(19) 대한민국특허청(KR)
(12) 공개특허공보(A)

(11) 공개번호 10-2015-0126245
(43) 공개일자 2015년11월11일

(51) 국제특허분류(Int. Cl.)
G06F 9/45 (2006.01)
(21) 출원번호 10-2014-0053712
(22) 출원일자 2014년05월02일
심사청구일자 없음

(71) 출원인
김현수
대전 유성구 엑스포로 448, 403동 902호 (전민동, 엑스포아파트)
(72) 발명자
김현수
대전 유성구 엑스포로 448, 403동 902호 (전민동, 엑스포아파트)

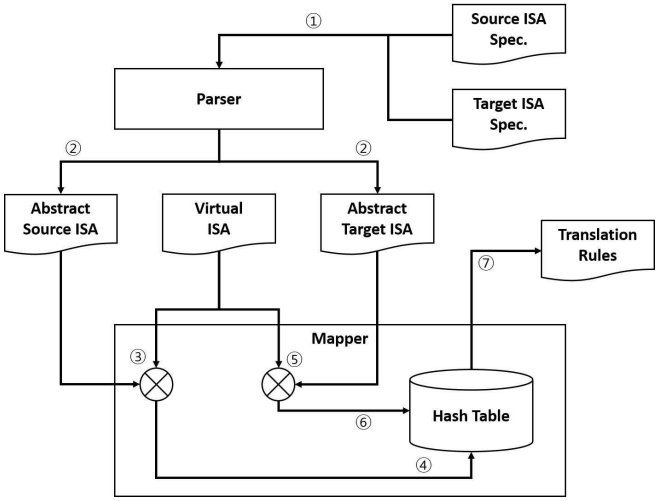
전체 청구항 수 : 총 1 항

(54) 발명의 명칭 **재목적성을 고려한 직접 매핑 기반의 이진 변환 규칙 생성 도구**

(57) 요 약

이진 변환은 특정 장치에서 동작하도록 구성된 프로그램을 다른 장치에서 동작할 수 있도록 재구성하는 과정을 말한다. 이진 변환을 수행하기 위해서는 두 장치 사이의 변환 규칙을 생성하는 것이 매우 중요하다. 변환 규칙을 생성하는 방법은 직접 매핑과 간접 매핑으로 나뉜다. 직접 매핑은 성능을 위한 방법인 반면, 간접 매핑은 재목적성을 위한 방법이다. 본 발명에서는 임베디드 시스템에 적합한 직접 매핑 기반의 이진 변환을 수행한다. 그렇지만 재목적성 역시 중요한 요구사항이기 때문에, 재목적성을 고려한 직접 매핑 기반의 이진 변환 방법을 제안한다. 또한 제안된 방법을 바탕으로 자동으로 변환 규칙을 생성하는 도구를 구현한다. 이 방법을 통해서 성능과 재목적성을 모두 고려한 변환 규칙을 생성할 수 있으며, 더 나아가 이진 변환을 수행하는데 소요되는 비용을 줄일 수 있다.

대 표 도



명 세 서

청구범위

청구항 1

.

발명의 설명

기 술 분 야

[0001] 특정 장치에서 동작하도록 구성된 프로그램을 다른 장치에서 동작할 수 있도록 재구성하는 과정인 이진 변환 기술에 관한 것이다.

배 경 기 술

[0002] 이진 변환(Binary Translation)은 특정 장치 M에서 동작하도록 구성된 이진 프로그램을 장치 M이 아닌 다른 장치에서도 동작할 수 있도록 프로그램을 재구성하는 기술을 말한다. 여기서 장치 M을 원본 장치(Source Machine)라 하고 장치 M 외의 다른 장치들을 대상 장치(Target Machine)라 한다. 원본 장치를 대상으로 구성된 프로그램은 원본 장치의 특성(예: 명령어, 레지스터, 메모리 등)이 반영되어 있기 때문에 대상 장치에서 동작할 수 없다. 따라서 원본 장치의 프로그램에 대상 장치의 특성이 반영되도록 재구성하는 과정이 필요하며, 이진 변환은 이와 같은 작업을 수행하는 기술이다. 이진 변환을 수행하는 과정에서 두 장치의 특성 사이의 관계를 설정하는 매핑(Mapping) 작업을 수행한다. 매핑을 통해 생성된 정보를 변환 규칙이라 한다.

[0003] 변환 규칙은 직접 매핑과 간접 매핑을 통해 생성된다. 직접 매핑은 두 장치의 특성 사이의 관계를 직접 설정하는 방법이다. 반면 간접 매핑은 컴파일 기술에서 사용되는 중간 표현을 활용하여 두 장치의 특성 사이의 관계를 간접적으로 설정하는 방법이다. 두 매핑 방법을 도식화하면 [도 1]과 같다. [도 1]에서 정적 시점은 이진 변환이 수행되기 이전을, 동작 시점(Runtime)은 이진 변환이 수행되는 시점을 의미한다. 동작 시점이 지나면 원본 장치의 프로그램이 동작 장치에 동작될 수 있는 형태로 변환된다. [도 1]을 통해 알 수 있는 직접 매핑과 간접 매핑의 가장 큰 차이점은 변환 규칙이 생성되는 시점이다. 직접 매핑은 정적 시점에 생성되며, 개발자가 직접 장치의 특성에 대해 이해한 뒤 변환 규칙을 작성하는 과정을 거친다. 따라서 변환 규칙을 생성하는데 많은 시간이 요구되는 방법이다. 또한 장치의 변경이 발생하면 변환 규칙을 다시 작성하여야 하는 문제를 갖는다. 하지만 동작 시점에서는 변환 규칙에 따라 변환만 수행되기 때문에 최적화된 변환을 수행할 수 있다. 간접 매핑은 동작 시점에 중간 표현 혹은 중간 코드를 이용하여 변환 규칙을 자동으로 생성한다. 간접 매핑은 변환 규칙이 자동으로 생성되기 때문에 장치가 변경되더라도 효과적으로 대처할 수 있다는 장점을 갖는다. 이러한 속성을 재목적성(Retargetability)라고 한다. 그러나 변환 규칙을 동작 시점에 생성하기 때문에 최적화된 변환을 수행하기 어렵다. 정리하면 직접 매핑은 성능에 최적화된 방법이며, 간접 매핑은 재목적성을 위한 방법이다. 두 방법은 서로 상충관계를 갖기 때문에 이진 변환을 적용하고자 하는 시스템의 특성에 맞게 적절한 매핑 방법을 선택하여 사용하여야 한다.

