(19)대한민국특허청(KR)
(12) 등록특허공보(B1)

(51) 。Int. Cl.
G06F 9/22 (2006.01)
G06F 12/00 (2006.01)

(45) 공고일자 2007년07월04일
(11) 등록번호 10-0735552
(24) 등록일자 2007년06월28일

(21) 출원번호 10-2005-0088927
(22) 출원일자 2005년09월23일
심사청구일자 2005년09월23일

(65) 공개번호 10-2007-0034342
(43) 공개일자 2007년03월28일

(73) 특허권자 삼성전자주식회사
경기도 수원시 영통구 매탄동 416

(72) 발명자 임근수
경기 용인시 기흥읍 삼성종합기술원 기숙사 A동 213호

이재돈
경기 파주시 금촌2동 대영장미아파트 302동 406호

유정준
경기 용인시 기흥읍 삼성종합기술원 Computing LAB

강경호
경기 용인시 기흥읍 상갈리 금화마을주공아파트 305동 105호

박정근
서울 성동구 행당동 347 행당대림아파트 123-504

임채석
서울 동작구 상도5동 299-1 신성원룸 101호

김운기
경기 수원시 영통구 망포동 동수원엘지빌리지 105동 902호

백창우
경남 고성군 거류면 가려리 399-4

(74) 대리인 정상빈
특허법인가산

(56) 선행기술조사문헌
KR1020010021253 A
KR1020050115875 A
US6842832 B1
KR1020040067063 A
KR1020060010004 A
US 2003/0229709 A1

심사관 : 노지명

전체 청구항 수 : 총 26 항

(54) 코드 메모리 상의 프로그램의 코드 크기를 줄이는 방법

(57) 요약

본 발명은 컴퓨터 시스템에서 소프트웨어적인 방법으로 프로그램의 제어 흐름을 조정하여 프로그램의 코드 크기를 줄이는 방법에 관한 것이다.

상기 방법은, 코드 메모리에 로딩된 복수의 인스트럭션 중 오프셋과 길이를 연산수로 갖는 제1 인스트럭션의 실행 중 에러가 발생되면 상기 제1 인스트럭션의 제1 프로그램 카운트를 제1 버퍼에 저장하는 단계와, 상기 코드 메모리의 현재 프로그램 카운트를 상기 제1 프로그램 카운트에 상기 오프셋을 가산한 제2 프로그램 카운트로 변경하는 단계와, 상기 복수의 인스트럭션 중 상기 제2 프로그램 카운트 위치에서 상기 길이 만큼 이후에 위치하는 제2 인스트럭션을 제2 버퍼에 저장하는 단계와, 상기 제2 인스트럭션을 마이크로 프로세서가 인식하지 못하는 제3 인스트럭션으로 치환하는 단계와, 상기 제3 인스트럭션 실행 중 에러가 발생되면 상기 제3 인스트럭션을 상기 제2 버퍼에 저장된 상기 제2 인스트럭션으로 환원하는 단계와, 상기 코드 메모리의 현재 프로그램 카운트를 상기 제1 버퍼에 저장된 상기 제1 프로그램 카운트의 다음 카운트로 변경하는 단계를 포함한다.

대표도

도 4

특허청구의 범위

청구항 1.

(a) 코드 메모리에 로딩된 복수의 인스트럭션 중 현재 프로그램 카운트와 처리될 첫 인스트럭션의 프로그램 카운트까지의 거리를 나타내는 오프셋과, 처리될 인스트럭션의 수를 나타내는 길이를 연산수로 갖는 제1 인스트럭션의 실행 중 에러가 발생되면 상기 제1 인스트럭션의 제1 프로그램 카운트를 제1 버퍼에 저장하는 단계;

(b) 상기 코드 메모리의 현재 프로그램 카운트를 상기 제1 프로그램 카운트에 상기 오프셋을 가산한 제2 프로그램 카운트로 변경하는 단계;

(c) 상기 복수의 인스트럭션 중 상기 제2 프로그램 카운트 위치에서 상기 길이 만큼 이후에 위치하는 제2 인스트럭션을 제2 버퍼에 저장하는 단계;

(d) 상기 제2 인스트럭션을 마이크로 프로세서가 인식하지 못하는 제3 인스트럭션으로 치환하는 단계;

(e) 상기 제3 인스트럭션 실행 중 에러가 발생되면 상기 제3 인스트럭션을 상기 제2 버퍼에 저장된 상기 제2 인스트럭션으로 환원하는 단계; 및

(f) 상기 코드 메모리의 현재 프로그램 카운트를 상기 제1 버퍼에 저장된 상기 제1 프로그램 카운트의 다음 카운트로 변경하는 단계를 포함하는, 코드 메모리 상의 프로그램의 코드 크기를 줄이는 방법.

청구항 2.

제1항에 있어서,

상기 제1 인스트럭션은 Offset 및 Length를 연산수로 갖는 Echo 인스트럭션인, 코드 메모리 상의 프로그램의 코드 크기를 줄이는 방법.

청구항 3.

제1항에 있어서,

상기 제1 프로그램 카운트의 다음 카운트는 상기 제1 프로그램 카운트에 비하여 4만큼 큰, 코드 메모리 상의 프로그램의 코드 크기를 줄이는 방법.

청구항 4.

제1항에 있어서,

상기 (a) 내지 (f) 단계들은 운영 체제에 의하여 실행되는, 코드 메모리 상의 프로그램의 코드 크기를 줄이는 방법.

청구항 5.

제4항에 있어서, 상기 (d) 단계 및 상기 (e) 단계 사이에,

상기 운영 체제가 리턴 명령에 의하여 상기 마이크로 프로세서에게 인스트럭션 실행권을 넘겨 주는 단계를 더 포함하는, 코드 메모리 상의 프로그램의 코드 크기를 줄이는 방법.

청구항 6.

(a) 코드 메모리에 로딩된 복수의 인스트럭션 중 현재 프로그램 카운트와 처리될 첫 인스트럭션의 프로그램 카운트까지의 거리를 나타내는 오프셋과, 처리될 인스트럭션의 수를 나타내는 길이를 연산수로 가지는 제1 인스트럭션 및 시스템 함수로 분기할 것을 명하는 제2 인스트럭션을 실행하는 단계;

(b) 상기 제2 인스트럭션의 제1 프로그램 카운트를 제1 버퍼에 저장하는 단계;

(c) 상기 제1 프로그램 카운트에 상기 오프셋을 가산한 값을 제2 프로그램 카운트라 할 때, 상기 제2 프로그램 카운트 위치에서 상기 길이 만큼 이후에 위치하는 제3 인스트럭션을 제2 버퍼에 저장하는 단계;

(d) 상기 코드 메모리 상에 저장되어 있는 제3 인스트럭션을, 시스템 함수로 분기할 것을 명하는 제4 인스트럭션으로 치환하는 단계;

(e) 상기 코드 메모리의 현재 프로그램 카운트를 상기 제2 프로그램 카운트로 변경하는 단계;

(f) 상기 제4 인스트럭션이 실행되면, 상기 제4 인스트럭션을 상기 제2 버퍼에 저장된 상기 제3 인스트럭션으로 환원하는 단계; 및

(g) 상기 코드 메모리의 현재 프로그램 카운트를 상기 제1 버퍼에 저장된 상기 제1 프로그램 카운트의 다음 카운트로 변경하는 단계를 포함하는, 코드 메모리 상의 프로그램의 코드 크기를 줄이는 방법.

청구항 7.

제6항에 있어서,

상기 (b) 내지 (g) 단계는 시스템 함수에 의하여 실행되는, 코드 메모리 상의 프로그램의 코드 크기를 줄이는 방법.

청구항 8.

제6항에 있어서,

상기 제1 인스트럭션은 Offset 및 Length를 연산수로 갖는 Echo 인스트럭션인, 코드 메모리 상의 프로그램의 코드 크기를 줄이는 방법.

청구항 9.

(a) 코드 메모리에 로딩된 복수의 인스트럭션 중 현재 프로그램 카운트와 처리될 첫 인스트럭션의 프로그램 카운트까지의 거리를 나타내는 오프셋과, 처리될 인스트럭션의 수를 나타내는 길이를 연산수로 갖는 제1 인스트럭션의 실행 중 에러가 발생되면, 상기 오프셋이 나타내는 위치에서부터 상기 길이 만큼의 인스트럭션들을 임시 버퍼에 복사하는 단계;

(b) 상기 임시 버퍼 중 상기 복사된 인스트럭션들의 다음 위치에, 상기 제1 인스트럭션의 다음 인스트럭션으로 복귀할 것을 명하는 제2 인스트럭션을 삽입하는 단계;

(c) 상기 복사된 인스트럭션들을 실행하는 단계; 및

(d) 상기 제2 인스트럭션의 실행에 따라서 상기 코드 메모리 중 상기 제1 인스트럭션 다음의 인스트럭션으로 복귀하는 단계를 포함하는, 코드 메모리 상의 프로그램의 코드 크기를 줄이는 방법.

청구항 10.

제9항에 있어서, 상기 코드 메모리는

읽기 전용 메모리(ROM)에 미리 기록된 인스트럭션들의 집합으로 구성되는, 코드 메모리 상의 프로그램의 코드 크기를 줄이는 방법.

청구항 11.

제10항에 있어서, 상기 임시 버퍼는

비휘발성 RAM에 의하여 구현되는, 코드 메모리 상의 프로그램의 코드 크기를 줄이는 방법.

청구항 12.

제9항에 있어서,

상기 제1 인스트럭션은 Offset 및 Length를 연산수로 갖는 Echo 인스트럭션인, 코드 메모리 상의 프로그램의 코드 크기를 줄이는 방법.

청구항 13.

