(19) 대한민국특허청(KR)
(12) 공개특허공보(A)

(11) 공개번호 10-2011-0093965
(43) 공개일자 2011년08월19일

(51) Int. Cl.
G06F 17/22 (2006.01) *G06F 17/21* (2006.01)
(21) 출원번호 10-2011-0065573(분할)
(22) 출원일자 2011년07월01일
심사청구일자 없음
(62) 원출원 특허 10-2009-0109753
원출원일자 2009년11월13일
심사청구일자 2009년11월16일
(30) 우선권주장
1020080116417 2008년11월21일 대한민국(KR)

(71) 출원인
고려대학교 산학협력단
서울 성북구 안암동5가 1
(72) 발명자
김선옥
경기도 남양주시 도농동 부영아파트2단지 203동 1604호
누엔띠홍장
서울 성북구 안암동3가 16번지
(74) 대리인
전종학

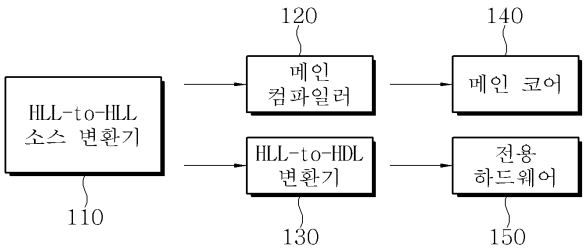
전체 청구항 수 : 총 1 항

(54) 고수준 언어 코드를 HDL 코드로 변환하는 방법 및 시스템

(57) 요 약

본 발명은 C, C++, 포트란 및 자바와 같은 고수준 언어로 된 코드를 Verilog 또는 VHDL과 같은 HDL코드로 변환하는 방법 및 시스템에 관한 것으로서, 본 발명에 따른 시스템은 Programming Directive로 표시된 변환대상 고수준 언어 코드로부터 상기 Programming Directive를 읽은 후 상기 변환대상 고수준 언어 코드를 하드웨어 부분 코드와 소프트웨어 부분 코드로 파티션하는 HLL-to-HLL 소스 변환기, 상기 하드웨어 부분 코드와 소프트웨어 부분코드를 컴파일하는 메인 컴파일러, 상기 하드웨어 부분 코드를 HDL 코드로 변환하는 HLL-to-HDL 변환기, 컴파일된 상기 소프트웨어 부분 코드를 실행하는 메인코어 및 변환된 상기 HDL 코드를 실행하는 전용하드웨어를 포함하는 것을 특징으로 한다.

대 표 도 - 도1



특허청구의 범위

청구항 1

(1) 고수준 언어 코드의 변환 대상 부분(translation-targeted section)을 프로그래밍 디렉티브(programming directive)를 이용하여 표시하는 단계;

(2) 표시된 상기 고수준 언어 코드의 변환 대상 부분을 하드웨어부분 코드 및 상기 고수준 언어 코드의 나머지 부분을 소프트웨어부분 코드로 분할(partition)하는 단계;

(3) 상기 소프트웨어부분 코드 및 하드웨어부분 코드를 메인 컴파일러에 의해 컴파일하고, 상기 하드웨어부분 코드를 HLL-to-HDL(High-level language code to hardware description language) 변환기에 의해 HDL 코드로 변환되는 단계; 및

(4) 상기 소프트웨어 부분 코드는 메인코어에서 실행되며, 상기 HDL 코드는 전용하드웨어에서 실행되는 단계를 포함하며,

상기 (4) 단계의 전용 하드웨어는 상기 소프트웨어 부분 코드와 상기 하드웨어부분 코드 사이의 크로스 콜을 지원하기 위한 SW/HW 인터페이스를 포함하는 것을 특징으로 하는 고수준 언어 코드의 HDL 코드로의 변환 방법.

명 세 서

기 술 분 야

[0001] 본 발명은 C, C++, 포트란, 자바 등과 같은 고수준 언어 코드를 Verilog 및 VHDL과 같은 HDL 코드로 변환하는 방법 및 시스템에 관한 것으로, 보다 구체적으로는 고수준 언어코드의 컴파일러에 덧붙여지는 HDL 백엔드(Back End)를 이용하여 고수준 언어로된 코드를 HDL(Hardware Description Language) 코드로 변환한 후 변환된 HDL 코드를 논리 합성하고 메인코어와 연동되도록 함으로써 응용프로그램의 속도를 향상시킬 수 있는 고수준 언어 코드를 HDL 코드로 변환하는 방법 및 시스템에 관한 것이다. 또한, 전용하드웨어에서 실행되는 변환된 HDL 코드와 메인코어에서 실행되는 소프트웨어 코드 사이의 크로스 콜 및 하드웨어 내부에서의 재귀호출(recursive call)을 위한 프레임 워크에 관한 것이다.