발명의 내용

해결하려는 과제

[0004] 본 발명은 이진 변환을 실시간성과 자원제약성이 요구되는 임베디드 시스템에 적용하고자 할 때 발생되는 문제를 해결하는데 목적이 있다. 본래 임베디드 시스템에 이진 변환을 적용하기 위해서는 가장 중요한 것은 성능이기 때문에 직접 매핑 기법을 사용하는 것을 올바르다. 그러나 이는 단일 시스템에 대해 이진 변환을 수행할 때에 한정된다. 만일 다수의 임베디드 시스템에 대해서 이진 변환을 수행하기 위해서는 재목적성 역시 달성되어야 한다. 따라서 본 발명에서는 직접 매핑 기법을 사용하되 재목적성도 함께 고려할 수 있는 방법을 제공하는데 목적이 있다.

과제의 해결 수단

[0005] 본 발명에서는 직접 매핑 기법을 기반으로 이진 변환을 수행하면서 재목적성도 함께 고려할 수 있는 방법을 제안한다. 이를 위해서 간접 매핑 기법을 응용하여 변환 규칙이 갖는 의존성을 낮추면서 정적 시점에 변환 규칙을 생성하는 방법을 제시한다. 재목적성을 달성하기 위해서 다음과 같은 요소를 고려한다.

[0006] 1. 장치의 특성을 작성할 수 있는 별도의 명세 언어를 제공한다.

[0007] 2. 변환 규칙을 자동으로 생성한다.

[0008] 본 발명에서는 제시된 두 가지 요소를 고려한 직접 매핑 기법을 통해 상기의 문제를 해결한다.

발명의 효과

[0009] 본 발명에서 제시한 방법을 이용하면 이진 변환을 수행함에 있어서 각 장치에 대한 깊은 이해가 없어도 변환 규칙을 생성할 수 있어 인력과 시간, 비용 효율성을 향상시킬 수 있다. 또한 한정된 자원을 바탕으로 이진 변환을 수행할 수 있어 임베디드 시스템에도 이진 변환을 적용할 수 있다.

도면의 간단한 설명

[0010] 도 1은 직접 매핑과 간접 매핑의 과정을 도식화한 것

도 2는 재목적성을 고려한 직접 매핑 기법

도 3은 명령어 추상 모델의 정의

도 4는 명령어 추상 모델을 통해 표현된 덧셈 명령어

도 5는 명령어 추상 모델을 통해 변환 규칙을 생성하는 알고리즘

도 6은 가상 ISA 명령어의 정의

도 7은 가상 ISA를 통해 중간 규칙을 생성하는 알고리즘

도 8은 가상 ISA를 통해 변환 규칙을 생성하는 방안을 수식화한 것

도 9는 이진 변환 규칙 생성 도구의 구조

도 10은 SLAB 언어의 문법

도 11은 SLAB 언어를 통해 작성된 명세서의 구조

도 12는 SLAB 언어에서 사용되는 연산자의 종류

도 13은 SLAB 언어에서 사용되는 자료 유형의 종류

도 14는 SLAB 언어를 통해 명세를 작성하는 예제

도 15는 이진 변환 규칙 생성 도구의 한 모듈인 매퍼의 동작 과정

발명을 실시하기 위한 구체적인 내용

[0011] 1. 재목적성을 고려한 직접 매핑 기반의 이진 변환 방법

[0012] 본 발명은 상기에 제시된 재목적성 달성 요소를 통해 [도 2]와 같은 매핑 방법을 제안한다. 제안한 매핑 방법의 핵심 아이디어는 중간 표현을 정적 시점에서 활용하여 변환 규칙을 생성하는 것이다. 여기서 중간 표현을 통해 추상화된 ISA(Instruction Set Architecture)를 생성하는 과정이 규칙 생성 도구에 반쯤 걸쳐있는 형태로 표현

된 것은 이 과정이 방법에 따라 수동 혹은 자동으로 수행될 수 있기 때문이다. 중요한 것은 이를 통해서 정적 시점에 변환 규칙을 자동으로 생성할 수 있으며, 이는 곧 장치의 변경에 대해 효과적으로 대처할 수 있다는 것이다. ISA를 추상화하는 것보다는 변환 규칙을 생성하는 것이 더 많은 비용을 요구하기 때문에, 혹여 ISA 추상화가 수동으로 수행된다고 하더라도 이 방법이 갖는 효과에 끼치는 영향은 적다. [도 2]의 방법은 간접 매핑처럼 중간 표현을 사용하지만, 이를 이용하여 변환 규칙을 생성한다는 측면에서 차이가 있다. 간접 매핑에서는 원본 장치의 프로그램으로부터 변환 규칙을 생성하지만, [도 2]의 방법에서는 원본 장치의 ISA로부터 변환 규칙을 생성한다.

[0013] 제안한 매핑 방법은 재목적성을 달성할 수 있는 방법이기 때문에, [도 2]와 같은 방법으로 변환 규칙을 생성한다면 재목적성을 고려한 직접 매핑 기반의 이진 변환을 수행할 수 있다. 이제는 재목적성 달성의 핵심이 되는 중간 표현을 통해 변환 규칙을 자동 생성하는 방법에 대해서 기술한다.

[0014] 1.1 명령어 추상 모델 기반의 변환 규칙 생성

[0015] 이 방법은 명령어 추상 모델을 중간 표현으로 이용하여 변환 규칙을 생성한다. 중간 표현으로 RTL(Register Transfer Language) 기반으로 작성된 명령어의 행위를 AST(Abstract Syntax Tree)로 표현하며, AST로 표현된 명령어를 비교함으로써 매핑을 수행한다. 이 방법은 프로그램을 구성하는 원본 장치의 명령어 하나가 대상 장치의 명령어로 변환될 수 있다면, 프로그램 전체를 대상 장치의 프로그램으로 변환할 수 있다는 생각을 바탕으로 하고 있다. 따라서 명령어 추상 모델은 단일 명령어를 위한 중간 표현이며, 이를 통해 각 명령어의 행위를 표현할 수 있다.

