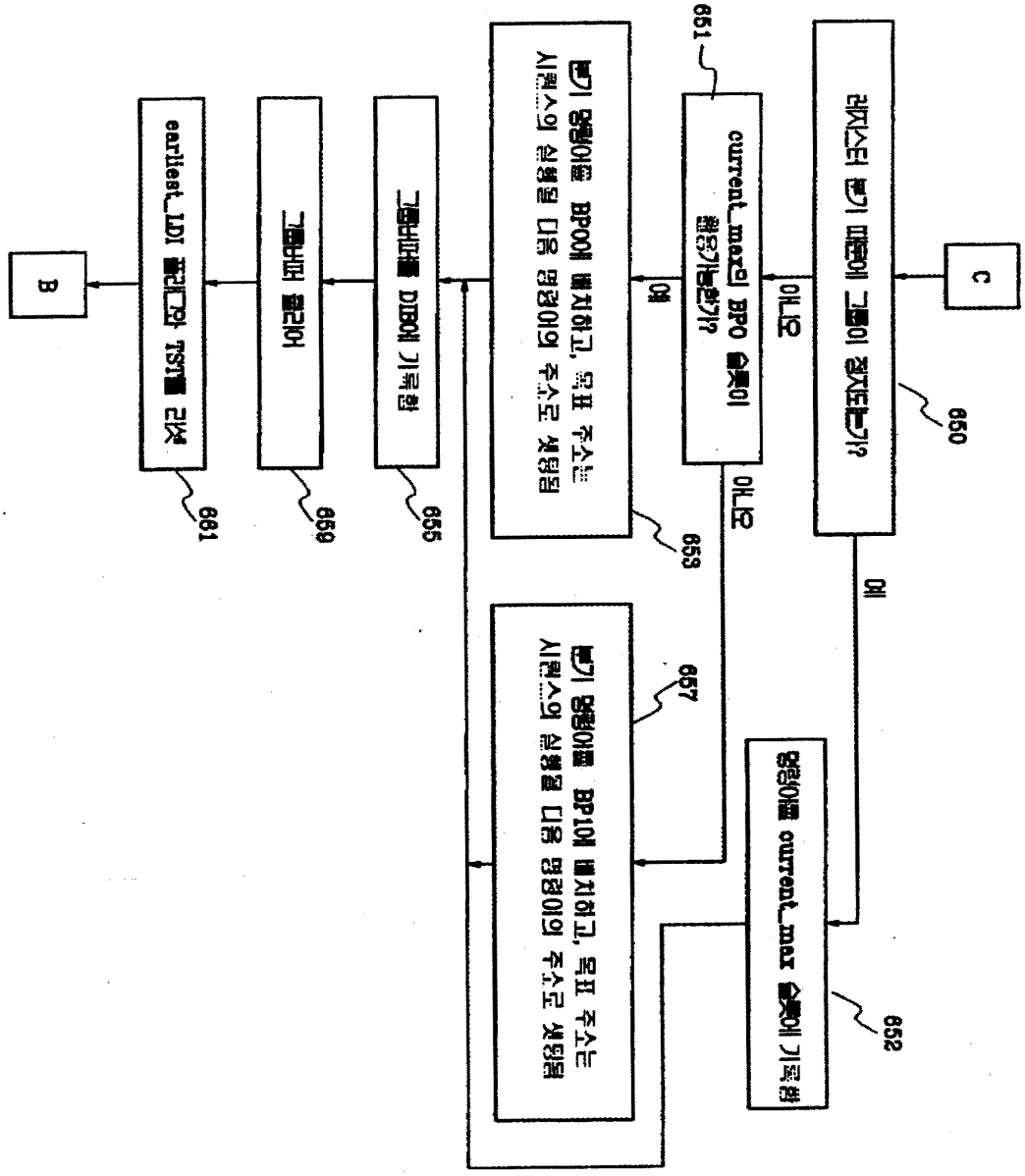
도면12a


B
대응 명령어가 조건부 복기 명령어인가?
600
아니오
이 명령어에 의해 사용되는 각각의 지원에 대해
그 지원이 최종 기록하게 되는
타임 스탬프 식별자를 TST로부터 검색
601
TSEnd(MAX)를 검색된 타임 스탬프
식별자중 최대의 것으로 정의함
603
이 명령어에 의해 사용되는 각각의 지원에 대해
그 지원이 최종 기록하게 되는
타임 스탬프 식별자를 TST로부터 검색
605
TSEnd(MAX)를 검색된 타임 스탬프
식별자중 최대의 것으로 정의함
607
809로
예
602
B