배 경 기 술

[0002] 지난 수 십년간 디지털 회로 설계는 반도체 제조 기술의 급격한 발전에 따라 빠르게 진화해왔다. 초기의 디지털 회로는 진공관과 트랜지스터로 설계되었으며, 하나의 칩위에 논리회로를 집적한 집적회로(IC, Integrated Circuit)가 그 뒤를 이었다. 기술의 발전으로 논리회로(Gate)의 수가 아주 적었던 SSI(Small Scale Integration)로 부터 MSI(Medium Scale Integration), 칩 위에 수 천개의 Gate를 집적해야하는 LSI(Large Scale Integration), 및 LSI의 복잡성보다 매우 큰 복잡성을 갖는 VLSI(Very Large Scale Integration) 기술로 발전하였다. 이후 범용 프로세서(General Purpose Processor), DSPs(Digital Signal Processors), 및 ASICs(Application Specific Integrated Circuit) 등과 같은 VLSI는 점점 더 활용되어지고 있다.

[0003] Fortran, Pascal, C와 같은 프로그래밍언어들은 오랫동안 순차적으로 동작하는 컴퓨터 프로그램을 기술하는데 사용되었다. 이와 비슷하게 디지털 회로 설계자들은 디지털회로를 기술할 수 있는 표준화된 언어의 필요성을 느껴왔으며, 이에 따라 만들어진 것이 HDL(Hardware Description Language)이다. HDL은 하드웨어의 특성에 맞는 동시발생의 과정을 기술할 수 있었으며, 대표적인 HDL이 VerilogHDL과 VHDL이다. Verilog HDL은 Gateway Design Automation에서 시작되었고, VHDL은 DARPA(미국방위고등연구계획국)의 지원으로 개발되었다. 디지털 회로 개발자들은 디지털 시스템의 발전을 위해 HDL을 활용하였으며, 설계된 HDL 코드는 HDL 컴파일러에 의해 합성(synthesize)되었다.

[0004] 현재 많은 시스템 개발자들은 마이크로프로세서보다 와트당 성능이 뛰어나다는 점 및 ASIC보다 설계의 유연성 때문에 FPGA(Field Programmable Gate Array)를 자신들의 개발 플랫폼에 더할 필요성을 느끼고 있으며, 종종 범용 프로세서 또는 DSPs를 FPGA로 대체하여 활용하고 있다. 그러나, HDL 코드를 이용하여 FPGA를 활용하기 위해서는 하드웨어 설계에 대한 많은 지식이 요구되며, 이에 따라 대부분의 소프트웨어 개발자들이 FPGA를 활용하는데 현실적인 어려움이 있다. 따라서, 하드웨어의 구성에 대한 걱정없이 개념적인 수준의 언어인 고수준 언어

코드로 하드웨어 시스템을 개발할 필요성이 증가하고 있으며, 이를 위해 일반적으로 활용되는 고수준 언어코드를 HDL코드로 변환하는 도구에 대한 필요성이 증가하고 있다.

[0005] 따라서, 이러한 문제를 해결하기 위해 종례 소프트웨어 수준의 언어 코드를 HDL 코드로 변환하여 주는 도구 내지 방법에 대한 많은 연구가 행하여져 왔다. 그러나, 고수준 언어 코드를 HDL 코드로 변환함에 있어서 하드웨어와 소프트웨어의 프로그래밍에 개념적으로 큰 차이가 존재하고, 이에 따라 포인터, 2차원 어레이 등의 소프트웨어 언어의 모든 syntax를 HDL 언어로 표현하는데 커다란 어려움이 있기 때문에 종례의 도구 내지 방법은 단지 SystemC, SA-C 또는 Streams-C와 같은 시스템레벨 언어 코드의 HDL 코드로의 변환 만을 지원하거나, 또는 고수준 언어의 HDL로의 직접 변환을 지원하더라도 단지 고수준 언어로 된 syntax의 일부분만을 직접 변환할 뿐으로 많은 한계점을 가지고 있었다.

[0006] 나아가, 하드웨어와 소프트웨어에 대한 코드 생성에서의 차이점 때문에 소프트웨어와 하드웨어 사이의 크로스 콜(cross call) 내지 하드웨어 내부에서의 재귀호출(recursive call)을 지원하지 않는 문제점이 있었다.