[0016] 1.1.1 명령어 추상 모델

[0017] 명령어 추상 모델은 SLAB(Specification Language for Automatic rule generation of Binary translation)으로 작성된 ISA 명세로부터 생성되며, SLAB은 RTL 기반으로 명령어의 행위를 작성하도록 구성된다. 명령어 추상 모델은 도구를 통해 자동으로 생성되기 때문에, ISA 명세는 동일한 포맷으로 작성되어야 한다. 따라서 기존의 ISA 명세를 이용하지 않고 SLAB을 통해 새로운 명세를 작성한다. SLAB을 통한 ISA 명세 작성은 개발자에 의해 이루어진다. SLAB의 문법과 이를 통한 작성 예는 2.1절에서 다룬다.

[0018] 명령어 추상 모델은 트리(Tree) 구조로 [도 3]과 같이 정의된다. 노드 집합 N 은 명령어의 기능과 자료 유형을 표현하기 위한 단위 요소들의 집합이며, 에지 집합 E는 노드 간의 연결을 표현한다. 이때, n_i가 n_j의 부모 노드가 되며, 부모 노드와 자식 노드는 연산자와 피연산자 관계를 갖는다. 각 단위 요소의 의미는 M:n→s에 의해 부여된다. 의미 집합 S는 기능을 표현하는 기능 의미인 S_f와 자료 유형을 표현하는 자료 의미인 S_d로 구성된다. S_f는 20개의 원소를 가지며, 기능을 표현하는 값으로 구성된다. S_d는 자료 유형 t와 자료 크기 l로 구성되며, 자료 유형의 종류에는 명령어 단계에서 공통적으로 사용되는 자료 유형인 정수, 실수, 상수가 존재한다.

[0019] 명령어 추상 모델을 통해서 SAPRC v7의 ADD 명령어에 대해서 추상화하면 [도 4]과 같이 표현된다.

[0020] 1.1.2 명령어 추상 모델 사이의 비교 연산을 이용한 변환 규칙 생성

[0021] 명령어 추상 모델은 명령어의 행위를 표현하기 때문에, 어떤 두 명령어 추상 모델이 서로 동치 관계에 있다면 두 명령어가 서로 동일한 행위를 한다고 볼 수 있다. 이러한 관계가 성립하기 때문에, 본 발명에서는 모델 사이의 비교를 통해 변환 규칙을 생성한다. 단, 동일한 행위를 하는 명령어끼리만 변환 규칙을 생성할 수 있는 것은 아니다. 여기서는 특정 명령어의 행위를 대신 수행할 수 있는 명령어를 찾는 것이 변환 규칙을 생성하는 것이다. 따라서 모델 사이의 유사성에 대한 정의가 필요하며, 다음과 같은 조건이 성립할 때 모델이 서로 유사하다고 판단한다.

[0022] 1. 두 모델의 트리 형태는 동일하여야 한다.

[0023] 2. 두 모델의 트리 형태가 동일할 때, 동일한 위치에 존재하는 노드에 부여된 의미가 일치하여야 한다.

[0024] 3. 동일한 위치에 존재하는 두 노드에 부여된 의미가 기능 의미라면 두 노드의 의미는 동일하여야 한다.

[0025] 4. 동일한 위치에 존재하는 두 노드에 부여된 의미가 자료 의미라면 대상 장치의 노드가 갖는 의미가 원본 장치의 노드가 갖는 의미를 포함하여야 한다.

[0026] 자료 의미에 대한 유사성을 위와 같이 정의한 이유는 명령어는 자신이 처리할 수 있는 자료 유형보다 작은 자료 유형을 수용할 수 있기 때문이다. 예를 들어 64비트의 정수 덧셈이 가능한 명령어는 64비트보다 작은 자료 유형인 32비트 혹은 16비트의 덧셈도 수행할 수 있다. 따라서 에 대한 유사성을 위와 같이 정의한다. 정의된 유사성을 토대로 명령어 추상 모델 기반의 변환 규칙 생성은 [도 5]의 알고리즘과 같다.

[0027] [도 5]의 알고리즘은 원본 장치의 모든 명령어에 대해서 대상 장치의 모든 명령어와 비교하며 수행된다. 5번째 줄의 내용은 의미 없는 비교를 피하기 위한 것이다. 서로 노드의 수가 같지 않으면, 두 모델의 형태가 같을 수 없으므로 비교를 수행하지 않는다. 9번째의 CompareTree()는 16번째에 기술된 함수로써, 두 모델의 루트 노드부터 순차적으로 비교하여 두 모델이 같은지 판별한다. 18번째 줄은 두 노드에 부여된 의미가 기능 의미인 경우에 수행되는 비교이며, 통상적으로 기능 의미를 의미로 갖는 노드는 말단 노드가 아니기 때문에 자식 노드에 대해 재귀적으로 호출한다. 자료 의미를 의미로 갖는 노드는 말단 노드이기 때문에, 23번째 줄과 같이 참 값을 반환한다. 이와 같은 방법을 통해 두 모델을 비교하고 변환 규칙을 생성한다.

[0028] 그러나 이 방법을 통해 생성되는 변환 규칙은 원본 장치의 명령어에 대해 모두 커버하지 못한다. 그 이유는 원본 장치와 대상 장치가 갖는 명령어가 모두 동일한 행위를 갖는 명령어로만 구성되지 않기 때문이다. 한 예로 SPARC v7과 TI의 정수 덧셈 명령어를 들 수 있다. 두 장치의 명령어는 모두 덧셈을 수행하지만, 그 방법은 서로 다르다. SPARC v7에서는 오직 32비트 정수형에 대한 덧셈을 제공하되 캐리 비트를 사용하여 다양한 자료 유형에 대한 덧셈을 수행하는 반면, TI에서는 애초에 다양한 자료 유형에 대한 덧셈 명령어를 각각 제공한다. 두 장치의 명령어는 목적은 같지만, 행위적인 측면에서는 다른 명령어로 분류될 수밖에 없다. 본 발명에서는 이를 보완하기 위해 가상 ISA 기반의 변환 규칙 생성 방법을 제시한다.