(a) 코드 메모리에 로딩된 복수의 인스트럭션 중 현재 프로그램 카운트와 처리될 첫 인스트럭션의 프로그램 카운트까지의 거리를 나타내는 오프셋과, 처리될 인스트럭션의 수를 나타내는 비트마스크를 연산수로 갖는 제1 인스트럭션의 실행 중 에러가 발생되면 상기 제1 인스트럭션의 제1 프로그램 카운트를 제1 버퍼에 저장하는 단계;

(b) 상기 코드 메모리의 상기 제1 프로그램 카운트에 상기 오프셋을 가산한 값을 제2 프로그램 카운트라 할 때, 상기 제2 프로그램 카운트 위치에서부터 존재하는 인스트럭션 세트 중에서, 상기 비트마스크에 의하여 실행할 필요가 없는 것으로 표시되는 인스트럭션을 제2 버퍼에 저장한 후, 실행할 필요가 없는 것으로 표시되는 인스트럭션을 no operation 인스트럭션으로 치환하는 단계;

(c) 상기 복수의 인스트럭션 중 상기 제2 프로그램 카운트 위치에서 상기 비트마스크의 크기 만큼 이후에 위치하는 제2 인스트럭션을 제2 버퍼에 저장한 후, 상기 제2 인스트럭션을 마이크로 프로세서가 인식하지 못하는 제3 인스트럭션으로 치환하는 단계;

(d) 상기 코드 메모리의 현재 프로그램 카운트를 상기 제2 프로그램 카운트로 변경하는 단계;

(e) 상기 제3 인스트럭션 실행 중 에러가 발생되면 상기 no operation 인스트럭션을 상기 제2 버퍼에 저장된 상기 인스트럭션으로 환원하고, 상기 제3 인스트럭션을 상기 제2 인스트럭션으로 환원하는 단계; 및

(f) 상기 코드 메모리의 현재 프로그램 카운트를 상기 제1 버퍼에 저장된 상기 제1 프로그램 카운트의 다음 카운트로 변경하는 단계를 포함하는, 코드 메모리 상의 프로그램의 코드 크기를 줄이는 방법.

청구항 14.

제13항에 있어서,

상기 비트마스크는 상기 크기 만큼의 1비트의 숫자열로 이루어지는, 코드 메모리 상의 프로그램의 코드 크기를 줄이는 방법.

청구항 15.

제14항에 있어서,

상기 숫자열을 이루는 각각의 1비트 숫자는 대응되는 인스트럭션의 중복 여부를 역순으로 표시하는, 코드 메모리 상의 프로그램의 코드 크기를 줄이는 방법.

청구항 16.

제13항에 있어서,

상기 제1 인스트럭션은 Offset 및 Bitmask를 연산수로 갖는 비트마스크 Echo 인스트럭션인, 코드 메모리 상의 프로그램의 코드 크기를 줄이는 방법.

청구항 17.

제13항에 있어서,

상기 (a) 내지 (f) 단계들은 운영 체제에 의하여 실행되는, 코드 메모리 상의 프로그램의 코드 크기를 줄이는 방법.

청구항 18.

(a) 코드 메모리에 로딩된 복수의 인스트럭션 중 현재 프로그램 카운트와 처리될 첫 인스트럭션의 프로그램 카운트까지의 거리를 나타내는 오프셋과, 처리될 인스트럭션의 수를 나타내는 비트마스크를 연산수로 가지는 제1 인스트럭션 및 시스템 함수로 분기할 것을 명하는 제2 인스트럭션을 실행하는 단계;

(b) 상기 제2 인스트럭션의 제1 프로그램 카운트를 제1 버퍼에 저장하는 단계;

(c) 상기 코드 메모리의 상기 제1 프로그램 카운트에 상기 오프셋을 가산한 값을 제2 프로그램 카운트라 할 때, 상기 제2 프로그램 카운트 위치에서부터 존재하는 인스트럭션 세트 중에서, 상기 비트마스크에 의하여 실행할 필요가 없는 것으로 표시되는 인스트럭션을 제2 버퍼에 저장한 후, 상기 실행할 필요가 없는 인스트럭션을 no operation 인스트럭션으로 치환하는 단계;

(d) 상기 복수의 인스트럭션 중 상기 제2 프로그램 카운트 위치에서 상기 비트마스크의 크기 만큼 이후에 위치하는 제3 인스트럭션을 제3 버퍼에 저장한 후, 상기 코드 메모리 상에 저장되어 있는 상기 제3 인스트럭션을, 시스템 함수로 분기할 것을 명하는 제4 인스트럭션으로 치환하는 단계;

(e) 상기 코드 메모리의 현재 프로그램 카운트를 상기 제2 프로그램 카운트로 변경하는 단계;

(f) 상기 제4 인스트럭션 실행이 실행되면, 상기 no operation 인스트럭션을 상기 제2 버퍼에 저장된 상기 인스트럭션으로 환원하고, 상기 제4 인스트럭션을 상기 제3 인스트럭션으로 환원하는 단계; 및

(g) 상기 코드 메모리의 현재 프로그램 카운트를 상기 제1 버퍼에 저장된 상기 제1 프로그램 카운트의 다음 카운트로 변경하는 단계를 포함하는, 코드 메모리 상의 프로그램의 코드 크기를 줄이는 방법.

청구항 19.

제18항에 있어서,

상기 비트마스크는 상기 크기 만큼의 1비트의 숫자열로 이루어지는, 코드 메모리 상의 프로그램의 코드 크기를 줄이는 방법.

청구항 20.

제18항에 있어서,

상기 제1 인스트럭션은 Offset 및 Bitmask를 연산수로 갖는 비트마스크 Echo 인스트럭션인, 코드 메모리 상의 프로그램의 코드 크기를 줄이는 방법.

청구항 21.

제18항에 있어서,

상기 (a) 내지 (g) 단계들은 시스템 함수에 의하여 실행되는, 코드 메모리 상의 프로그램의 코드 크기를 줄이는 방법.

청구항 22.

(a) 코드 메모리에 로딩된 복수의 인스트럭션 중 현재 프로그램 카운트와 처리될 첫 인스트럭션의 프로그램 카운트까지의 거리를 나타내는 오프셋과, 해당 인스트럭션의 실행 여부를 나타내는 비트의 열로 구성되는 비트마스크를 연산수로 갖는 제1 인스트럭션의 실행 중 에러가 발생되면, 상기 제1 인스트럭션과 별도로 상기 코드 메모리에 로딩된 인스트럭션 세트 중에서, 상기 제1 비트마스크에 의하여 실행할 필요가 있는 것으로 표시되는 인스트럭션들을 임시 버퍼에 복사하는 단계;

(b) 상기 임시 버퍼 중 상기 복사된 인스트럭션들의 다음 위치에, 상기 제1 인스트럭션의 다음 인스트럭션으로 복귀할 것을 명하는 제2 인스트럭션을 삽입하는 단계;

(c) 상기 복사된 인스트럭션들을 실행하는 단계; 및

(d) 상기 제2 인스트럭션의 실행에 따라서 상기 코드 메모리 중 상기 제1 인스트럭션 다음의 인스트럭션으로 복귀하는 단계를 포함하는, 코드 메모리 상의 프로그램의 코드 크기를 줄이는 방법.

청구항 23.

제22항에 있어서,

상기 비트마스크는 1비트의 숫자열로 이루어지는, 코드 메모리 상의 프로그램의 코드 크기를 줄이는 방법.

청구항 24.

제22항에 있어서, 상기 코드 메모리는

읽기 전용 메모리(ROM)에 미리 기록된 인스트럭션들의 집합으로 구성되는, 코드 메모리 상의 프로그램의 코드 크기를 줄이는 방법.

청구항 25.

제22항에 있어서, 상기 임시 버퍼는

비휘발성 RAM에 의하여 구현되는, 코드 메모리 상의 프로그램의 코드 크기를 줄이는 방법.

청구항 26.

제22항에 있어서,

상기 제1 인스트럭션은 Offset 및 Bitmask를 연산수로 갖는 비트마스크 Echo 인스트럭션인, 코드 메모리 상의 프로그램의 코드 크기를 줄이는 방법.

명세서

발명의 상세한 설명

발명의 목적

발명이 속하는 기술 및 그 분야의 종래기술

본 발명은 컴퓨터 시스템에 관한 것으로, 특히 마이크로 프로세서와 메모리로 구성된 일반적인 컴퓨터시스템에서 소프트웨어적인 방법으로 프로그램의 제어 흐름을 조정하여 프로그램의 코드 크기를 줄이는 방법에 관한 것이다.

냉장고, 청소기, 프린터, 플래시메모리 카드와 같은 소형 사무용, 가정용, 휴대용 전자제품의 경우 내부의 모터 등과 같은 장치를 실시간으로 제어하기 위하여 실시간 운영체제와 관련 미들웨어를 탑재하고 있다. 이들 전자제품은 연간 수 백만 대에서부터 수 십억 대까지 대량 생산되고 있기 때문에 생산 단가를 낮추는 것이 매우 중요한 이슈 중 하나이다. 전자제품 내부의 실시간 제어를 담당하는 컴퓨터시스템은 프로세서와 메모리 그리고 주변 입출력 장치로 구성되는데, 일반적으로 이 중에서 메모리 장치에 높은 생산 비용이 소요된다. 따라서 SRAM, DRAM, 등의 메모리 장치의 비용을 줄이기 위해서는 프로그램의 크기를 줄일 필요성이 있는 것이다.