발명의 내용

해결하려는 과제

[0007] 상기와 같은 문제점을 해결하기 위한 본 발명의 목적은 변환대상 고수준 언어 코드에 대한 수정없이도 모든 고수준언어의 syntax 및 semantics를 변환가능하여, 메인 컴파일러가 지원하는 모든 고수준 언어 코드를 HDL 코드로 변환하는 방법을 제공하고자 하는 것이다.

[0008] 본 발명의 다른 목적은 소프트웨어의 메인 컴파일러가 지원하는 모든 고수준 언어 코드의 모든 syntax 및 semantics를 HDL 코드로 변환하며, 동시에 고수준 언어로된 코드에 사용하는 모든 최적화(optimization)를 활용가능한 HLL-to-HDL(High-Level Language Code to HDL Code) 변환기를 제공하고자 하는 것이다.

[0009] 본 발명의 또 다른 목적은 메인코어에서 실행되는 소프트웨어 코드와 전용하드웨어에서 실행되는 HDL 코드 사이에 cross call 및 전용하드웨어에서 실행되는 HDL 코드 사이의 재귀호출(recursive call)을 제한 없이 지원하는 컴파일(compilaton) 및 실행(execution) 프레임워크(framework)를 제공하고자 하는 것이다.

과제의 해결 수단

[0010] 상기의 목적을 달성하기 위한 본 발명의 일 실시예에 따르면 (1) 고수준 언어 코드의 변환 대상 부분(translation-targeted section)을 프로그래밍 디렉티브(programming directive)를 이용하여 표시하는 단계; (2) 표시된 상기 고수준 언어 코드의 변환 대상 부분을 하드웨어부분 코드 및 상기 고수준 언어 코드의 나머지 부분을 소프트웨어부분 코드로 분할(partition)하는 단계; (3) 상기 소프트웨어부분 코드 및 하드웨어부분 코드를 메인 컴파일러에 의해 컴파일하고, 상기 하드웨어부분 코드를 HLL-to-HDL(High-level language code to hardware description language) 변환기에 의해 HDL 코드로 변환되는 단계; 및 (4) 상기 소프트웨어 부분 코드는 메인코어에서 실행되며, 상기 HDL 코드는 전용하드웨어에서 실행되는 단계를 포함하는 고수준 언어 코드의 HDL 코드로의 변환 방법이 제공되어 질 수 있다.

[0011] 바람직한 실시예에서, 상기 (2) 단계의 상기 하드웨어부분 코드는 상기 변환대상 고수준 언어 코드의 하나 이상의 서브루틴(subroutine)/함수(function)를 포함하며, 상기 소프트웨어부분 코드는 상기 하드웨어부분 코드로의 서브루틴/함수 콜(call) 스테이트먼트(statement) 또는 콜 명령을 포함하는 것을 특징으로 한다.

[0012] 또한, 상기 (3) 단계는: 상기 소프트웨어부분 코드 및 하드웨어부분 코드를 메인 컴파일러에 의해 실행가능한 코드로 컴파일하는 단계; 상기 메인 컴파일러 링커(linker)에 의해 일정한 데이타 오프셋(constant data offset)을 포함하는 링킹(linking) 로그 파일을 생성하는 단계; 상기 메인 컴파일러에 의해 실행 가능한 소프트웨어 코드의 모든 소프트웨어 함수 및 변수의 주소를 포함하는 심벌 테이블(symbol table)을 생성하는 단계; 및 상기 HLL-to-HDL 변환기에 의해 상기 링킹 로그 파일 및 심벌 테이블을 참조하여 상기 하드웨어부분 코드의 HDL 코드를 생성하는 단계를 포함하는 것을 특징으로 한다.

[0013] 또한, 상기 HLL-to-HDL 변환기는 상기 소프트웨어부분 코드에 정의된 소프트웨어의 함수 및 변수의 주소 레솔루션(resolution)을 위해 상기 링킹 로그 파일 및 심벌 테이블을 활용하며, 상기 하드웨어부분 코드의 스트링 컨

스턴트(String constant) 등의 주소 레솔루션(address resolution)을 위해 상기 링킹 로그 파일 및 심벌 테이블을 활용하는 것을 특징으로 한다.

[0014] 또한, 상기 (3) 단계의 상기 HLL-to-HDL 변환기에 의한 상기 하드웨어부분 코드의 HDL 코드로의 변환은: 상기 하드웨어 부분 코드를 상기 메인 컴파일러의 목표 어셈블리 코드 생성 바로 이전 단계의 저수준 IR(Intermediate Representation) 코드로 변환하는 단계; 및 상기 저수준 IR 코드를 HDL 코드로 변환하는 단계를 포함하며, 상기 변환은 상기 메인코어의 표준 콜 링키지 컨벤션(standard call linkage convention)을 유지하는 것을 특징으로 한다.