[0029] 1.2 가상 ISA 기반의 변환 규칙 생성

[0030] SPARC v7과 TI가 갖는 정수 덧셈 명령어의 구성은 다르지만, 이는 결국 다양한 정수 자료 유형에 대한 덧셈 기능 제공이라는 동일한 목적을 갖는다. 따라서 본 발명에서는 행위적 측면에서의 중간 표현 대신 결과적 측면에서의 중간 표현을 사용하여 변환 규칙을 생성한다. 결과적 측면에서의 중간 표현을 위해 가상 ISA를 제안한다. 이를 통해서 AST에서 고려하지 못한 특성에 대해서도 반영할 수 있다.

[0031] 1.2.1 가상 ISA

[0032] 가상 ISA는 어떤 ISA든지 가질 수 있는 명령어의 집합이며, 가상 ISA 내의 명령어는 기능과 출력을 통해 결과적 측면에서 분류된다. 하지만 실제로 기능과 출력만을 이용하여 명령어 사이의 비교를 수행하는 것은 불가능하다. 만일 가상 ISA의 명령어 중 64비트 정수를 출력하는 덧셈 명령어가 존재한다면, 32비트 정수 덧셈 명령어만 지원하는 SPARC v7에서는 해당 기능을 수행하는 명령어를 찾아낼 수 없다. 그러나 SPARC v7은 올림수(Carry bit)를 이용하여 64비트 정수 덧셈을 수행할 수 있으며, 이는 SPARC v7의 덧셈 명령어가 올림수를 사용하여 동작한다는 사실을 파악해야 알 수 있다. 즉, 결과적 측면에서의 중간 표현이라고 할지라도 행위에 대한 기술은 필요하다.

[0033] 앞서 수행된 연구를 통해 단일 명령어의 행위에만 집중하지 않아야 함을 파악하였다. 본 발명에서는 명령어의 결과적 측면에 집중하되 행위적 측면도 함께 고려할 수 있는 방법으로 가장 단순한 방법을 적용한다. 가상 ISA의 명령어를 기능의 결과에 대한 정보와 결과를 도출하기 위해 가질 수 있는 모든 행위로 구성하는 것이다. 이와 같은 방법을 적용하기 위해 가상 ISA 명령어를 정의한다.

[0034] 가상 ISA는 다수의 명령어 (f1, B)로 구성되며 [도 6]과 같이 정의된다. f1은 명령어의 기능을 결과적인 측면에서 분류한 값으로, 기능 라벨(Functional Label)을 의미한다. 명령어 추상 모델의 기능 의미와 유사해 보이지만, 명령어의 기능과 자료 유형을 하나로 묶어 표현하였으며 기능 의미보다 세분된 형태를 갖는다. B는 가상 ISA의 명령어의 행위 집합을 의미하며, B_i는 명령어의 기능을 수행할 수 있는 행위 중 하나의 행위 인스턴스를 의미한다. B_i는 명령어 추상 모델의 집합으로 표현되는데, 이는 가상 ISA의 명령어 기능을 수행할 때 다수의 실제 명령어가 필요하기 때문이다. 예를 들어 64비트 정수 덧셈 명령어에 대해서 필요한 SPARC v7의 명령어의 개수는 두 개이다. SPARC v7은 32비트 정수 덧셈 명령어만 존재하기 때문에 두 개의 명령어를 통해 64비트 덧셈 연산을 수행하여야 한다. 또한 B는 두 가지로 구분된다. 그 중 B_T는 기능에 대한 통상적인 행위(Typical behavior)를 의미하며, 항상 1개의 원소로 구성된다. B_A는 그 외의 행위(Alternative behavior)를 의미하며, B_T의 원소를 제외한 모든 B의 원소로 구성된다. 따라서 B_A의 원소 개수는 0보다 큰 값을 갖는다.

[0035] 하나의 기능에 대해 다수의 행위를 표현함으로써 명령어 추상 모델이 갖는 약점을 해결할 수 있지만, 이는 또 다른 문제를 야기한다. 이와 같은 표현으로 인해 가상 ISA의 명령어에 대해서 다수의 원본 장치 혹은 대상 장치의 명령어가 매핑될 가능성이 있다. 이런 문제를 해결하기 위해 우선순위 함수를 사용한다. 즉, 다수의 명령어와 매핑 되었을 경우에는 우선순위 함수를 통해 가장 높은 우선순위를 갖는 명령어를 채택한다.

[0036] **[정의] 행위 우선순위 함수**

[0037] 명령어 (f1, B)가 갖는 행위 인스턴스 B_T와 B_A에 대해서 항상 $priority(B_T) > priority(B_A)$가 성립한다. 또한 B_A를 구성하는 행위 인스턴스 사이의 우선순위는 각 행위와 매핑된 명령어의 개수에 반비례한다. 다시 말해 매핑된 명령어의 개수가 적을수록 우선순위가 높아진다. 이는 매핑된 명령어의 개수가 적을수록 더 낮은 실행 시간을 가질 수 있기 때문이다. 마지막으로 우선순위 함수는 절대 값이 아닌 상대 값을 갖는다.

[0038] 위와 같은 정의를 통해서 가상 ISA는 명령어의 기능에 대해서 결과적 측면과 함께 행위적 측면을 표현할 수 있으며, 특정 기능이 수행될 수 있는 모든 행위를 표현함으로써 명령어 추상 모델이 갖는 문제를 해결할 수 있다.

[0039] 1.2.2 가상 ISA 기반의 변환 규칙 생성 방법

[0040] 가상 ISA도 기본적으로 명령어 추상 모델을 기반으로 정의되었기 때문에, 명령어 추상 모델과 마찬가지로 트리의 유사성을 통해 변환 규칙을 생성한다. 다만, 명령어 추상 모델 기반의 변환 규칙 생성에서는 원본 장치와 대상 장치의 명령어를 직접 비교하였던 것과 달리 가상 ISA 기반의 변환 규칙 생성에서는 원본 장치와 가상 ISA의 명령어, 대상 장치와 가상 ISA의 명령어 사이의 비교가 수행된다. 본 발명에서는 이 과정을 중간 규칙 생성 과정이라고 하며, 이를 통해 중간 규칙이 생성된다. 중간 규칙은 가상 ISA와 실제 ISA 사이의 변환 규칙을 말한다. 중간 규칙 생성 과정이 끝나고 난 뒤, 가상 ISA의 명령어를 기준으로 원본 장치와 대상 장치 사이의 변환 규칙을 생성한다. 먼저, 중간 규칙 생성은 [도 7]의 알고리즘의 반복을 통해 수행되며, 이는 [도 5]의 알고리즘과 크게 다르지 않다.