도면12b


607로 부터
609
earliest_LDI를 TSE def(MAX)와
TSEuse(MAX)중 큰것으로 셋팅함
718로 부터
earliest_LDI>3?
613
C
617
아니오
earliest_LDI 증가
earliest_LDI에 할당가능한 슬롯이 있는가?
615
예
earliest_LDI의 슬롯에 상기
명령어에 대응하는 OP-필드를 배치함
619
명령어에 의해 정의된 각각의 자원에 대해
TST의 타임 스탬프 식별자를
earliest_LDI+1로 갱신
621
현재 MAX < earliest_LDI 이면
현재 MAX를 earliest_LDI로 셋팅
622
C
예
상기 명령어가 정지 명령어인가?
123
아니오
B

도면13


C
650
레지스터 분기 명령에 그룹이 정지되는가?
아니오
예
651
current_max의 BP0 슬롯이 활성가능한가?
아니오
예
652
명령어를 current_max 슬롯에 기록함
653
분기 명령어를 BP0에 배치하고, 목표 주소는 시퀀스의 실행될 다음 명령어의 주소로 설정됨
655
그룹버퍼를 DIB0에 기록함
659
그룹버퍼 플러시
661
earliest_LDI 플러그와 TST를 리셋
B
657
분기 명령어를 BP1에 배치하고, 목표 주소는 시퀀스의 실행될 다음 명령어의 주소로 설정됨