[0015] 또한, 상기 고수준 언어 코드의 서브루틴(subroutine)/함수(function)로부터 생성된 상기 HDL 코드는: 타겟 어셈블리코드 생성과 동일한 방식으로 상기 저수준 IR 코드로부터 HDL로 맵핑(mapping)하여 생성되는 데이타패스(datapath); 및 상기 저수준 IR 코드의 베이직 블락들(basic blocks) 사이의 콘트롤 플로우 정보(control flow information)에 기초하여 상기 HDL 백엔드 내부의 HDL 코드 생성기(HDL code generator)에 의해 생성되며, 상기 데이타패스를 제어하는 제어 유닛(control unit)을 포함하는 것을 특징으로 한다.

[0016] 또한, 상기 고수준 언어 코드의 서브루틴/함수의 HDL로의 변환 방법은 생성된 상기 HDL 코드가 하나 이상의 블락모듈을 상기 데이타패스로서 포함하고, 탑 모듈을 상기 제어 유닛으로 포함하는 basic-block-by-basic-block 생성 방법 또는 생성된 상기 HDL 코드가 상기 데이타패스 및 상기 제어유닛 모두를 포함하는 하나의 모듈을 포함하는 function-by-function 생성 방법 중 어느 하나인 것을 특징으로 한다.

[0017] 또한, 상기 basic-block-by-basic-block 코드 생성 방법의 상기 하나 이상의 블락 모듈은 상기 저수준 IR 코드의 베이직 블락(basic blocks)과 동일한 기능을 수행하고, 상기 탑 모듈은 상기 하나 이상의 블락 모듈의 인스턴스(instances)를 생성하고, 상기 하나 이상의 블락모듈을 활성화(enabling)/비활성화(disabling)에 의해 실행순서를 정하는 것을 특징으로 한다.

[0018] 또한, 상기 basic-block-by-basic-block 코드 생성 방법의 상기 블락모듈 및 상기 Top 모듈은 유한상태기계(Finite State Machine)의 형태이고, 하나의 블락모듈은 상기 저수준 IR 코드의 베이직 블락 내부의 연속적인(consecutive) 저수준 IR 인스트럭션(instructions)의 하나의 시퀀스를 수행하는 유한상태기계의 스테이트(state) 세트(set)이며, 유한상태기계의 스테이트는 상기 저수준 IR 인스트럭션의 딜레이 추정값(delay estimation) 및 data/control 의존성에 기초한 것을 특징으로 한다.

[0019] 또한, 상기 basic-block-by-basic-block 코드 생성 방법의 상기 탑 모듈은 상기 하나 이상의 블락모듈 가운데 하나 또는 그 이상의 블락모듈이 동작하도록 할 수 있으며, 상기 블락모듈 중 하나 또는 그 이상의 블락모듈에 입력을 할당하고 작동시킨 후 작동된 상기 하나 또는 그 이상의 블락모듈의 동작이 끝날때까지 기다리고, 작동된 상기 하나 또는 그 이상의 블락모듈은 할당된 입력을 가지고 자신의 코드를 실행하고 자신의 레지스터 세트 또는 다른 블락 모듈과 공유하는 레지스터 세트에 출력을 저장(buffer)한 후 상기 Top 모듈에 제어흐름출력(Control Flow Outcome)과 함께 레지스터 세트에 저장된 상기 출력을 제공(return)하고, 상기 Top 모듈은 상기 제어흐름출력에 기초하여 상기 하나 이상의 블락모듈 가운데 다음에 실행될 블락모듈을 결정하는 것을 특징으로 한다.

[0020] 또한, 상기 function-by-function 코드 생성 방법의 상기 데이타패스 및 하드웨어 함수의 제어 유닛은 하나의 HDL 모듈에 포함되며 하나의 유한상태기계로 구성(implement)되고, 상기 데이타패스는 상기 유한상태기계의 출력 신호(output signals)를 생성하고 상기 제어 유닛은 상기 유한상태기계의 스테이트 변화(transition)를 제어하며, 상기 데이타패스는 고수준 언어의 서브루틴/함수에 대한 모든 베이직 블락으로부터 생성되고 상기 고수준 언어 코드의 서브루틴/함수의 베이직 블락의 저수준 IR 인스트럭션으로부터 생성된 유한상태기계의 하나의 세트이며, 유한상태기계의 스테이트는 상기 저수준 IR 인스트럭션의 딜레이 추정값(delay estimation) 및 data/control 의존성에 기초한 것을 특징으로 한다.