[0041] 기본적으로 [도 7]의 알고리즘은 [도 5]의 알고리즘과 마찬가지로 CompareTree()를 이용하여 비교 연산을 수행한다. 차이가 있다면, 가상 ISA의 명령어가 다수의 행위를 가지고 있기 때문에 더 많은 비교 연산을 수행한다는 것이다. 또한 가상 ISA의 명령어가 갖는 행위 중 어느 행위와 규칙이 생성될지 알 수 없기 때문에 3번째 줄의 중간 규칙 테이블 T를 사용한다. 4번째 줄부터 13번째 줄까지는 중간 규칙 테이블 T를 채우는 작업을 수행한다. 중간 규칙 테이블 T를 모두 채우고 나면, 중간 규칙 테이블 T의 행 중에서 모든 원소를 채운 행에 대해서만 중간 규칙을 생성한다. 이때, 우선순위가 가장 높은 행위에 대해서 중간 규칙을 생성하도록 한다(17번째 줄). 이와 같은 방법을 통해 원본 장치와의 중간 규칙 집합 IR_S와 대상 장치와의 중간 규칙 집합 IR_T를 생성한다. 생성된 중간 규칙을 바탕으로 가상 ISA 기반의 변환 규칙 생성을 수행한다. 이는 관계 대수(Relational Algebra)를 이용하여 표현할 수 있으며 [도 8]과 같다.

[0042] 2. 직접 매핑 기반의 이진 변환을 위한 변환 규칙 생성 도구

[0043] 본 발명에서는 가상 ISA 기반의 직접 매핑 방법을 통해 변환 규칙을 생성하는 도구를 구현한다. 또한 이 도구는 직접 매핑과 유사한 단계로 동작하도록 구성되며, 그 구성은 [도 9]과 같다. 직접 매핑은 총 세 단계를 통해 진행된다. 첫 번째 단계는 이해 단계로, 변환 관계에 있는 두 장치의 특성을 파악하는 단계이다. 두 번째 단계는 탐색 단계이다. 탐색 단계에서는 이해 단계에서 파악된 특성을 바탕으로 매핑을 수행한다. 세 번째 단계는 작성 단계로, 변환기를 생성한다. 각 단계를 수행하는 모듈의 이름은 각각 파서(Parser), 매퍼(Mapper), 생성기(Generator)이다.

[0044] 2.1 SLAB(Specification Language for Automatic rule generation of Binary translation)

[0045] 이해 단계를 수행하기 위해서는 도구가 ISA 명세를 파악할 수 있어야 한다. SLAB은 이와 같은 목적으로 정의되었으며, 명령어의 행위, 다루는 자료 유형, 이진 포맷 등을 표현할 수 있다. SLAB의 문법은 [도 10]과 같다. 전체 문법 중 일부만을 첨부한다. SLAB은 크게 프로세서 심볼 정의부와 명령어 정의부로 나뉜다. 프로세서 심볼 정의부에는 프로세서에서 사용하는 고유 레지스터를 기술하며, 8 번째 줄의 ‘symbol’을 통해 심볼 입력을 시작한다, 명령어 정의부는 명령어의 행위, 이진 포맷 등을 입력하는 부분이며, 예약어인 ‘instruction’을 통해 시작된다. 이후에 명령어의 식별자를 기술하며(28번째 줄), 명령어의 식별자는 문자열로 작성되는 Identifier와 이진수로 작성되는 BitIdentifier로 구성된다. 명령어의 행위와 관련된 문법은 62 번째 줄부터 시작된다. 여기서 사용된 instructionBehavior는 C 언어의 표현식을 참조하여 구성하였다. SLAB의 문법을 통해 작성되는 문서의 구조는 [도 11]과 같다. [도 11]을 보면 알 수 있듯이 명령어에 대한 기술은 크게 명령어의 행위와 명령어의 이진 포맷으로 나뉜다. 명령어의 행위는 [도 10]의 instrucionBehavior와 관련된 부분이며, 이를 위해 제공하는 연산자는 [도 12]와 같다. 연산자 외에도 사용가능한 자료 유형 역시 중요하며, SLAB에서 제공하는 자료 유형은 [도 13]과 같다.

[0046] [도 12]의 연산자는 총 21개이며, 이 중에서 대입 연산자(=)를 제외한 20개의 연산자는 명령어의 행위를 표현하는데 사용된다. 20개의 연산자는 통상적으로 명령어 단계의 행위를 표현하는데 필요한 것을 정리한 것이다. 대입 연산자는 명령어의 자료 유형을 기술할 때 사용되는 연산자이다. [도 13]의 자료 유형은 명령어 추상 모델에서 사용되는 자료 유형에 맞추어 정의된다. 또한 통상적으로 명령어 단계에서 사용되는 자료 유형은 정수형 레지스터, 실수형 레지스터, 명령어 내부에 포함되는 상수이므로, [도 13]의 자료 유형을 통해 명령어의 행위를 충분히 기술할 수 있다. SLAB에서는 이진 포맷을 기능 식별자와 자료 식별자로 나누어 입력한다. 먼저, 기능 식별자는 [도 10]의 BitIdentifier와 관련된 부분으로, 특정 명령어에 대해서 변하지 않는 부분을 의미한다. 기능 식별자를 입력할 때에는 실제 비트 위치와 관계없이 하나의 이진수 형태로 입력한다. 경우에 따라 “_”를 통해 0과 1의 값을 모두 표현하기도 한다. 자료 식별자는 레지스터 번호, 상수, 메모리 주소 등과 같이 수행될 때마다 변경될 수 있는 부분을 의미한다. 자료 식별자의 입력은 [도 12]의 대입 연산자(=)를 통해 이루어지며, 기능 식별자와 달리 정확한 비트 위치를 기입하여야 한다.