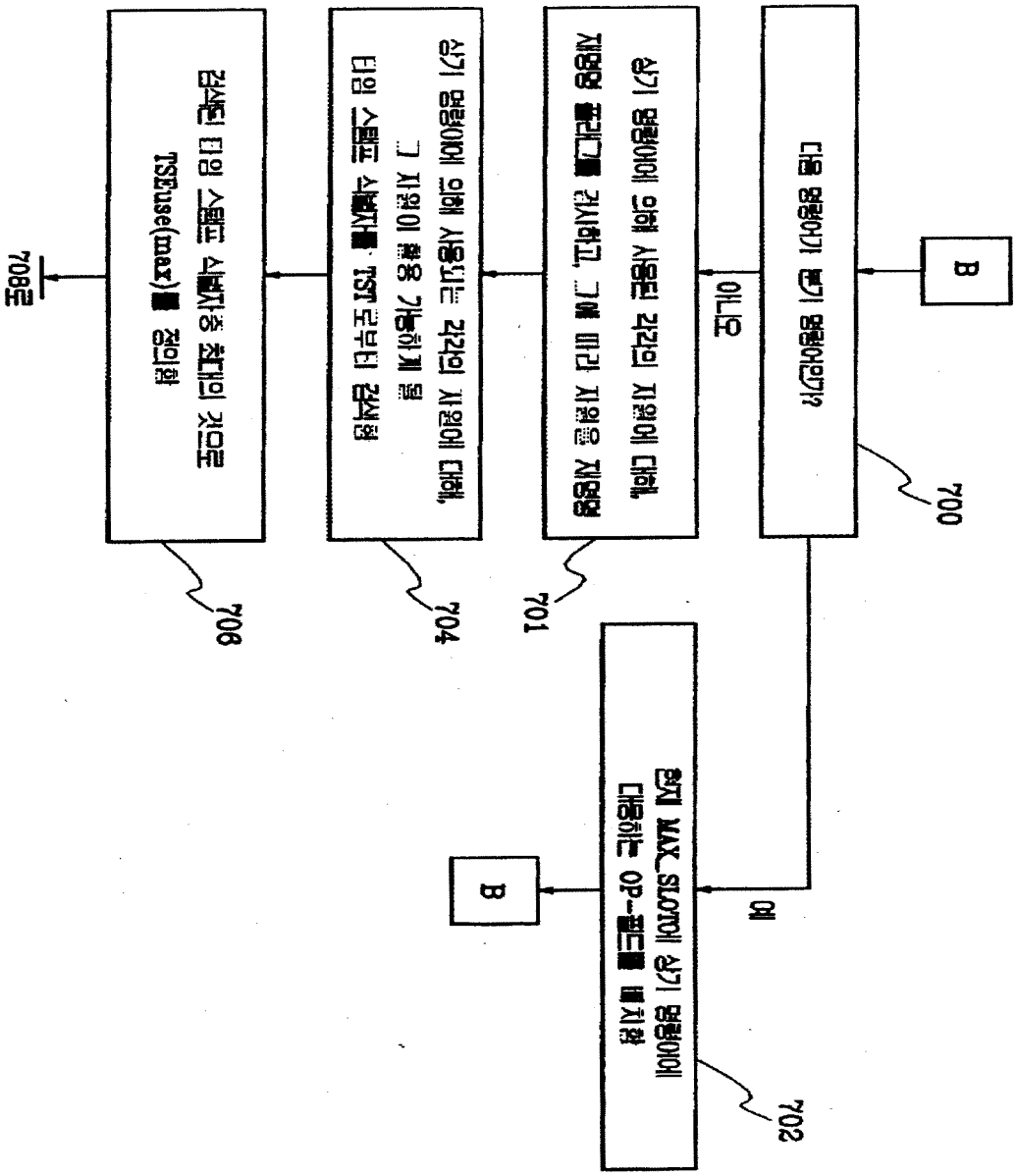
도면14a

명 령 어	사 용	정 의	TST의 섹션					
			r1	r2	r3	r4	r5	...
그룹 시작			0	0	0	0	0	...
명령어 1	r2,r3	r1	1	0	0	0	0	...
명령어 2	r1,r5	r4	1	0	0	2	0	...
명령어 3	r1,r4	r3	1	0	3	2	0	...
명령어 4	r2,r5	r4	1	0	3	3	0	...
명령어 5	r4	r4	1	0	3	4	0	...