프로그램의 크기를 줄이면 메모리 장치에 소요되는 생산 비용이 절감되는데 이러한 이득은 특히 372KB, 1MB, 2MB 등과 같은 2의 배수 형태로 양산되는 메모리 칩의 경계를 넘어설 때 실질적으로 나타난다. 예를 들어 프로그램의 크기가 1.2MB 여서 2MB의 메모리 칩을 사용해야 하는 경우에 프로그램의 크기를 1MB 이하로 줄일 수 있다면 1MB 크기의 메모리 칩을 사용해서 시스템을 제작할 수 있다. 이를 통해서 메모리 칩의 크기가 줄어들게 되면 메모리 장치에 소요되는 전력량을 줄일 수 있다. 또한, 메모리 칩의 크기가 줄지 않아도 프로그램의 크기가 줄어들면 메모리 장치를 저전력으로 동작시키는 방식을 함께 사용하여 전력 소모를 줄일 수 있다. 이러한 저전력 동작 특성은 모바일 전자 제품에 적용될 때 특히 유용한 장점으로 작용한다.

이러한 점을 고려하여, 다양한 코드 크기를 최소화하는 다양한 종래기술들이 제시되었다. 여기에는, 크게 프로그램을 직접 실행이 불가능한 형태로 구성하는 압축 기법과, 실행 가능한 형태로 구성하는 축약(Compaction) 기법이 있다. 압축 기법은 다시 하드웨어 압축기 및 복원기를 사용하는 하드웨어 압축 기법과 CPU를 사용해 소프트웨어로 압축 및 복원을 실행하는 소프트웨어 압축 기법이 있다.

하드웨어 압축 기법에도 다양한 기법이 존재하지만 대표적인 방식은 주 메모리에 내장 캐시의 라인 단위로 데이터를 압축해 저장해 두고 추후에 해당 압축 캐시 라인이 억세스가 될 경우 하드웨어 복원기를 사용하여 압축을 복원함으로써, 내장 캐시에 압축이 풀린 상태로 저장하는 방식이다. 이 방식은 복원에 따른 시간 상의 추가 비용이 비교적 적다는 장점이 있기는 하지만 하드웨어 복원기를 사용함에 따라 생산 비용의 증가가 불가피하기 때문에, 전자제품의 내부에 탑재된 실시간 제어를 위한 컴퓨터시스템에는 적합하지 않을 수 있다.

소프트웨어 압축 기법은 주 메모리를 압축된 영역과 그렇지 않은 영역(비압축 영역)으로 분할하고 이를 운영체제를 사용해 관리한다. 세부적으로 특정 주소가 필요하거나 앞으로 사용될 확률이 높은 데이터는 이를 운영체제의 메모리 관리자가 직접 복원하여 비압축 영역에 저장한다. 이때, 비압축 영역이 부족하면 이 영역에서 앞으로 사용될 가능성이 낮은 부분을 압축하여 압축 영역에 저장한다. 이 방식은 압축기와 같은 부가적인 하드웨어를 필요로 하지 않는 장점은 있지만, 소프트웨어적인 압축 및 복원에 따라 많은 시간이 소요되며, 기존의 운영체제를 수정해야 한다는 단점이 있다. 결국, 소프트웨어 압축 기법 및 하드웨어 압축 기법 모두 전자제품 등에 내장된 컴퓨터시스템에는 적합하지 않다고 볼 수 있다.

다음으로 축약 기법은, 명령어 집합 구조(ISA; Instruction Set Architecture)를 변환하는 방법과 명령어를 사용해 작성한 프로그램을 변경하는 방법으로 나뉜다. ISA를 변환하는 방법은 ARM 社의 THUMB(16Bit)과 ARM(14Bit) 명령어 집합과 MIPS 社의 MIPS16(16Bit)과 MIPS14(14Bit) 명령어 집합처럼 크기가 다른 명령어 집합을 정의하고 이를 상황에 따라 사용하는 것이다. 하지만 이를 위해서는 ARM 또는 MIPS와 같이 비교적 계산능력이 큰 프로세서를 사용해야 하므로 이는 응용에 따라서는 과도한 하드웨어 설계 사양일 수 있다.

다음으로 프로그램을 재구성하는 방법은 최근 5년 사이에 다양하게 연구된 분야로서, 기본적으로 프로그램 내부에서 같거나 유사한 명령어들의 배열을 찾아내서 이를 하나로 구성하고, 해당 명령어 배열이 존재하는 모든 부분에서 새로 구성된 영역으로 분기하는 방식이다. 이 방법은 세부적으로 함수 추상화(PA; procedural abstraction)과 상호 분기(CJ; cross jumping or tail merging) 기법에 바탕을 두고 있다. PA는 일치하는 명령어 배열이 N개 존재하는 경우 이를 하나의 함수로 분리해 내고 N개의 명령어 배열이 있는 장소에 이 분리해 낸 함수 호출 명령어를 대치한다. 그리고 PA의 특수한 경우로 일치하는 명령어들의 끝 부분이 모두 같은 부분으로 분기하는 경우 일치하는 명령어들을 분기하는 지점의 시작 위치로 이동시킨다.

실제 프로그램의 경우 명령어의 순서가 바뀌어 있거나 레지스터의 이름 등이 부분적으로 다른 경우가 많이 존재한다. 따라서 명령어 재배치(instruction reordering)와 레지스터 재할당(register renaming) 등의 기법을 활용하여 유사한 명령어 배열을 동일한 배열로 치환하는 기법이 많이 사용된다. 공개된 실험 결과에 따르면 컴파일 시점에 기법을 적용하면 평균적

으로 5% 정도 코드 크기를 줄일 수 있으며, 링킹(linking) 이후에 바이너리 파일에 적용하면 평균적으로 30% 정도 코드 크기를 줄일 수 있다. 바이너리 파일의 경우 프로그램의 제어 흐름(control flow)과 관련한 모든 정보를 가지고 있기 때문에 보다 효율적으로 코드 크기를 최적화할 수 있다는 것이다.

이 방식의 경우 반복된 명령어 배열을 함수로 분리하기 때문에 함수 호출 및 실행을 위하여 추가적으로 명령어들을 사용하게 된다. 세부적으로 함수 호출시에 현재의 리턴 주소를 스택에 저장해야 하고 호출된 함수에서 레지스터들을 스택에 저장해 보호해야 할 수 있으며 함수의 실행이 종료되면 다시 원래 위치로 복귀해야 한다. 따라서 이러한 추가 명령어의 사용을 줄이기 위하여 "Sequential Echo <offset>,<length>" 명령어가 고안되었다.

이 명령어는 <offset>위치에서부터 <length>개의 명령어를 실행하라는 의미로, 이 명령어 하나만 사용하여 해당 부분의 명령어를 실행하고 원 위치로 복귀할 수 있다. 이는 프로세서의 ISA를 수정해 추가하거나 자바(Java)와 같은 인터프리터 또는 가상기계를 수정해 구현될 수 있다. 나아가서 "Bitmask Echo <offset>,<bitmask>"는 <offset>부터 존재하는 명령어들의 <bitmask>를 보고 마스크 상의 해당 번째 비트가 1로 설정이 되어 있는 경우에만 해당 명령어들을 실행한다. 따라서, 명령어들 중간에 부분적으로 일치하지 않는 명령어가 있는 경우에도 코드를 공유함으로써 프로그램의 크기를 보다 많이 줄일 수 있다. 성능 평가 결과에 따르면 이 방법을 기존의 전통적인 코드 최적화 기법과 병용하면 기존의 코드 축약 방식에 비하여 코드 크기를 2배 가량 더 줄일 수 있다고 한다.

이상의 Echo 명령어들을 사용하는 방법에 의하여 프로그램의 크기를 크기 줄일 수 있는 장점이 있는 반면에, 실제 사용을 위해서는 마이크로 프로세서의 ISA를 변경해야 하는 단점이 있다. 이는 하드웨어 변경을 요구하게 되며 가전제품 등에 탑재된 소형 컴퓨터 시스템에서는 개발 비용의 부담으로 이어질 수 있다. 따라서, 하드웨어의 변경 없이도 Sequential Echo와 Bitmask Echo 명령어들을 지원하는 기법을 고안할 필요가 있다.

발명이 이루고자 하는 기술적 과제

본 발명이 이루고자 하는 기술적 과제는 컴퓨터 시스템에서 소프트웨어적인 방법으로 프로그램의 제어 흐름을 조정하여 프로그램의 코드 크기를 줄일 수 있는 방법을 제공하는 것이다.

본 발명의 기술적 과제들은 이상에서 언급한 기술적 과제로 제한되지 않으며, 언급되지 않은 또 다른 기술적 과제들은 아래의 기재로부터 당업자에게 명확하게 이해될 수 있을 것이다.

발명의 구성

상기한 기술적 과제를 달성하기 위하여, 본 발명의 일 실시예에 따른 코드 메모리 상의 프로그램의 코드 크기를 줄이는 방법은, (a) 코드 메모리에 로딩된 복수의 인스트럭션 중 오프셋과 길이를 연산수로 갖는 제1 인스트럭션의 실행 중 에러가 발생되면 상기 제1 인스트럭션의 제1 프로그램 카운트를 제1 버퍼에 저장하는 단계; (b) 상기 코드 메모리의 현재 프로그램 카운트를 상기 제1 프로그램 카운트에 상기 오프셋을 가산한 제2 프로그램 카운트로 변경하는 단계; (c) 상기 복수의 인스트럭션 중 상기 제2 프로그램 카운트 위치에서 상기 길이 만큼 이후에 위치하는 제2 인스트럭션을 제2 버퍼에 저장하는 단계; (d) 상기 제2 인스트럭션을 마이크로 프로세서가 인식하지 못하는 제3 인스트럭션으로 치환하는 단계; (e) 상기 제3 인스트럭션 실행 중 에러가 발생되면 상기 제3 인스트럭션을 상기 제2 버퍼에 저장된 상기 제2 인스트럭션으로 환원하는 단계; 및 (f) 상기 코드 메모리의 현재 프로그램 카운트를 상기 제1 버퍼에 저장된 상기 제1 프로그램 카운트의 다음 카운트로 변경하는 단계를 포함한다.