[0021] 또한, 상기 function-by-function 코드 생성 방법의 상기 제어 유닛은 고수준 언어코드의 서브루틴/함수의 제어 흐름 정보(control flow information)에 기초하여 상기 유한상태기계의 스테이트를 변화시킴으로써 상기 데이타패스를 제어하는 것을 특징으로 한다.

[0022] 또한, 상기 (3) 단계의 상기 고수준 언어 코드로부터 생성된 상기 HDL 코드는 상기 컴파일된 소프트웨어 코드와 스택(stack)을 포함한 메모리 영역을 공유하고 상기 메인코어의 소프트웨어 스택 레이아웃을 따르며, 공유하는 상기 메모리 영역은 스필드 밸류(spilled value)의 저장, 레지스터 파일 및 리턴 어드레스의 저장, 상기 하드웨어코드와 컴파일된 상기 소프트웨어부분 코드 사이의 함수 아규먼트(argument)의 전달 등에 사용되는 것을 특징

으로 한다.

[0023] 또한, 상기 (4) 단계의 전용 하드웨어는 상기 고수준 언어 코드의 서브루틴/함수로부터 각각 생성되며: 상기 HLL-to-HDL 변환기에 의해 HWID(Hardware identification number, 하드웨어식별번호)가 부여된 HWIP(Hardware Intellectual Properties) 세트; 및 상기 HWIP를 제어하는 하드웨어 컨트롤러(hardware controller)를 포함하며, 상기 HWID는 상기 소프트웨어부분 코드와 상기 하드웨어부분 코드의 심벌테이블에 기술된 타겟 어드레스 또는 상기 타겟 어드레스로부터 재계산된 고유 번호인 것을 특징으로 한다.

[0024] 또한, 상기 하드웨어 컨트롤러는 상기 HWIP들과 상기 메인코어 사이에 아규먼트(arguments)를 전달함으로써 상기 HWIP들을 제어하고, 상기 HWIP들과 메인코어 사이의 호출 제어 신호(calling control signals)를 통해 상기 HWIP들과 메인코어의 실행 순서를 제어하며, 상기 하드웨어 컨트롤러는 상기 HWIP들과 상기 메인코어에서 실행되는 소프트웨어 사이의 호출 시퀀스에 따라 하나 이상의 HWIP들 또는 메인코어를 활성화 시키며, 상기 메인코어가 상기 하드웨어 컨트롤러로 사용될 수 있는 것을 특징으로 한다.

[0025] 또한, 상기 (4) 단계의 상기 메인코어와 상기 전용 하드웨어 사이의 통신 인터페이스는 상기 메인코어와 상기 전용 하드웨어가 느슨하게(loosely) 연결(coupled)되어 상기 전용 하드웨어가 상기 메인코어와 인터럽팅(interrupting)에 의해 통신하는 소결합(loose coupling), 또는 상기 메인코어와 상기 전용 하드웨어가 단단히(tightly) 연결되어 상기 전용 하드웨어와 상기 메인 코어가 제어 신호를 서로에게 직접 주고 받을 수 있는 밀결합(tight coupling)인 것을 특징으로 한다.

[0026] 또한, 상기 (4) 단계는: 컴파일된 상기 소프트웨어부분 코드를 상기 메인코어에서 실행하는 단계; 상기 컴파일된 소프트웨어부분 코드가 실행되는 동안에 하나 이상의 상기 HWIP의 실행이 필요한 경우, 상기 메인코어에 의해 상기 전용 하드웨어의 상기 하드웨어 컨트롤러를 활성화시키고 호출(calling) 데이타를 전송하는 단계; 상기 전용 하드웨어의 상기 하드웨어 컨트롤러에 의해 대상이 되는 하나 이상의 상기 HWIP를 활성화시키고 호출 데이타를 전송하는 단계; 활성화된 상기 HWIP의 코드를 실행하고, 실행결과를 메모리와 시스템 레지스터에 저장하고, 활성화된 상기 하나 이상의 HWIP의 실행 종료를 상기 하드웨어 컨트롤러에 통지하는 단계; 상기 하드웨어 컨트롤러에 의해 활성화된 상기 하나 이상의 HWIP의 실행 종료를 상기 메인코어에게 통지하는 단계; 및 상기 메인코어에 의해 상기 업데이트된 데이타를 가지고 상기 컴파일된 소프트웨어부분 코드를 계속하여 실행하는 단계를 포함하는 것을 특징으로 한다.