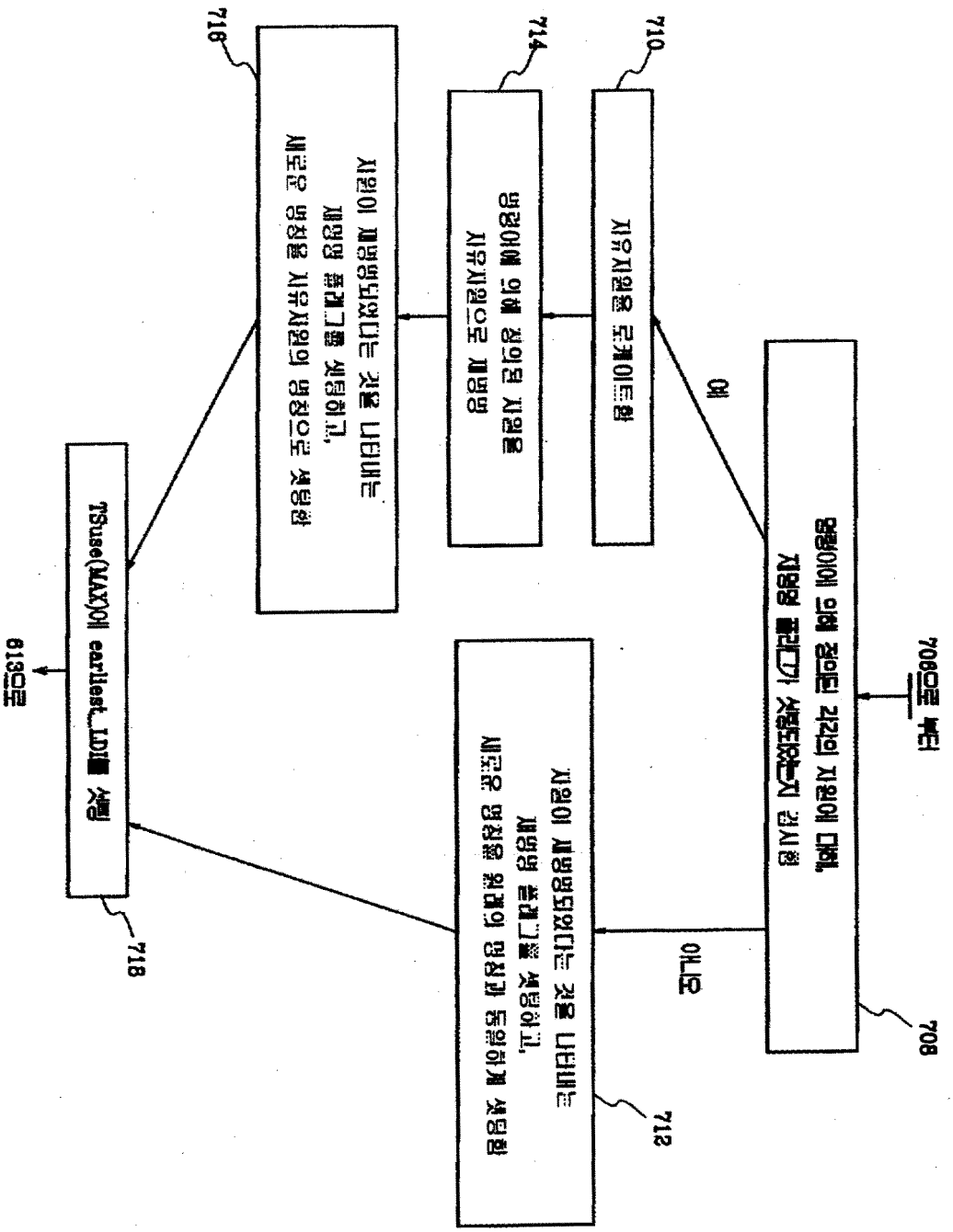
도면14b

	FU0	FU1	FU2	FU3	BP0	BP1
LDI0	명령어 1					
LDI1	명령어 2					
LDI2	명령어 3	명령어 4				
LDI3	명령어 5					

도면15a


다음 명령어가 존재 명령어인가?
700
아니오
B
상기 명령어에 의해 사용된 각각의 자원에 대해, 지원될 파라미터를 검사하고, 그에 따라 지원을 지원함
701
상기 명령어에 의해 사용되는 각각의 자원에 대해, 그 자원이 활용 가능하게 될 타임 스텝과 사용자를 TST 로부터 검색함
704
검색된 타임 스텝과 사용자중 최대의 것으로 TSTuse(max)를 정의함
706
708로
예
현재 MAX_SLOT에 상기 명령어에 대응하는 OP-코드를 배치함
702
B

도면15b


708
708으로 복귀
명령어에 의해 정의된 리간의 지원에 대해,
지원형 플래그가 세팅되었는지 검사함
예
아니오
710
지원지원을 로케이트함
712
지원이 세팅되었다는 것을 나타내는
세팅형 플래그를 세팅하고,
새로운 명령을 원래의 명령과 동일하게 세팅함
714
명령어에 의해 정의된 지원을
지원지원으로 재명명
716
지원이 재명명되었다는 것을 나타내는
세팅형 플래그를 세팅하고,
새로운 명령을 지원지원의 명령으로 세팅함
718
TSuse(MAX)에 earliest_LD를 세팅
718
813으로