상기한 기술적 과제를 달성하기 위하여, 본 발명의 다른 실시예에 따른 코드 메모리 상의 프로그램의 코드 크기를 줄이는 방법은, (a) 코드 메모리에 로딩된 복수의 인스트럭션 중 오프셋과 길이를 연산수로 갖는 제1 인스트럭션 및 시스템 함수로 분기할 것을 명하는 제2 인스트럭션을 실행하는 단계; (b) 상기 제2 인스트럭션의 제1 프로그램 카운트를 제1 버퍼에 저장하는 단계; (c) 상기 코드 메모리의 현재 프로그램 카운트를 상기 제1 프로그램 카운트에 상기 오프셋을 가산한 제2 프로그램 카운트로 변경하는 단계; (d) 상기 복수의 인스트럭션 중 상기 제2 프로그램 카운트 위치에서 상기 길이 만큼 이후에 위치하는 제3 인스트럭션을 제2 버퍼에 저장하는 단계; (e) 상기 제3 인스트럭션을, 시스템 함수로 분기할 것을 명하는 제4 인스트럭션으로 치환하는 단계; (f) 상기 제4 인스트럭션 실행이 실행되면, 상기 제4 인스트럭션을 상기 제2 버퍼에 저장된 상기 제3 인스트럭션으로 환원하는 단계; 및 (g) 상기 코드 메모리의 현재 프로그램 카운트를 상기 제1 버퍼에 저장된 상기 제1 프로그램 카운트의 다음 카운트로 변경하는 단계를 포함한다.

상기한 기술적 과제를 달성하기 위하여, 본 발명의 또 다른 실시예에 따른 코드 메모리 상의 프로그램의 코드 크기를 줄이는 방법은, (a) 코드 메모리에 로딩된 복수의 인스트럭션 중 오프셋과 길이를 연산수로 갖는 제1 인스트럭션의 실행 중 에

러가 발생되면, 상기 오프셋이 나타내는 위치에서부터 상기 길이 만큼의 인스트럭션들을 임시 버퍼에 복사하는 단계; (b) 상기 임시 버퍼 중 상기 복사된 인스트럭션들의 다음 위치에, 상기 제1 인스트럭션의 다음 인스트럭션으로 복귀할 것을 명하는 제2 인스트럭션을 삽입하는 단계; (c) 상기 복사된 인스트럭션들을 실행하는 단계; 및 (d) 상기 제2 인스트럭션의 실행에 따라서 상기 코드 메모리 중 상기 제1 인스트럭션 다음의 인스트럭션으로 복귀하는 단계를 포함한다.

기타 실시예들의 구체적인 사항들은 상세한 설명 및 도면들에 포함되어 있다.

본 발명의 이점 및 특징, 그리고 그것들을 달성하는 방법은 첨부되는 도면과 함께 상세하게 후술되어 있는 실시예들을 참조하면 명확해질 것이다. 그러나 본 발명은 이하에서 개시되는 실시예들에 한정되는 것이 아니라 서로 다른 다양한 형태로 구현될 것이며, 단지 본 실시예들은 본 발명의 개시가 완전하도록 하며, 본 발명이 속하는 기술분야에서 통상의 지식을 가진 자에게 발명의 범주를 완전하게 알려주기 위해 제공되는 것이며, 본 발명은 청구항의 범주에 의해 정의될 뿐이다. 명세서 전체에 걸쳐 동일 참조 부호는 동일 구성 요소를 지칭한다.

이하 첨부된 도면들을 참조하여 본 발명의 실시예를 상세히 설명한다.

도 1을 참조하여 보면, 본 발명을 구현하기 위한 예시적인 시스템이 도시되어 있으며, 이는 마이크로 프로세서(11)와, 시스템 버스(39)와, 시스템 버스(39)를 통하여 마이크로 프로세서(11)에 접속되는 RAM(30), ROM(13) 등의 메모리를 포함한다. 다만, 이러한 컴퓨터 시스템(30)은 일 예로서, 본 발명에 따른 시스템은 이외에, 가전 기기의 제어 시스템이나, 휴대폰, PDA 등의 모바일 기기의 제어 시스템 등에도 적용될 수 있다.

다른 메모리 장치로서, ROM(13)에는 시동(start-up) 등의 컴퓨터 시스템(30) 내의 구성 요소 간의 정보를 전달하기 위한 기본적 루틴을 포함하는 BIOS(basic input/output system)가 저장되어 있다. 컴퓨터 시스템(30)은 또한 하드 디스크 드라이브(HDD; 31)와, 광 디스크(33)를 판독하거나 다른 광 매체에 대해 판독 또는 기록하기 위한 광 디스크 드라이브(ODD; 32)도 포함한다. 하드 디스크 드라이브(31), 광 디스크 드라이브(32)는 각각 하드 디스크 인터페이스(14), 광 드라이브 인터페이스(15)에 의해 시스템 버스(39)에 연결된다. 상기 드라이브들과 그들과 관련된 컴퓨터 판독 가능 기록 매체는 컴퓨터 시스템(30)에 비휘발성 저장 매체를 제공한다. 이상과 같은 컴퓨터 판독 가능 기록 매체에 대한 설명은 하드 디스크, 광 디스크를 참조하고 있지만, 당업자라면 자기 카세트, 플래시 메모리 카드, DVD(디지털 비디오 디스크) 등 컴퓨터가 판독 가능한 다른 매체도 상기한 예시 동작 환경에 사용할 수 있음을 이해할 수 있을 것이다.

운영 체제(Operating System; 21), 하나 이상의 어플리케이션 프로그램(22), 다른 프로그램 모듈(23) 등을 포함하는 다수의 프로그램 모듈은 하드 디스크 드라이브(31), RAM(20), 또는 ROM(13)에 저장할 수 있다.

사용자는 키보드(35)와, 마우스(34) 등의 포인팅 장치를 통해 컴퓨터 시스템으로 명령 및 정보를 입력할 수 있다. 다른 입력 장치(미도시됨)로는 마이크로폰, 조이스틱, 게임 패드, 스캐너 등을 들 수 있다. 이상에서 예시한 입력 장치들은 시스템 버스에 접속되는 직렬 포트 인터페이스(16)를 통해 마이크로 프로세서(11)에 접속되는 것이 보통이나, 게임 포트 또는 USB(universal serial bus) 등의 다른 인터페이스에 의해 접속되어도 좋다. 또한 모니터(29)를 비롯한 다양한 디스플레이 장치가 비디오 어댑터(12) 등의 인터페이스를 거쳐서 시스템 버스(39)에 접속된다. 컴퓨터 시스템은 모니터 외에도, 스피커 또는 프린터 등의 다른 주변 출력 장치(도시하지 않음)를 포함할 수도 있다.

컴퓨터 시스템(30)은 원격 컴퓨터 서버(19) 등의 하나 이상의 원격 컴퓨터에 대한 논리적 접속(logical connections)을 사용하는 네트워크 환경에서 작동할 수도 있다. 원격 컴퓨터 서버(19)는 서버, 라우터, 피어 장치(peer device) 또는 공통 네트워크 노드(common network node)일 수 있으며, 주로 컴퓨터 시스템(30)와 관련하여 설명한 구성 요소의 대부분 또는 전부를 포함하지만, 여기서는 단지 메모리 저장 장치(36)만을 도시하였다.

상기 논리적 접속은 LAN(local area network; 37)과 WAN(wide area network; 38)을 포함한다. 이러한 네트워크 환경은 사무실, 기업 전체 컴퓨터 네트워크(enterprise-wide computer network), 인트라넷, 인터넷에 있어서 통상적인 것이다.

컴퓨터 시스템(30)를 LAN 네트워크 환경에서 사용할 경우, 네트워크 인터페이스(17)를 통해 LAN(37)에 접속된다. 컴퓨터 시스템(30)를 WAN 네트워크 환경에서 사용할 경우에는, 컴퓨터 시스템(30)이 인터넷 등의, WAN(38)을 거쳐서 통신을 가능하게 하는 모뎀(18) 또는 다른 수단을 포함하는 것이 보통이다. 내부 또는 외부에 장착하는 모뎀(18)은 직렬 포트 인터페이스(16)를 통하여 시스템 버스(39)에 접속된다. 이상의 네트워크 접속은 예시적인 것이며, 컴퓨터 간의 통신 링크를 가능하게 하는 다른 수단도 사용할 수 있음을 이해하여야 할 것이다.

이와 같이, 본 발명에 따른 컴퓨터 시스템은 마이크로 프로세서(11)와 메모리를 적어도 포함한다. 마이크로 프로세서(11)는 특정 조건하에 운영 체제(21)로 분기될 수 있도록 분기 명령어를 가지고 있으며, 상기 메모리는 실행 도중에 수정이 가능한 RAM(20), 실행 도중 수정이 가능하지 않은 ROM(13) 등으로 구현될 수 있다.

본 발명은, 이러한 환경 아래에서 마이크로 프로세서(11)의 일반적인 명령어 만을 사용하여 마이크로 프로세서(11)의 제어가 메모리(13, 20)상의 특정 위치(예를 들어, 운영 체제가 로딩되어 있는 위치)로 분기하여, 특정 개수의 명령어를 실행한 후 원래 위치로 복귀하는 방안을 제공한다(Echo 기능). 그리고, 마이크로 프로세서(11)의 제어가 메모리(13, 20) 상의 특정 위치로 분기하고, 비트마스크(Bitmask)를 사용하여 비트마스크가 설정된 위치의 명령어만을 선택적으로 실행하고, 다시 원래 위치로 복귀할 수 있도록 한다(Bitmask Echo 기능). 이에 따라, 기존의 하드웨어를 수정하거나 별도의 하드웨어의 지원 없이도 프로그램의 크기를 효율적으로 줄일 수 있다.