[0027] 또한, 상기 메인코어로부터 상기 전용하드웨어에 전송되는 상기 호출 데이타는: 스택 포인터, 링크 레지스터, 아규먼트 레지스터 및 상기 메인코어에 의존하는 다른 레지스터를 포함하는 시스템 레지스터; 및 대상이 되는 하나 이상의 상기 HWIP의 HWID 및 활성 신호(enable signal)를 포함하는 제어 신호를 포함하는 것을 특징으로 한다.

[0028] 또한, 대상이 되는 하나 이상의 상기 HWIP의 아규먼트는 상기 메인코어로부터 시스템 레지스터의 아규먼트 및 상기 HDL 코드와 컴파일된 상기 소프트웨어 코드 사이에 공유하는 스택 스페이스를 이용하여 전송되는 것을 특징으로 하는 고수준 언어 코드의 HDL 코드로의 변환 방법.

[0029] 또한, 상기 (4) 단계의 상기 메인코어는 상기 메인코어와 상기 전용 하드웨어가 밀결합(tight coupling)의 경우에는 상기 메인코어의 ISA(Instruction Set Architecture)의 전용 인스트럭션을 이용하여 상기 전용 하드웨어와 통신하거나 제어신호를 전송하며, 상기 메인코어와 상기 전용 하드웨어가 소결합(loose coupling)의 경우에는 통신 기능을 수행하는 API 펑션 매개체를 호출(calling)함으로써 상기 전용 하드웨어와 통신하거나 제어신호를 전송하는 것을 특징으로 한다.

[0030] 또한, 상기 메인코어와 상기 전용 하드웨어가 소결합의 경우, 상기 하드웨어 컨트롤러를 활성화하는 경우 상기 메인 코어는 자신의 레지스터 값으로 하드웨어의 레지스터를 업데이트하는 것을 특징으로 한다.

[0031] 또한, 상기 하드웨어 컨트롤러에 의해 상기 전용 하드웨어의 HWIP의 어느 하나가 활성화 되는 경우, 활성화된 상기 HWIP는 상기 메인코어와 상기 전용 하드웨어가 밀결합의 경우에는 상기 메인코어로부터 데이타를 직접 읽어들이며, 상기 메인코어와 상기 전용 하드웨어가 소결합의 경우에는 상기 하드웨어 컨트롤러의 레지스터 값을 읽어들이는 것을 특징으로 한다.

[0032] 또한, 상기 전용하드웨어의 상기 하드웨어 컨트롤러는 활성화된 상기 HWIP의 실행 완료에 대해 상기 메인코어와 상기 전용 하드웨어가 소결합의 경우에는 상기 메인코어를 인터럽팅(interrupting) 함으로써 상기 메인코어에게 통지하고, 상기 메인코어와 상기 전용 하드웨어가 밀결합의 경우에는 제어 신호를 상기 메인코어에 전송함으로

써 상기 메인코어에게 통지하는 것을 특징으로 한다.

[0033] 또한, 상기 전용 하드웨어의 상기 HWIP는 상기 메인코어에서 실행 중인 컴파일된 상기 소프트웨어부분 코드로의 크로스 콜(cross call)을 수행가능하며, 상기 크로스 콜 수행은: 상기 HWIP에 의해 상기 하드에어 컨트롤러에게 컴파일된 상기 소프트웨어를 호출(call)할 필요가 있음을 알리기 위한 호출 데이타(calling data)를 준비하고 제어 신호(control signal)를 전송하는 단계; 상기 하드웨어 컨트롤러에 의해 상기 메인코어에게 상기 크로스 콜에 대해 통지하는 단계; 상기 메인코어에 의해 상기 호출 데이타를 상기 메인코어의 레지스터에 복사하고, 호출된 상기 컴파일된 소프트웨어부분 코드의 코드를 실행하고, 호출된 상기 컴파일된 소프트웨어부분 코드의 실행이 완료되는 경우 이를 상기 하드웨어 컨트롤러에 통지하는 단계; 상기 하드웨어 컨트롤러에 의해 호출된 상기 컴파일된 소프트웨어 부분 코드의 실행 완료에 대해 호출한 상기 HWIP에게 통지하는 단계; 및 상기 HWIP에 의해 호출된 상기 컴파일된 소프트웨어부분 코드의 실행 결과를 가지고 자신의 코드를 계속하여 실행하는 단계를 포함하는 것을 특징으로 한다.