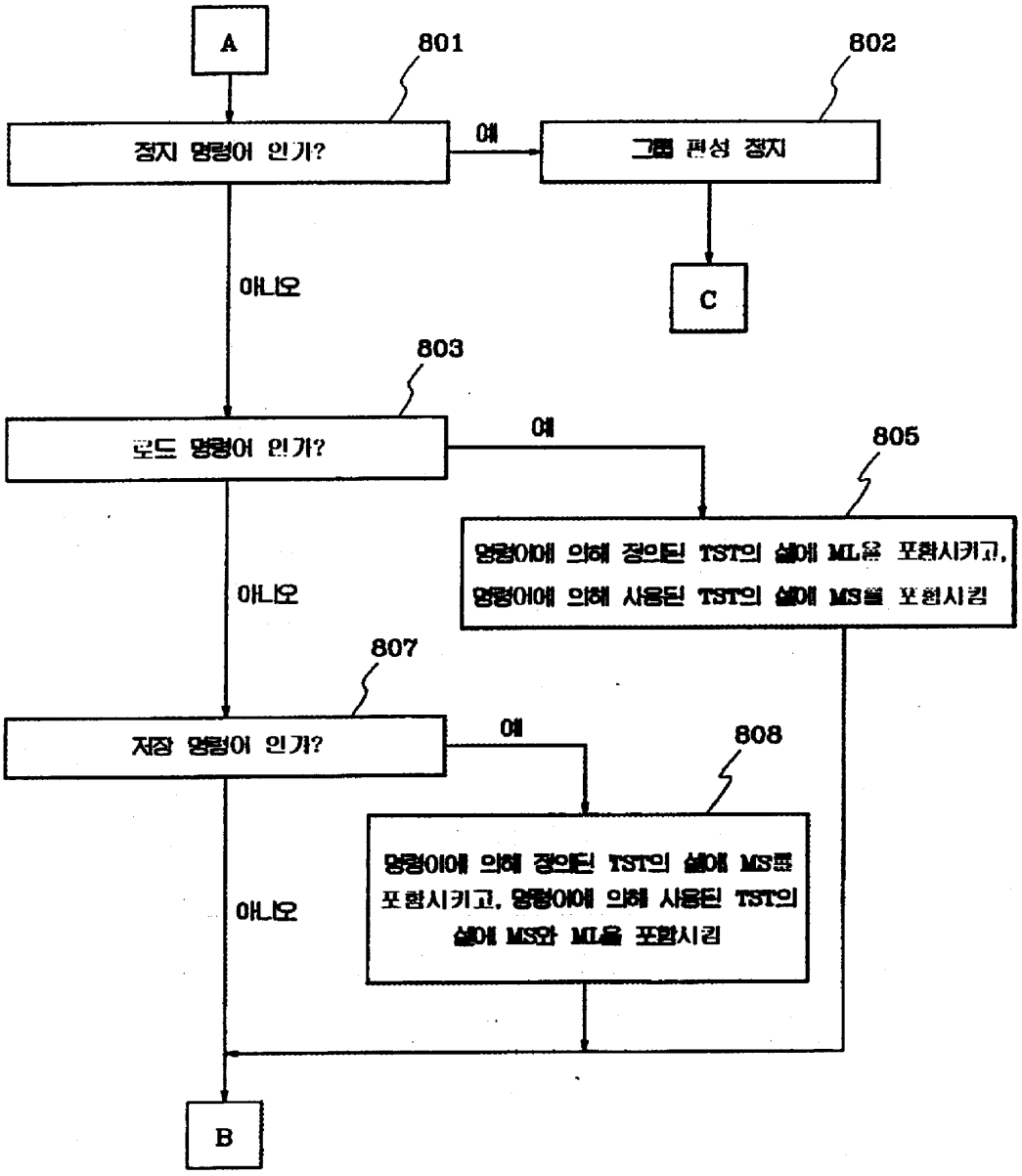
도면16a

명령어	사용	정의	TST의 섹션							
			r1	r2	r3	r4	r5	...	rN	...
그룹 시작			0	0	0	0	0	...	0	...
명령어 1	r2, r3	r1	1	0	0	0	0	...	0	...
명령어 2	r1, r5	r4	1	0	0	2	0	...	0	...
명령어 3	r1, r4	r3	1	0	3	2	0	...	0	...
명령어 4	r2, r5	r4→rN	1	0	3	2	0	...	1	...
명령어 5	r4→rN	r4→rN	1	0	3	2	0	...	2	...
명령어 6	r3, r4→rN	r3	1	0	4	2	0	...	2	...