도 2는 일반적인 마이크로 프로세서(11)가 메모리 상에서 실행하는 인스트럭션(instruction)의 형식의 예를 나타낸 것이다. 하나의 인스트럭션(50)은 마이크로 프로세서(11)의 동작 명령의 종류를 나타내는 동작 코드(Opcode)와, 적어도 하나 이상의 연산수(Operand; 52, 53)을 포함하여 구성된다. 현재 실행 중인 인스트럭션(50)이 코드 메모리에서 차지하는 위치에 대한 첫 바이트(first byte)를 기억하기 위하여, PC(Program Counter)라는 레지스터(register)가 사용된다. 상기 코드 메모리란, RAM(20), ROM(13) 등과 같은 메모리의 영역 중 마이크로 프로세서(11)에 의해 실행되는 코드 실행을 위하여 할당된 영역을 의미한다.

마이크로 프로세서(11)는 어떤 동작 코드를 실행하는 도중에, invoke 명령을 만나면, PC의 값을 현재 프레임(frame)의 PC를 시스템 스택에 저장하고, 새로운 프레임을 생성시킨 다음 새로이 실행하기 위한 메쏘드(method)의 시작 주소를 새로운 PC 값으로 넣어준다.

예를 들어, 코드 메모리의 소정 위치에 "mov <ax>, <1234H>"라는 인스트럭션이 존재한다고 한다면, 마이크로 프로세서(11)는 상기 인스트럭션의 작동 코드(mov) 및 연산수(ax, 1234H)를 읽고, 16비트의 1234라는 값을 ax라는 16비트 레지스터에 저장하게 된다.

도 3은 중복된 부분이 존재하는 코드 메모리(60) 구조를 도시하는 도면이다. 상기 코드 메모리(60)에는 복수의 인스트럭션(55)이 쌓여 있으며, 마이크로 프로세서(11)는 코드 메모리(60) 중 맨 위부터 순차적으로 인스트럭션(55)을 실행하게 된다.

도 3에서 음영으로 표시된 부분에 해당되는 인스트럭션 세트(61a, 61b)가 서로 동일하다고 한다면, 상기 중복된 인스트럭션의 크기 만큼 메모리가 낭비되는 것으로 볼 수 있다. 이러한 메모리의 비효율적 사용은 메모리의 제약이 특히 심한 가전기기나, 휴대용 기기 등에 있어서 더욱 문제가 된다. 본 발명에서 중복된 인스트럭션 세트(61a, 61b)는 적어도 하나 이상의 인스트럭션을 포함한다.

따라서, 도 3과 같은 인스트럭션의 구성으로 이루어진 프로그램에서, 상기 중복된 부분(61a, 61b) 중 하나를 제거하면서도 실제 마이크로 프로세서(11)가 동일한 인스트럭션을 실행하도록 할 필요가 있다. 하지만 실제로 이러한 역할을 할 수 있는 "Echo"와 같은 명령어를 마이크로 프로세서(11)가 직접 인식할 수 있도록 하기 위해서는 하드웨어적인 변경을 요하므로, 본 발명에서는 크게 6가지 실시예로서 이를 소프트웨어적인 방법으로 구현하고자 한다.

제1 실시예

도 4는 본 발명의 제1 실시예에 따른 코드 메모리(70) 구조를 도시하는 도면이다. 도 4의 코드 메모리(70)은 도 3의 코드 메모리(60)과 비교할 때, 중복된 인스트럭션 부분이 존재하지 않는 축약된 구조를 가진다. 특히, 도 3에서 중복된 첫번째 인스트럭션 세트(61a)는 사라지고 하나의 "Echo" 인스트럭션(71)이 그 대신에 추가된다. 또한, 두번째 인스트럭션 세트(61b) 다음에 소정의 인스트럭션(72)이 치환되어 삽입된 후 다시 원래의 인스트럭션으로 복원된다.

코드 메모리(70)의 4번째 위치, 즉 도 3에서 중복된 인스트럭션 세트(51)의 첫번째 위치에는 "Echo <Offset>, <Length>"라는 인스트럭션(71)이 위치한다. 물론, 본 발명에서 상기 Echo 인스트럭션(71)은 마이크로 프로세서(11)에 의하여 인식되지 못하는 명령어이다. 따라서, 마이크로 프로세서(11)가 상기 인스트럭션(71)을 실행할 때 정의되지 않은 인스트럭션(undefined instruction)에 의한 에러가 발생하며, 이에 따라 예외(exception) 처리 과정이 실행된다. 이 때, 코드 메모리(70)의 현재 프로그램 카운트는 PC1이다.

마이크로 프로세서(11)가 에러가 발생된 인스트럭션(71)의 내용을 운영 체제(21)에 전달하면, 운영 체제(21)는 상기 전달된 인스트럭션(71)의 동작 코드가 "Echo"임을 확인하고, 본 발명에 따른 예외 처리를 실행하게 된다. 이 때, 운영 체제(21)는 상기 인스트럭션(71)의 프로그램 카운트(PC1)를 읽어서 소정의 제1 버퍼에 저장한다. 그리고, 운영 체제(21)는 상기 PC1으로부터 상기 <Offset> 만큼 이동한 위치에서부터, 상기 이동한 위치에서 상기 <Length> 만큼의 이동한 위치까지, 반복 실행되어야 할 인스트럭션 세트(61b)가 존재함을 파악할 수 있다.

운영 체제(21)는 인스트럭션 세트(61b)의 마지막 위치의 다음 위치에 마이크로 프로세서(11)가 인식하지 못하는 임의의 인스트럭션(72)을 치환하여 기록한다. 그리고, 상기 다음 위치에 존재하던 인스트럭션은 소정의 제2 버퍼에 저장하여 둔다. 상기한 제1 버퍼 및 제2 버퍼는 특정 메모리 영역, 또는 코드 메모리(70) 중 일부 영역일 수 있다. 상기 인스트럭션(72)은 상기 "Echo" 인스트럭션(71)이어도 좋고 기타 마이크로 프로세서(11)가 인식하지 못하는 어떤 인스트럭션이어도 좋다.

또한, 운영 체제(21)는 코드 메모리(70)의 프로그램 카운트를 상기 PC1에서 <offset> 만큼 증가시키고 (PC2=PC1+ Offset), 리턴 명령에 의하여 마이크로 프로세서(11)에 인스트럭션 실행권을 넘겨 준다.

마이크로 프로세서(11)는 현재 프로그램 카운트가 PC2인 것을 확인하고, 인스트럭션 세트(61b)의 첫번째 인스트럭션부터 순차적으로 실행한다. 그런데, 마이크로 프로세서(11)가 인식할 수 없는 인스트럭션(72)를 실행하고자 하면 다시 에러가 발생하게 된다.

운영 체제(21)는 예외 처리를 위하여 마이크로 프로세서(11)로부터 인스트럭션(72)를 전달 받게 된다. 이 때 코드 메모리(70)의 현재 프로그램 카운트는 PC3이다. 운영 체제(21)는 상기 인스트럭션(72)이 자신이 치환/삽입한 인스트럭션임을 파악하고, 상기 인스트럭션(72)을 제2 버퍼에 저장된 원래의 인스트럭션으로 되돌려 놓는다.

그리고, 운영 체제(21)는 코드 메모리(70)의 현재 프로그램 카운트를 제1 버퍼에 저장된 프로그램 카운트(PC1)에 1단위만큼 더한 값, 즉 상기 PC1의 다음 프로그램 카운트로 변경한다. 상기 1단위는, 코드 메모리(70)의 인접한 인스트럭션 간의 프로그램 카운트의 차이를 의미하며, 코드 메모리(70)의 구현 방법에 따라서 다른 단위를 가질 수 있지만 통상 4비트로 나타난다. 운영 체제(21)는, 마지막으로 리턴 명령에 의하여 마이크로 프로세서(11)에 인스트럭션 실행권을 넘겨 준다.

마이크로 프로세서(11)는 현재 프로그램 카운트가 (PC1 + 1단위)인 것을 확인하고, 이에 해당되는 인스트럭션(73) 및 그 이후의 인스트럭션들을 실행한다. 그 결과, 모든 인스트럭션들의 실행 순서가 도 3에서와 동일하게 유지되면서 코드 메모리(70)에 포함된 인스트럭션의 수는 감소되었다.

제2 실시예

도 5는 본 발명의 제2 실시예에 따른 코드 메모리(80) 구조를 도시하는 도면이다. 제1 실시예는 예외 상황의 발생으로 인하여, 마이크로 프로세서(11)에 따라서 수개의 사이클(cycle) 만큼의 추가 비용을 야기할 수 있다. 제2 실시예는 이러한 추가 비용이 발생되지 않도록 제1 실시예와 같은 예외 처리 방법을 이용하지 않는다.

도 5에 제시된 것과 같이, 마이크로 프로세서(11)는 Echo 인스트럭션(71)을 실행할 때, 에러가 발생하지만 바로 예외 처리로 넘어가지 않고, 그 다음 "Branch" 인스트럭션(81)을 실행한다. 상기 인스트럭션(81)은 기 정의된 제1 시스템 함수로 분기할 것을 나타낸다.

제1 시스템 함수는 상기 제1 실시예에서 운영 체제(21)가 실행한 예외 처리 과정과 유사한 과정을 실행할 수 있도록 정의된다. 보다 구체적으로 보면, PC4을 읽어서 제1 버퍼에 저장하고, 코드 메모리(80)의 현재 프로그램 카운트를 PC4에서 <Offset> 만큼 증가시키고, 인스트럭션 세트(61b)의 다음 위치의 인스트럭션을 제2 버퍼에 저장하는 과정을 실행한다. 다만, 상기 다음 위치에 치환되는 인스트럭션은 "Branch" 인스트럭션(82)이다.