[0034] 또한, 상기 HWIP로부터 상기 제어 신호(control signal)를 수신하는 경우, 상기 하드웨어 컨트롤러는 상기 메인코어와 상기 하드웨어 컨트롤러가 소결합의 경우에는 상기 메인코어를 인터럽팅(interrupting)함으로써 상기 크로스 콜에 대해 상기 메인코어에게 통지하며, 상기 메인코어와 상기 하드웨어 컨트롤러가 밀결합의 경우에는 상기 제어 신호를 상기 메인코어에게 직접 전송함으로써 상기 크로스 콜에 대해 상기 메인코어에게 통지하는 것을 특징으로 한다.

[0035] 또한, 호출된 상기 컴파일된 소프트웨어부분 코드의 실행이 완료되면, 상기 메인코어와 상기 하드웨어 컨트롤러가 소결합의 경우에는 상기 메인코어는 상기 하드웨어 컨트롤러의 시스템 레지스터 값으로 자신의 레지스터를 업데이트하고 상기 소프트웨어부분 코드의 리턴 값 및 리턴 어드레스를 상기 하드웨어 컨트롤러의 레지스터에 기입(write)하며, 상기 메인코어와 상기 하드웨어 컨트롤러가 밀결합의 경우에는 상기 메인코어는 호출하는 상기 HWIP로부터 입력 값을 직접 읽어들이는 것을 특징으로 한다.

[0036] 또한, 상기 메인코어와 상기 하드웨어 컨트롤러가 소결합의 경우에는 상기 HWIP는 호출된 상기 컴파일된 소프트웨어부분 코드의 리턴 데이타 및 리턴 어드레스를 상기 하드웨어 컨트롤러의 시스템 레지스터로부터 읽어들이며, 상기 메인코어와 상기 하드웨어 컨트롤러가 밀결합의 경우에는 상기 HWIP는 상기 컴파일된 소프트웨어부분 코드의 리턴 데이타 및 리턴 어드레스를 상기 메인코어의 레지스터 값으로부터 직접 읽어들이는 것을 특징으로 한다.

[0037] 또한, 상기 전용 하드웨어의 어느 하나의 상기 HWIP는 다른 HWIP 또는 자기 자신으로의 호출(call)을 수행가능하며, 상기 호출은: 상기 어느 하나의 HWIP에 의해 상기 하드에어 컨트롤러에게 호출(call)할 필요가 있음을 알리기 위한 호출 데이타(calling data)를 준비하고 제어 신호(control signal)를 전송하는 단계; 상기 하드웨어 컨트롤러에 의해 호출 어드레스를 검사(examining)하고 상기 다른 HWIP 또는 상기 어느 하나의 HWIP를 활성화하는 단계; 상기 다른 HWIP 또는 상기 어느 하나의 HWIP에 의해 자신의 레지스터에 상기 호출 데이타(calling data)를 복사하고, 자신의 코드를 실행하며, 실행 완료시 이를 상기 하드웨어 컨트롤러에게 통지하는 단계; 상기 하드웨어 컨트롤러에 의해 호출된 상기 다른 HWIP 또는 상기 어느 하나의 HWIP의 실행 완료에 대해 상기 어느 하나의 HWIP에 통지하는 단계; 및 상기 어느 하나의 HWIP에 의해 호출된 상기 다른 HWIP 또는 상기 어느 하나의 HWIP의 실행 결과를 가지고 자신의 코드를 계속하여 실행하는 단계를 포함하는 것을 특징으로 한다.

[0038] 또한, 상기 호출 데이타는: 스택 포인터, 링크 레지스터, 아규먼트 레지스터, 리턴 레지스터를 포함하는 시스템 레지스터; 및 호출 대상인 상기 다른 HWIP 또는 자기 자신의 어드레스 및 활성 신호(enable signal)를 포함하는 제어 신호를 포함하며, 상기 메인코어와 상기 하드웨어 컨트롤러가 소결합되어 있고 모든 상기 HWIP가 상기 하드웨어 컨트롤러 내부의 시스템 레지스터 세트를 공유하는 경우에는 호출 대상인 상기 다른 HWIP 또는 상기 어느 하나의 HWIP를 호출하기 전에 상기 어느 하나의 HWIP에 의해 상기 호출 데이타로 상기 하드웨어 컨트롤러 내부의 공유하는 상기 시스템 레지스터를 업데이트하는 것을 특징으로 한다.

[0039] 또한, 상기 다른 HWIP의 아규먼트는 아규먼트 레지스터 및 상기 HWIP들 사이에 공유하는 스택 스페이스를 통해 상기 어느 하나의 레지스터로부터 전송되는 것을 특징으로 한다.