도면16b

	FU0	FU1	FU2	FU3	BP0	BP1
LDI_0	명령어 1	명령어 4				
LDI_1	명령어 2	명령어 5				
LDI_2	명령어 3					
LDI_3	명령어 6					

도면17



도면18a

명 령 어	사 용	정 의	TST의 섹션							
			r2	r3	r6	r7	r9	r10	ML	MS
그 룹 시 작			0	0	0	0	0	0	0	0
명령어 1	r7, r10 ML, MS	MS	0	0	0	0	0	0	0	1
명령어 2	r3, r4	r2	1	0	0	0	0	0	0	1
명령어 3	r2, MS	r3, ML	1	2	0	0	0	0	2	1
명령어 4	r6, MS	r9, ML	1	2	0	0	3	0	3	1
명령어 5	r2, r9 ML, MS	MS	0	2	0	0	3	0	3	4

도면18b

	FU0	FU1	FU2	FU3	BP0	BP1
LDI_0	명령어 1	명령어 2				
LDI_1	명령어 3					
LDI_2	명령어 4					
LDI_3	명령어 5					

도면18c

명 령 어	사 용	정 의	TST의 섹션							
			r2	r3	r6	r7	r9	r10	ML	MS
그 룹 시 작			0	0	0	0	0	0	0	0
명령어 1	r7, r10 ML, MS	MS	0	0	0	0	0	0	0	1
명령어 2	r3, r4	r2	1	0	0	0	0	0	0	1
명령어 3	r2, MS	r3, ML	1	2	0	0	0	0	2	1
명령어 4	r6, MS	r9, ML	1	2	0	0	2	0	2	1
명령어 5	r2, r9, ML, MS	MS	0	2	0	0	2	0	2	3

도면18d

	FU0	FU1	FU2	FU3	BP0	BP1
LDI_0	명령어 1	명령어 2				
LDI_1	명령어 3	명령어 4				
LDI_2	명령어 5					
LDI_3						

도면19a

	FU0	FU1	FU2	FU3	BP0	BP1
LDI0	a	b	e (x=0)			
LDI1	c	f (x=0)			LK (x=1)	
LDI2	d (x=0)	g (x=0)	h (x=0) (y=1)		LP (y=0)	
LDI3	i (x=0) (y=1)				LJ (x=0) (y=1)	

비으로의 분기

	FU0	FU1	FU2	FU3	BP0	BP1
LDI0	a	b	e (x=0)	k (x=1)		
LDI1	c	f (x=0)				
LDI2	d (x=0)	g (x=0)	h (x=0) (y=1)		LP: (y=0)	
LDI3	i (x=0) (y=1)				LJ: (x=0) (y=1)	

k 이후

	FU0	FU1	FU2	FU3	BP0	BP1
LDI0	a	b	e (x=0)	k (x=1)		
LDI1	c	f (x=0)	l (x=1)			
LDI2	d (x=0)	g (x=0)	h (x=0) (y=1)		LP: (y=0)	
LDI3	i (x=0) (y=1)				LJ: (x=0) (y=1)	

L 이후

도면19b

	FU0	FU1	FU2	FU3	BP0	BP1
LDI_0	a	b	e (x=0)	k (x=1)		
LDI_1	c	f (x=0)	l (x=1)	m (x=1)		
LDI_2	d (x=0)	g (x=0)	h (x=0) (y=1)	n (x=1)	LP: (y=0)	
LDI_3	i (x=0) (y=1)				LJ (x=0) (y=1)	

n 이후

	FU0	FU1	FU2	FU3	BP0	BP1
LDI_0	a	b	e (x=0)	k (x=1)		
LDI_1	c	f (x=0)	l (x=1)	m (x=1)		
LDI_2	d (x=0)	g (x=0)	h (x=0) (y=1)	n (x=1)	LP: (y=0)	
LDI_3	i (x=0) (y=1)	o (x=0)			LJ (x=0) (y=1)	

o 이후

도면20