마이크로 프로세서(11)는 인스트럭션 세트(61b)를 실행한 후, Branch 인스트럭션(82)을 실행하고 다시 제2 시스템 함수로 분기하게 된다. 제2 시스템 함수는 현재의 프로그램 카운트(PC6)를 Branch 인스트럭션(81)의 다음 인스트럭션(73)을 가리키는 프로그램 카운트(PC4 + 1단위)로 변경한다. 이 때, 상기 PC4는 제1 버퍼로부터 제공된다. 또한, 제2 시스템 함수는 상기 Branch 인스트럭션(82)을 제2 버퍼에 저장된 원래의 인스트럭션으로 되돌려 놓는다.

그러면, 마이크로 프로세서(11)는 상기 프로그램 카운트(PC4 + 1단위)가 가리키는 인스트럭션(73) 및 그 이하 인스트럭션들을 실행하게 된다.

이와 같이, Branch 인스트럭션 및 시스템 함수를 이용하게 되면, 제1 실시예와 같이 하나의 Echo 명령을 위해서 두 번씩 발생하는 예외 상황을 해소할 수 있다. 다만, 하나의 인스트럭션(81)이 더 부가되므로 메모리의 효율적 활용 면에서는 제1 실시예가 보다 유리할 수 있다.

제3 실시예

도 6은 발명의 제2 실시예에 따른 코드 메모리(90) 구조를 도시하는 도면이다. 본 발명의 제3 실시예는, 도 6에 제시된 것과 같이 첫 번째와 두 번째 실시예의 일부를 변형한 것으로 운영 체제(21)의 예외 상황 처리부나, 기 정의된 시스템 함수에서 대상 분기 명령어들을 수정하지 않는다. 대신에, 실행해야 할 명령어를 자체적으로 가질 수 있는 임시 버퍼에 저장하고 이 위치에 맞게 주소 등을 변환한 후 실행하는 것이다. 제3 실시예는 원래 프로그램이 ROM(13)과 같이 수정이 불가능한 메모리로 구성되어 있을 때 유용하게 사용될 수 있다.

마이크로 프로세서(11)가 Echo 인스트럭션(71)을 만나게 되면, 현재 프로그램 카운트로부터 상기 인스트럭션(71)에 포함된 <Offset> 만큼 이동한 위치의 인스트럭션으로부터, <Length> 크기 만큼의 인스트럭션들 즉, 인스트럭션 세트(61b)를 임시 버퍼에 복사한다. 그 후, 마이크로 프로세서(11)는 임시 버퍼(99)에 쌓인 인스트럭션들을 순서대로 실행한다. 인스트럭션 세트(61b) 다음의 인스트럭션(91)은 코드 메모리(90)에서 마지막으로 실행된 Echo 인스트럭션(71)의 다음 인스트럭션으로 복귀하도록 명령하는 인스트럭션이다. 그 다음 마이크로 프로세서(11)는 복귀한 위치의 인스트럭션(73) 및 그 이하의 인스트럭션들을 실행한다.

이와 같이, 제3 실시예는 쓰기가 불가능한 코드 메모리(90)에서의 인스트럭션 실행시에, 임시적으로 데이터를 쓰고 지울 수 있는 임시 버퍼(99)를 이용함으로써 제1 실시예나 제2 실시예와 마찬가지의 효과를 얻을 수 있다.

이상에서는 일반적인 Echo 인스트럭션을 사용한 실시예들을 살펴 보았다. 그런데, 여기서 한 걸음 나아가서, 비트마스크 Echo 인스트럭션을 사용하는 실시예들을 위의 실시예와 마찬가지로 살펴 볼 필요가 있다. 일반적인 Echo 인스트럭션은 소정 개수 만큼 완전히 동일한 인스트럭션 세트에 대하여 적용될 수 있지만, 실제 인스트럭션은 일부만 중복이 일어날 수도 있는데 이러한 일부 중복된 인스트럭션 세트에 대하여 마이크로 프로세서 차원에서 코드 축약을 구현하는 기술이 바로, 비트마스크 Echo 인스트럭션이다.

도 7과 같은 코드 메모리(100)이 존재한다고 하면, 상기 코드 메모리(100) 내에는 상호간에 일부 중복만 존재하는 인스트럭션 세트들(101a, 101b)이 포함되어 있다. 즉, 두 인스트럭션 세트는 모두 인스트럭션 1, 3, 5, 6을 공통으로 포함하고 있지만, 인스트럭션 2, 4는 제2 인스트럭션 세트(101b)에만 포함되어 있다.

제4 실시예

제4 실시예는 제1 실시예와 대응되는 실시예로서, 그에 따른 코드 메모리(110)의 구조가 도 8에 도시된다. 제1 실시예와 차이가 나는 점은 비트마스크 Echo 인스트럭션(101)은 <Length> 대신에 <Bitmask>라는 연산수를 가지는 점에서 차이가 있다.

마이크로 프로세서(11)는 상기 인스트럭션(101)을 실행할 때 정의되지 않은 인스트럭션(undefined instruction)에 의한 에러가 발생하며, 이에 따라 예외(exception) 처리 과정이 실행되어야 한다. 이 때, 코드 메모리(110)의 현재 프로그램 카운트는 PC1이다.

마이크로 프로세서(11)가 에러가 발생된 인스트럭션(101)의 내용을 운영 체제(21)에 전달하면, 운영 체제(21)는 상기 전달된 인스트럭션(101)의 동작 코드가 "Echo"임을 확인하고, 본 발명에 따른 예외 처리를 실행하게 된다.

이 때, 운영 체제(21)는 상기 인스트럭션(101)의 프로그램 카운트(PC1)를 읽어서 소정의 제1 버퍼에 저장한다. 그리고, 운영 체제(21)는 상기 PC1으로부터 상기 <Offset> 만큼 이동한 위치에서부터, 상기 이동한 위치(PC2)에서부터 반복 실행되어야 할 인스트럭션 세트(101c)가 존재함을 파악할 수 있다.

운영 체제(21)는 <Bitmask>를 읽음으로써, 인스트럭션 세트(101c) 중 중복된 인스트럭션을 파악할 수 있다. 도 8과 같은 경우, <Bitmask>는 예를 들어 2진수 "110101"로 표기할 수 있다. 여기서, <Bitmask>의 크기는 상기 2진수의 자리수, 즉 6이며, "1"은 중복이 존재함을, "0"은 중복이 존재하지 않음을 나타낸다. 상기 예에서 2진수는 최하위 비트가 첫 번째 인스

트럭션의 실행 여부를 나타내는 역순의 형태로 구성되었지만, 정순의 형태로 구성되어도 무방하다. 상기 <Bitmask>를 뒤에서부터 읽어 보면, 6개의 인스트럭션 중 두번째 인스트럭션과, 네번째 인스트럭션은 중복이 발생되지 않은 인스트럭션임을 알 수 있다.

따라서, 운영 체제(21)는 두번째 인스트럭션 및 네번째 인스트럭션을 소정의 제2 버퍼에 저장한 후, 상기 인스트럭션들(104, 105)을 아무 명령도 실행하지 않는다는 의미의 인스트럭션인 "nop"(no operation; 104, 105)으로 치환한다. 또한, 제1 실시예에서와 마찬가지로, 운영 체제(21)는 인스트럭션 세트(101c)의 마지막 위치의 다음 위치에 마이크로 프로세서(11)가 인식하지 못하는 임의의 인스트럭션(102)을 치환하여 기록한다. 그리고, 상기 다음 위치에 존재하던 인스트럭션은 소정의 제3 버퍼에 저장하여 둔다.

그 다음, 운영 체제(21)는 코드 메모리(110)의 프로그램 카운트를 상기 PC1에서 비트마스크의 크기 만큼 증가시키고 (PC2= PC1 + 비트마스크의 크기 ×1단위), 리턴 명령에 의하여 마이크로 프로세서(11)에 인스트럭션 실행권을 넘겨 준다.

마이크로 프로세서(11)는 현재 프로그램 카운트가 PC2인 것을 확인하고, 인스트럭션 세트(101c)의 첫번째 인스트럭션부터 순차적으로 실행한다. 만약, 마이크로 프로세서(11)가 "nop" 인스트럭션(104, 105)을 만나면 단순히 그 다음 인스트럭션으로 넘어간다. 마이크로 프로세서(11)가 인스트럭션 세트(101c)를 실행한 후, 그 다음 인스트럭션(102)를 실행하고자 하면 다시 에러가 발생하게 된다.

운영 체제(21)는 예외 처리를 위하여 마이크로 프로세서(11)로부터 인스트럭션(102)를 전달 받게 된다. 이 때 코드 메모리(110)의 현재 프로그램 카운트는 PC3이다. 이 때, 운영 체제(21)는 nop 인스트럭션(104, 105)을 상기 비트마스크에 0으로 기록된 인스트럭션(104, 105), 즉 제2 버퍼에 기록된 인스트럭션으로 되돌려 놓는다. 이렇게 함으로써, 원래의 인스트럭션 세트(101c) 중 두번째 인스트럭션 및 네번째 인스트럭션이 복원된다. 또한, 운영 체제(21)는 상기 에러가 발생한 인스트럭션(102)을 제3 버퍼에 저장된 원래의 인스트럭션으로 되돌려 놓는다.

그리고, 운영 체제(21)는 코드 메모리(110)의 현재 프로그램 카운트를 제1 버퍼에 저장된 프로그램 카운트(PC1)에 1단위만큼 더한 값으로 변경한다. 운영 체제(21)는, 마지막으로 리턴 명령에 의하여 마이크로 프로세서(11)에 인스트럭션 실행권을 넘겨 준다.

마이크로 프로세서(11)는 현재 프로그램 카운트가 (PC1 + 1단위)인 것을 확인하고, 이에 해당되는 인스트럭션(103) 및 그 이후의 인스트럭션들을 실행한다. 그 결과, 모든 인스트럭션들의 실행 순서가 도 7에서와 동일하게 유지되면서 코드 메모리(110)에 포함된 인스트럭션의 수는 감소되었다.