[0047] 자세한 입력 방법은 예제를 통해 설명한다. [도 14]는 SLAB을 통해 작성된 JMPL 명령어의 명세이다. SLAB을 이용하여 명령어를 명세하기 위해서는 먼저 사용되는 심볼을 찾아야 한다. SLAB에서는 일반 레지스터 외에는 전부 특수 레지스터로 판단하기 때문에, 프로그램 카운터(PC: Program Counter)를 심볼로 입력한다. 이후에 명령어에 대한 기술을 시작한다. 기능 식별자는 이진 포맷 중 변경되지 않는 일련의 비트이다. 이때, 비트의 실제 위치는 무시하고 상대적인 위치만 유지한다. 따라서 JMPL 명령어의 기능 식별자는 “10111000_”가 된다. “_”는 0과 1 모두 올 수 있는 것으로, i 비트를 표기한 것이다. i 비트는 두 번째 피연산자의 자료 유형을 결정하는 비트로, 기능 식별자에서는 “_”로 표시한 뒤 selection 구문을 이용하여 추가적인 정보를 입력한다. 명령어의 행위는 명령어가 수행하는 기능에 의거하여 SLAB의 연산자를 통해 표현한다. JMPL는 특정 레지스터에 현재의 위치를 저장한 뒤 “src1 + src2”의 주소로 이동하는 연산자이므로, Transfer 연산자와 Branch 연산자를 이용하여 이를 표현한다. 행위를 입력하는 과정에서 레지스터의 식별자는 임의의 문자열 식별자로 표기하되, 각 식별자의 자료 유형과 자료 식별자도 함께 표기한다. 이러한 과정을 통해서 SLAB을 통해 명령어를 명세할 수 있다.

[0048] 2.2 매퍼

[0049] 매퍼는 주로 가상 ISA의 명령어와 원본 혹은 대상 장치의 명령어 사이의 변환 규칙을 생성하는 역할을 수행한다. 매퍼는 1.2.2절에서 제시된 [도 7]의 알고리즘을 바탕으로 구현되었으며, [도 15]와 같은 방식으로 동작한다. 동작 과정에 대한 설명은 아래와 같다.

[0050] ① 원본 장치와 대상 장치의 ISA를 SLAB을 통해 기술한다.

[0051] ② 기술된 SLAB 명세로부터 파서가 추상화된 ISA, 즉 명령어 추상 모델을 도출한다.

[0052] ③ 가상 ISA와 원본 장치의 명령어 추상 모델과의 비교를 수행한다.

[0053] ④ 수행 결과는 해시 테이블에 저장되며, 이때 해시 테이블의 키 값은 가상 ISA의 명령어가 갖는 기능 라벨이다.

[0054] ⑤ 가상 ISA와 대상 장치의 명령어 추상 모델과의 비교를 수행한다.

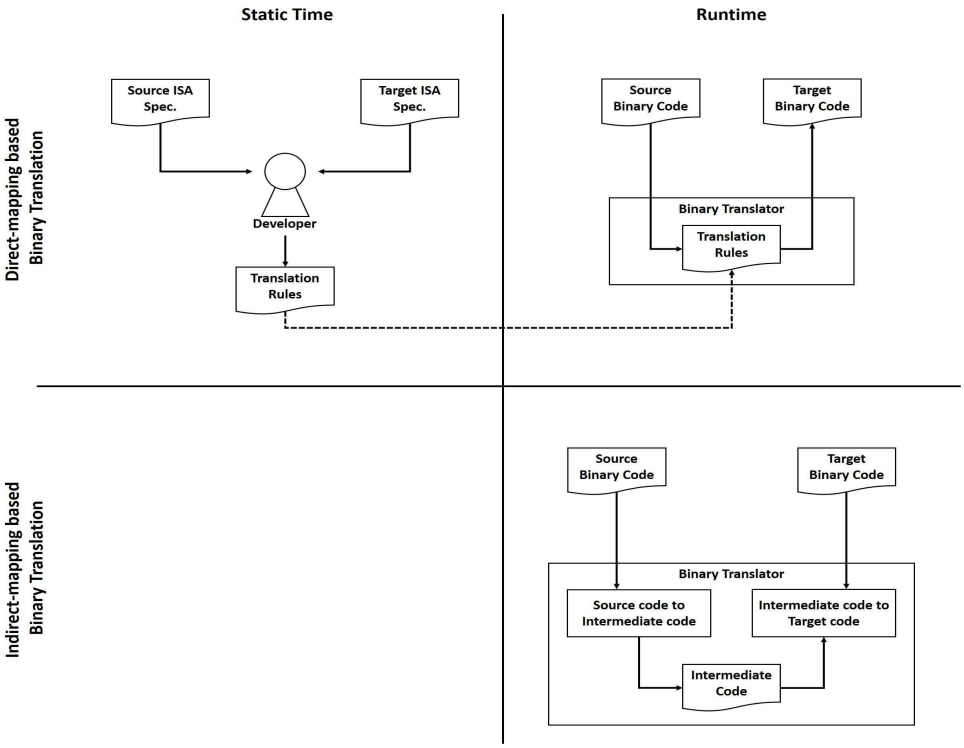
[0055] ⑥ 수행 결과를 해시 테이블에 저장한다. 이때 사용되는 해시 테이블은 ④의 해시 테이블과 같다.

[0056] ⑦ 해시 테이블의 저장 결과를 토대로 변환 규칙을 생성한다. 같은 키 값을 갖는 요소를 엮어 변환 규칙을 생성한다.

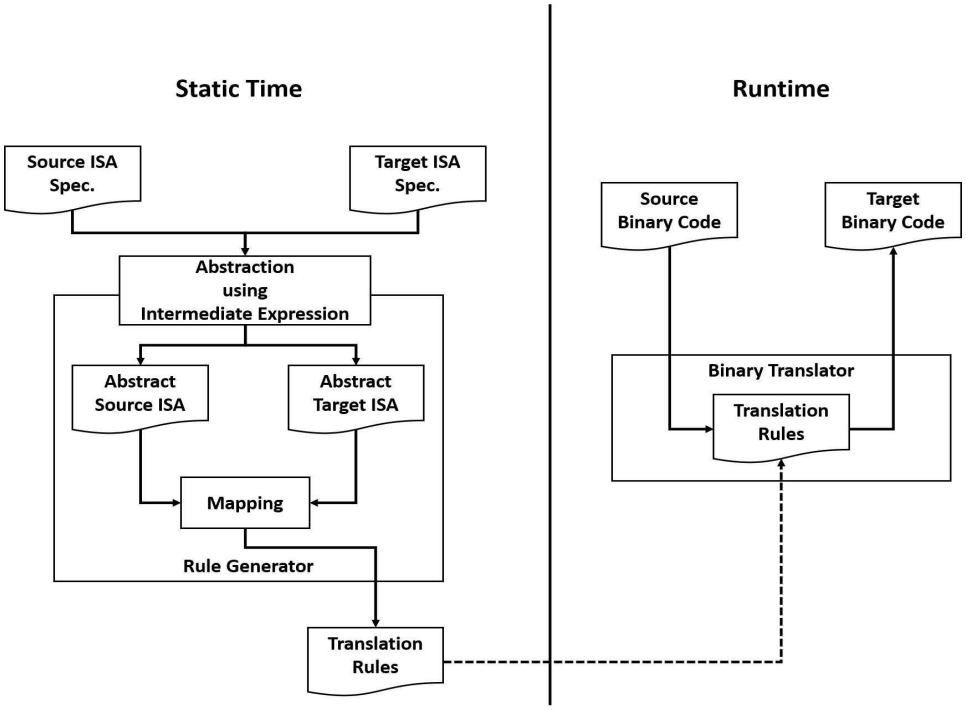
[0057] 이와 같은 방식을 통해 매퍼는 가상 ISA 기반의 직접 매핑 기법을 수행하게 되며, 이는 자동으로 수행되기 때문에 재목적성을 달성할 수 있다.

도면

도면1



도면2



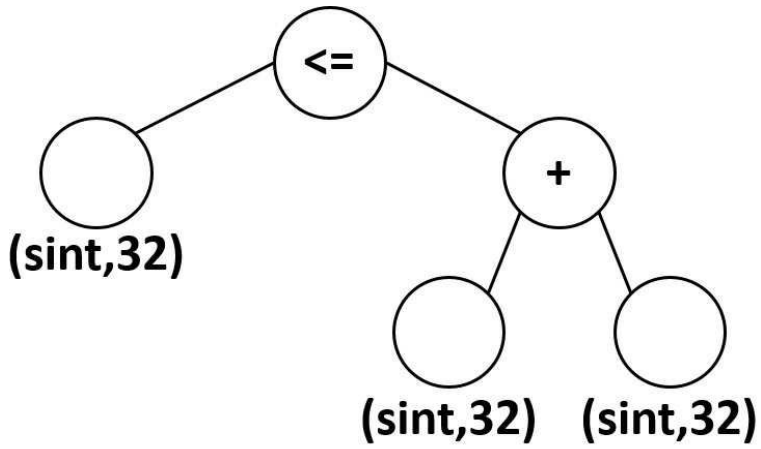
도면3

$$I=(N,E)$$
$$E=\{(n_i,n_j)\,|\,n_i,n_j\in N\}$$

$$M: n\rightarrow s,\ n\in N,\ s\in S=S_f\cup S_d$$

$$S_f=\{transfer,add,\ldots\},\ n(S_f)=20$$
$$S_d=\{(t,l)\,|\,t\in T,\ l\in \mathbb{N}\}$$
$$T=\{uint,sint,float,uimm,simm\}$$

도면4



$$rd \leftarrow rs1 + rs2$$

도면5

Algorithm GenerateTRs_usingIAM

Input : a set of source instructions ISA_s

: a set of target instructions ISA_t

Output : a set of translation rules TR

```
1   procedure GenerateTRs_usingIAM
2       TR ← ∅
3       for each I_s ∈ ISA_s do
4           for each I_t ∈ ISA_t do
                // I_s = (N_s, E_s), I_t = (N_t, E_t)
5               if |N_s| ≠ |N_t| then
6                   go back to line 4
7               end if
                // n_i is a root node of I_s
                // n_j is a root node of I_t
8               if CompareTree(n_i, n_j) = true then
9                   TR ← TR ∪ {(I_s, I_t)}
10              end if
11          end for
12      end for
13      return TR
14  end procedure
15
16  procedure CompareTree
17      isEqual ← true
18      if M(n_i), M(n_j) ∈ S_f ∧ M(n_i) = M(n_j) then
19          for each child node pair
                    (n_k, n_l) | (n_i, n_k) ∈ E_s, (n_j, n_l) ∈ E_t do
20              isEqual ← isEqual ∧ CompareTree(n_k, n_l)
21          end for
22      else if M(n_i), M(n_j) ∈ S_d ∧ t_i = t_j ∧ l_i ≤ l_j then
23          isEqual ← true
24      else
25          isEqual ← false
26      end if
27      return isEqual
28  end procedure
```

도면6

$$VI_i = (fl, B)$$

$$B = \{B_i \mid B_i = \{I_1, I_2, \ldots, I_j\}, j \geq 1\}, |B| = 1$$

$$B = B_T \cup B_A, |B_T| = 1, |B_A| \geq 0$$

도면7

Algorithm GenerateIR

Input : a virtual instruction VI

: a set of actual instructions ISA

Output : intermediate rule ir

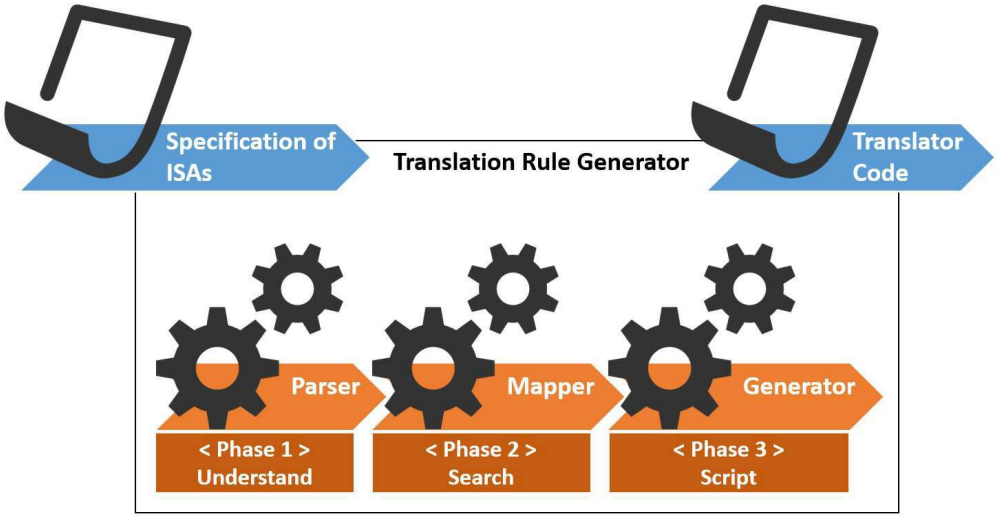
```
1   procedure GenerateIR
2     ir ← ∅
3     T ← ∅  // T is 2-dimensional array
4     for each I_i ∈ ISA do
5       for each I_k ∈ B_j, B_j ∈ B do
          // I_i = (N_i, E_i), I_k = (N_k, E_k)
6         if |N_i| ≠ |N_k| then
7           go back to line 4
8         end if
          // n_i is a root node of I_i
          // n_k is a root node of I_k
9         if CompareTree(n_k, n_i) = true then
10          T_jk ← I_i
11        end if
12      end for
13    end for
14    for each T_j ∈ T do
15      if ∀k, T_jk ≠ ∅ then
16        // if ir is not null, ir = (fl, T_z)
17        if ir ≠ ∅ ∨ priority(T_z) < priority(T_j) then
18          ir ← (fl, T_j)
19        end if
20      end if
21    end for
22    return ir
23  end procedure
```