제5 실시예

제5 실시예는 제2 실시예와 대응되는 실시예로서, 그에 따른 코드 메모리(120)의 구조가 도 9에 도시된다. 제5 실시예는 제4 실시예가 운영 체제(21)의 예외 처리를 통하여 이루어지던 것과 달리, 시스템 분기 함수(Branch to; 81, 82)에 의하여, 코드 메모리(110) 내의 인스트럭션으로부터 시스템 함수로 분기된다는 점을 제외하고는 제4 실시예와 마찬가지이므로 중복된 설명은 생략하기로 한다.

제6 실시예

제6 실시예는 제3 실시예와 대응되는 실시예로서, 그에 따른 코드 메모리(130)의 구조가 도 10에 도시된다. 본 발명의 제6 실시예는, 실행해야 할 명령어를 자체적으로 가질 수 있는 임시 버퍼에 저장하고 이 위치에 맞게 주소 등을 변환한 후 실행하는 것이다. 제6 실시예는 원래 프로그램이 ROM(13)과 같이 수정이 불가능한 메모리로 구성되어 있을 때 유용하게 사용될 수 있다.

마이크로 프로세서(11)가 비트마스크 Echo 인스트럭션(101)을 만나게 되면, 현재 프로그램 카운트로부터, 비트마스크의 크기 만큼 이동한 위치의 인스트럭션으로부터, 소정의 인스트럭션들(101a)을 임시 버퍼에 복사한다. 상기 소정의 인스트럭션들(101a)은 인스트럭션 세트(101b) 중에서 <Bitmask>에 의하여 중복이 발생되는 인스트럭션임이 표시된 인스트럭션을 의미한다. 만약, <Bitmask>가 "110101"로 표시되어 있다면, 인스트럭션 1, 3, 5, 6은 중복이 발생되는 인스트럭션이고, 인스트럭션 2, 4는 중복이 발생되지 않는 인스트럭션임을 의미한다.

그 후, 마이크로 프로세서(11)는 임시 버퍼(139)에 쌓인 인스트럭션들을 순서대로 실행한다. 인스트럭션 세트(101a) 다음의 인스트럭션(131)은 코드 메모리(130)에서 마지막으로 실행된 Echo 인스트럭션(101)의 다음 인스트럭션으로 복귀하도록 명령하는 인스트럭션이다. 그 다음, 마이크로 프로세서(11)는 복귀한 위치의 인스트럭션(103) 및 그 이하의 인스트럭션들을 실행한다.

이와 같이, 제6 실시예는 쓰기가 불가능한 코드 메모리(130)에서의 인스트럭션 실행시에, 임시적으로 데이터를 쓰고 지울 수 있는 임시 버퍼(139)를 이용함으로써 제4 실시예나 제5 실시예와 마찬가지의 효과를 얻을 수 있다.

본 발명은 프로그램의 크기를 크게 줄이는데 효율적으로 사용될 수 있는데 반하여 실행 속도를 매 사용시마다 수 사이클씩 지연시킬 수 있다. 이러한 문제점은 일반적은 컴파일로 최적화 기법에서 사용하는 프로파일링(profiling)을 통해서 개선할 수 있다. 세부적으로 프로그램의 코드 중에서 빈번하게 실행되는 부분과 그렇지 않은 부분을 분석하여 빈번하게 실행되지 않는 부분에만 본 발명을 적용한다면 프로그램의 크기도 크게 줄일 뿐 아니라 실행 속도에도 거의 영향을 주지 않게 될 것이다.

이상 첨부된 도면을 참조하여 본 발명의 실시예를 설명하였지만, 본 발명이 속하는 기술분야에서 통상의 지식을 가진 자는 본 발명이 그 기술적 사상이나 필수적인 특징을 변경하지 않고서 다른 구체적인 형태로 실시될 수 있다는 것을 이해할 수 있을 것이다. 그러므로 이상에서 기술한 실시예들은 모든 면에서 예시적인 것이며 한정적이 아닌 것으로 이해해야만 한다.

발명의 효과

본 발명에 따르면, 코드 메모리 상에서 실행되는 프로그램의 크기를 줄임으로써 컴퓨터 시스템에서 메모리의 요구 사양을 낮출 수 있다.

또한, 메모리 사용량을 줄임으로써 전력 소모를 감소시킬 수 있다.

도면의 간단한 설명

도 1은 본 발명을 구현하기 위한 예시적인 시스템을 도시하는 도면.

도 2는 일반적인 마이크로 프로세서가 메모리 상에서 실행하는 인스트럭션의 형식의 예를 나타낸 도면.

도 3은 중복된 부분이 존재하는 코드 메모리 구조의 일 예를 도시하는 도면.

도 4는 본 발명의 제1 실시예에 따른 코드 메모리 구조를 도시하는 도면.

도 5는 본 발명의 제2 실시예에 따른 코드 메모리 구조를 도시하는 도면.

도 6은 본 발명의 제3 실시예에 따른 코드 메모리 구조를 도시하는 도면.

도 7은 중복된 부분이 일부만 존재하는 코드 메모리 구조의 일 예를 도시하는 도면.

도 8은 본 발명의 제4 실시예에 따른 코드 메모리 구조를 도시하는 도면.

도 9는 본 발명의 제5 실시예에 따른 코드 메모리 구조를 도시하는 도면.

도 10은 본 발명의 제6 실시예에 따른 코드 메모리 구조를 도시하는 도면.

(도면의 주요부분에 대한 부호 설명)

11 : 마이크로 프로세서 13 : ROM

20 : RAM 21 : 운영 체제

30 : 컴퓨터 시스템

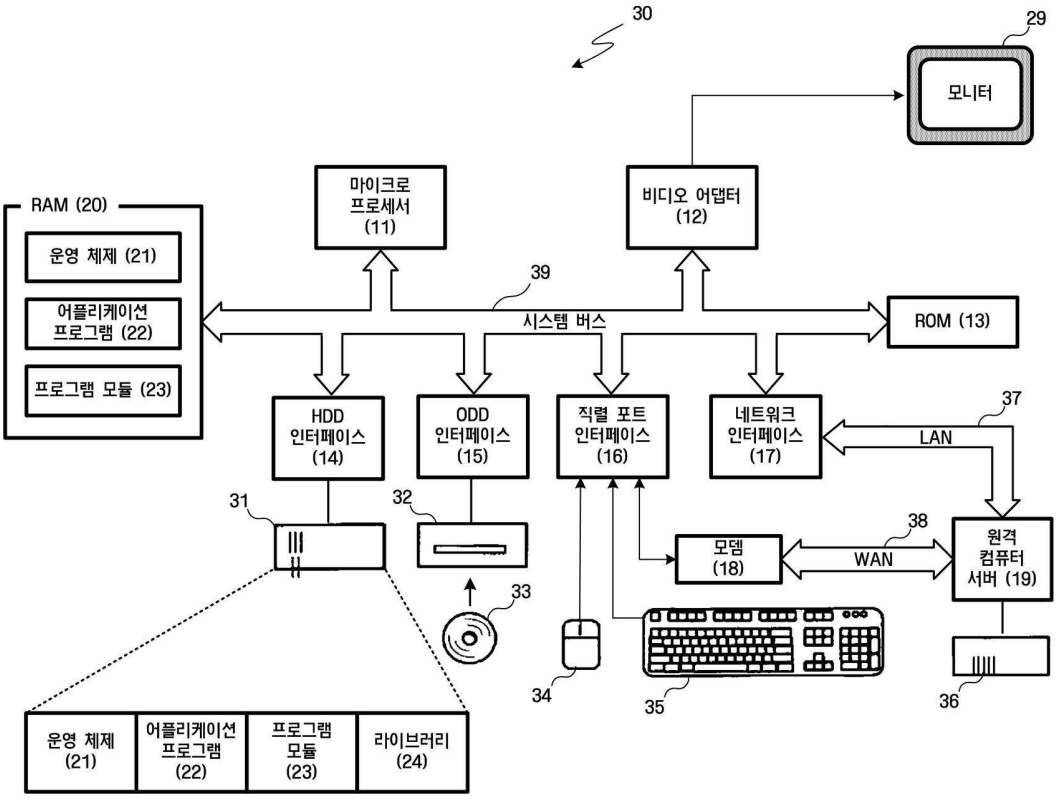
60, 70, 80, 90, 100, 110, 120, 130 : 코드 메모리