도면8

$$IR_s = \{(fl_s, B_s)\}$$
$$IR_t = \{(fl_t, B_t)\}$$

$$TR = \Pi_{B_s, B_t}\left(IR_s \bowtie_{fl_s = fl_t} IR_t\right)$$

도면9



도면10

```
....
specification
: processSymbolDeclearation?
instructionDescriptionList
;
processSymbolDeclearation
: 'symbol' '(' processSymbolList ')'
;
.....
instructionDescription
: 'instruction' instrID compoundStatement
;

instrID
: Identifier? '(' BitIdentifier ')'
;
.....
compoundStatement
: '{' instructionBehavior? statement* '}'
;
.....
instructionBehavior
: transferExpression+
;
....
```

도면11

```
symbol { /* processor symbol list */ }

instruction "name" ("functional identifier")
{
// Description of instruction's behavior

// Description of data identifier

// Optional :
// Description of selection statements
}
....
```

도면12

Category	Operator
Arithmetic Operations	+ (Add), - (Sub), * (Multiply), / (Division), % (Modular)
Logical Operations	& (And), | (Or), ~ (Not), ^ (Xor), << (Left Shift), >> (Right Shift)
Comparison Operations	== (Equal), != (Not Equal), < (Less Than), <== (Less or Equal), > (Greater Than), >==(Greater or Equal)
Transfer Operation	<= (Transfer), @ (Branch)
Assignment Operation	= (Type Assignment)
Memory Operation	[ ] (Memory Access)

도면13

Data Type	Description
uint<size>	• Unsigned integer register with <size> • Available value of <size> is 8, 16, 32, 64
sint<size>	• Signed integer register with <size> • Available value of <size> is 8, 16, 32, 64
float<size>	• Floating-point register with <size> • Available value of <size> is 32, 64
uimm<size>	• Unsigned immediate value with <size> • Available value range of <size> is 1 to 32
simm<size>	• Signed immediate value with <size> • Available value range of <size> is 1 to 32

도면14

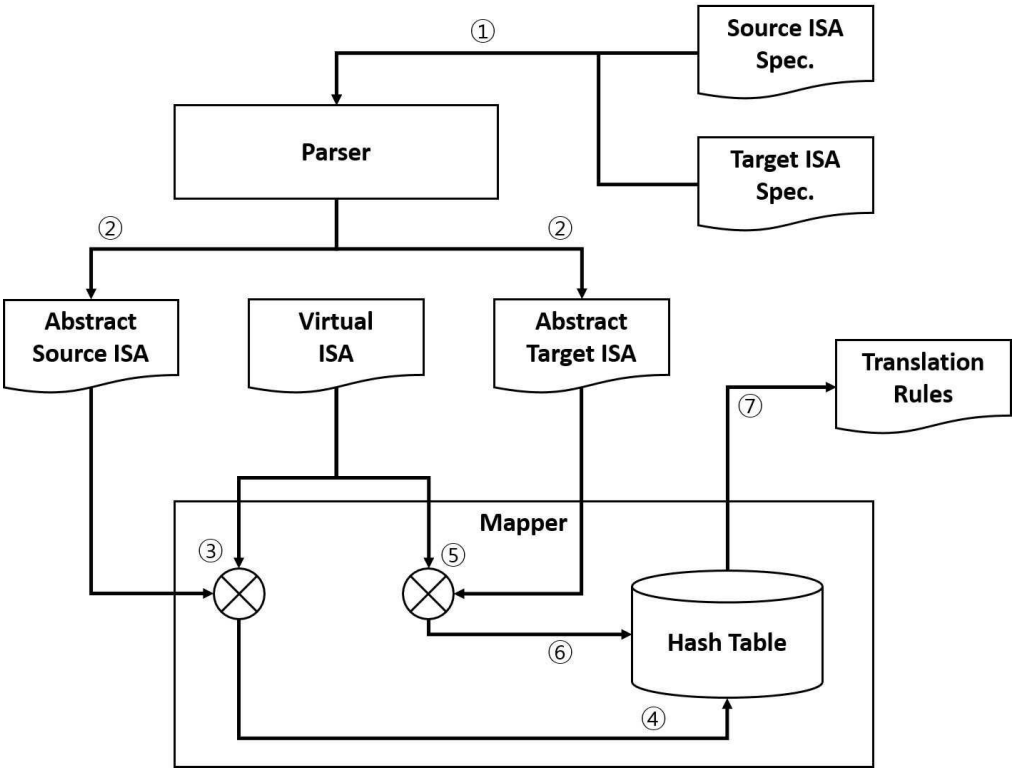
```
symbol { pc }

instruction JMPL ( B'10111000_ )
{
    dest <= pc ;
    @(src1 + src2) ;

    dest = sint[32]<25:29> ;
    src1 = sint[32]<14:18> ;
    pc = sint[32] ;

    selection
    {
        B'0 :
            src2 = sint[32]<0:4> ;
        B'1 :
            src2 = simm[13]<0:12> ;
    }
}
```

도면15


①
Source ISA Spec.
Parser
Target ISA Spec.
②
②
Abstract Source ISA
Virtual ISA
Abstract Target ISA
Translation Rules
⑦
Mapper
③
⑤
Hash Table
⑥
④