71 : Echo 인스트럭션 81 : Branch to 인스트럭션

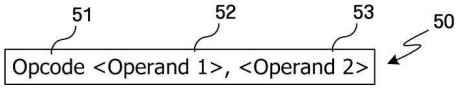
101 : 비트마스크 Echo 인스트럭션

도면

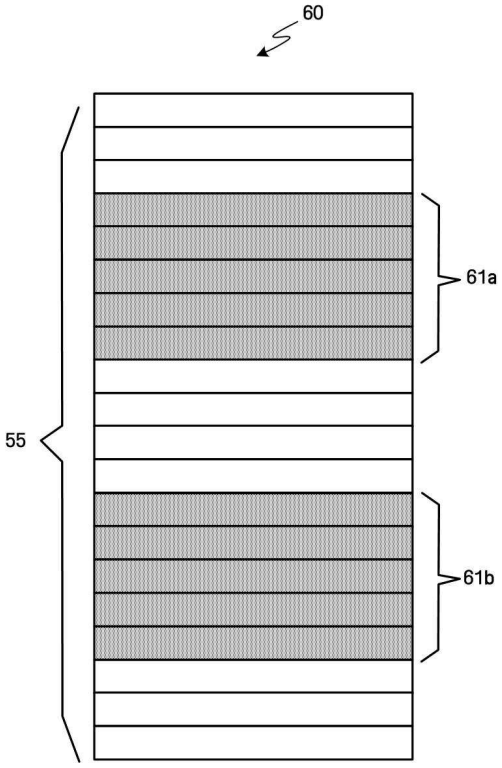
도면1



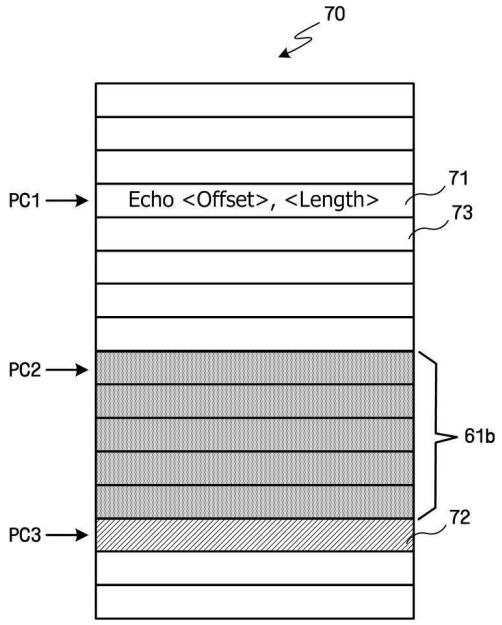
도면2



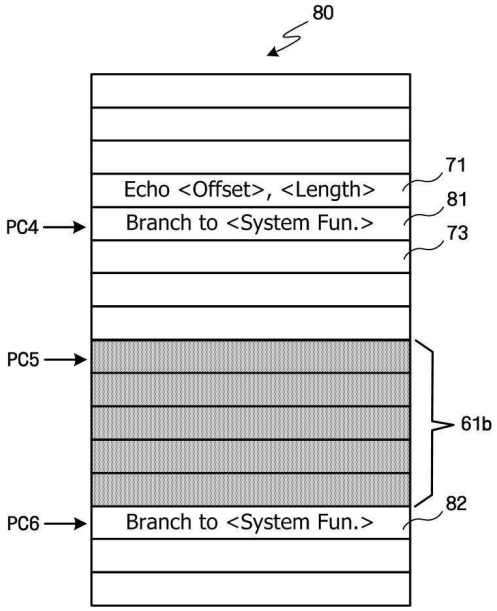
도면3



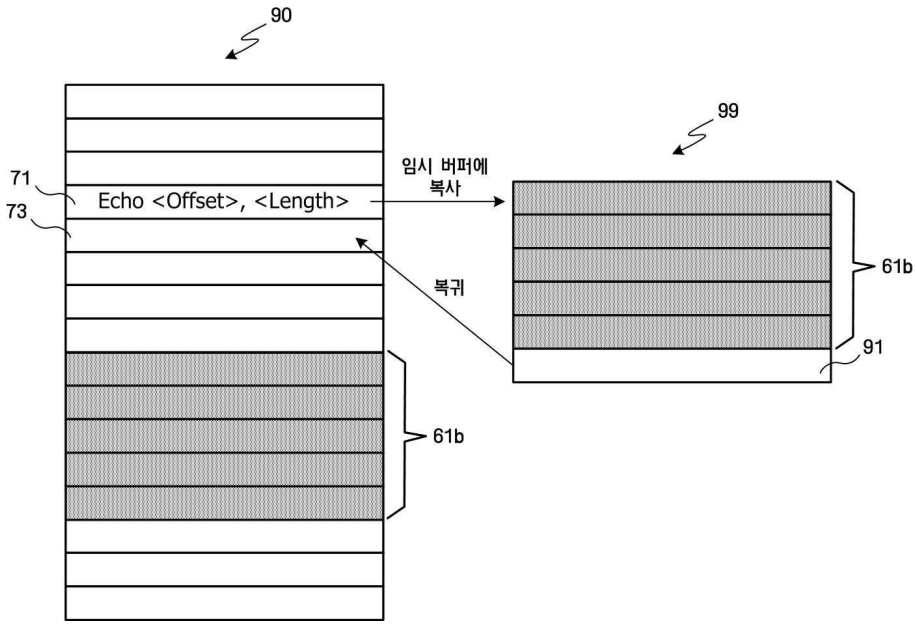
도면4



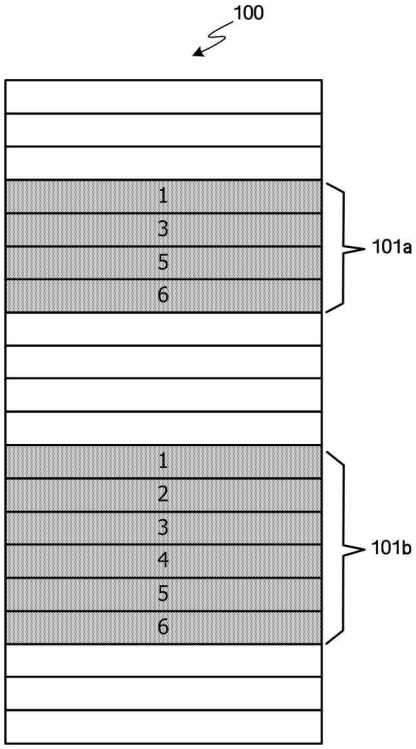
도면5


80
71
Echo <Offset>, <Length>
81
PC4
Branch to <System Fun.>
73
PC5
61b
82
PC6
Branch to <System Fun.>

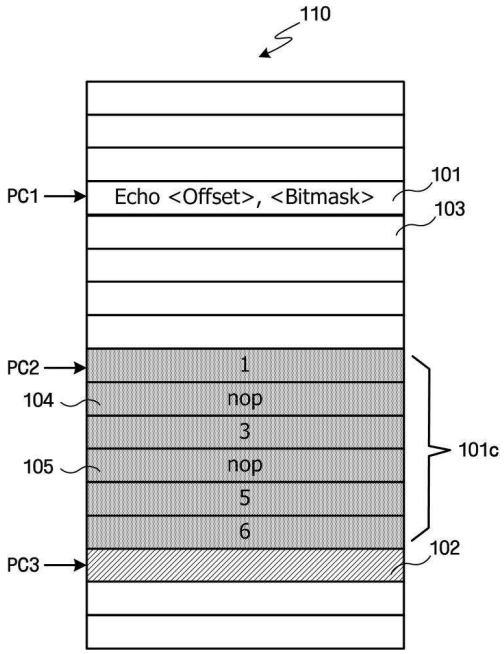
도면6


90
99
71
73
Echo <Offset>, <Length>
임시 버퍼에
복사
복귀
61b
61b
91

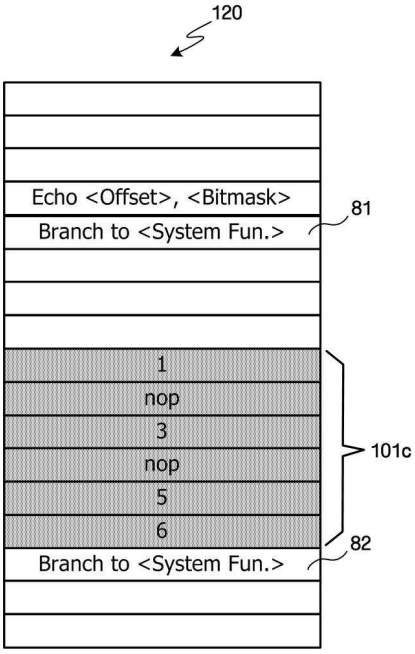
도면7


100
1
3
5
6
101a
1
2
3
4
5
6
101b

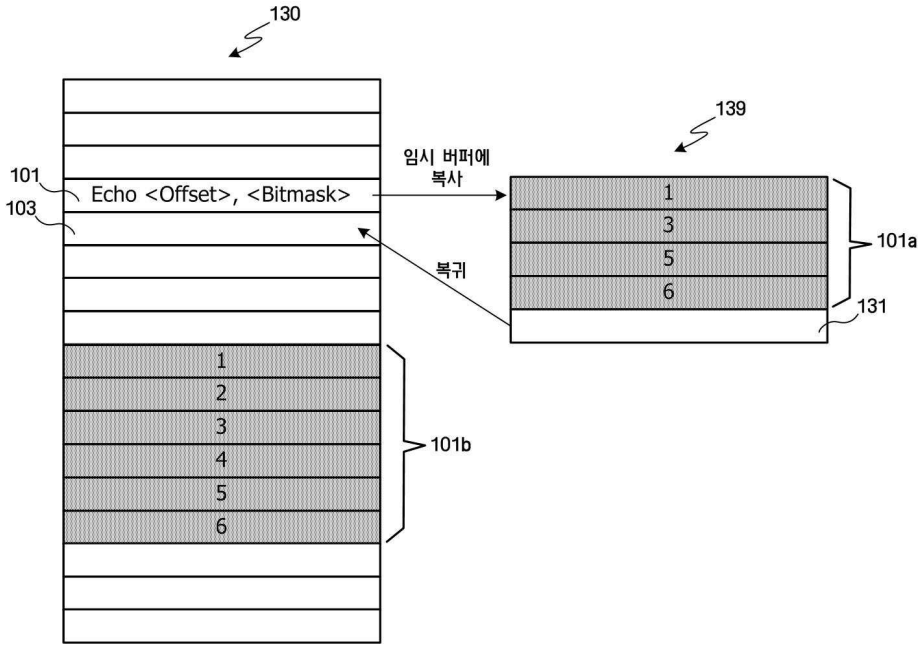
도면8


110
PC1
Echo <Offset>, <Bitmask>
101
103
PC2
1
104
nop
3
105
nop
5
6
101c
PC3
102

도면9


120
Echo <Offset>, <Bitmask>
Branch to <System Fun.>
81
1
nop
3
nop
5
6
101c
82
Branch to <System Fun.>

도면10


130
101
103
Echo <Offset>, <Bitmask>
1
2
3
4
5
6
101b
임시 버퍼에
복사
복귀
139
1
3
5
6
101a
131