[0040] 또한, 상기 메인코어와 상기 하드웨어 컨트롤러가 소결합의 경우에는 상기 다른 HWIP는 활성화된 경우 상기 하드웨어 컨트롤러 내부의 시스템 레지스터로 자신의 레지스터를 업데이트하며, 상기 메인코어와 상기 하드웨어 컨트롤러가 밀결합의 경우에는 상기 다른 HWIP는 활성화된 경우 상기 어느 하나의 HWIP의 레지스터 값을 직접 읽어 들이는 것을 특징으로 한다.

[0041] 또한, 상기 메인코어와 상기 하드웨어 컨트롤러가 소결합되어 있고 모든 상기 HWIP가 상기 하드웨어 컨트롤러 내부의 시스템 레지스터 세트를 공유하는 경우에는 상기 다른 HWIP는 실행 완료시 자신의 리턴 값(value) 및 리턴 어드레스를 상기 하드웨어 컨트롤러 내부의 레지스터에 기입(write)하는 것을 특징으로 한다.

[0042] *또한, 상기 메인코어와 상기 하드웨어 컨트롤러가 소결합되어 있고 모든 상기 HWIP가 상기 하드웨어 컨트롤러 내부의 시스템 레지스터 세트를 공유하는 경우에는 상기 어느 하나의 HWIP는 상기 다른 HWIP의 리턴 값(value) 및 리턴 어드레스를 상기 하드웨어 컨트롤러 내부의 레지스터로부터 읽어들이며, 상기 메인코어와 상기 하드웨어 컨트롤러가 밀결합의 경우에는 상기 어느 하나의 HWIP는 상기 다른 HWIP의 리턴 값(value) 및 리턴 어드레스를 상기 다른 HWIP의 레지스터로부터 직접 읽어들이는 것을 특징으로 한다.

[0043] 본 발명의 다른 실시예에 따르면, 프로그래밍 디렉티브로 표시된 변환대상 고수준 언어 코드로부터 상기 프로그래밍 디렉티브를 읽은 후 상기 변환대상 고수준 언어 코드를 하드웨어 부분 코드와 소프트웨어 부분 코드로 분할하는 HLL-to-HLL 소스 변환기; 상기 소프트웨어부분 코드 및 하드웨어부분 코드를 컴파일하는 메인 컴파일러; 상기 하드웨어 부분 코드를 데이타 패스 및 컨트롤 유닛을 포함하는 HDL 코드로 변환하는 HLL-to-HDL 변환기; 컴파일된 상기 소프트웨어 부분 코드를 실행하는 메인코어; 및 상기 메인코어와 연동하여 변환된 상기 HDL 코드를 실행하는 전용하드웨어를 포함하는 고수준 언어 코드의 HDL 코드로의 변환 시스템이 제공되어질 수 있다.

[0044] 바람직한 실시예에서, 상기 하드웨어부분 코드는 상기 변환대상 고수준 언어 코드의 하나 이상의 서브루틴/함수를 포함하며, 상기 소프트웨어부분 코드는 상기 변환대상 고수준 언어 코드의 나머지 부분 및 상기 하드웨어 부분 코드로의 함수 콜을 포함하는 것을 특징으로 한다.

[0045] 또한, 상기 HLL-to-HDL 변환기는 상기 메인컴파일러의 프론트 엔드(front end) 및 미들 엔드(middle end)와 HDL 백엔드(backend)를 포함하여, 상기 HLL-to-HDL 변환기는 자연적으로(natuarally) HDL 코드 생성 과정에서 컴파일된 상기 소프트웨어부분 코드의 링키지 컨벤션(linkage convention)을 따르며, 상기 HDL 백엔드는: 신택스(syntax)가 HDL 신택스와 세만틱스(semantics)를 따르며 저수준 IR 코드를 HDL로 맵핑하는 기계 묘사 파일(machine description file); 및 HDL 코드 제네레이터(generator)를 포함하는 것을 특징으로 한다.

[0046] 또한, 상기 HLL-to-HDL 변환기의 상기 메인컴파일러의 프론트 엔드 및 미들 엔드는 상기 하드웨어부분 코드를 저수준 IR 코드로 변환하며, 상기 HLL-to-HDL 변환기의상기 HDL 백엔드는 상기 저수준 IR 코드를 HDL 코드로 변환하는 것을 특징으로 한다.

[0047] 또한, 상기 고수준 언어 코드의 서브루틴/함수에 해당하는 상기 HDL 코드의 생성은, 생성된 상기 HDL 코드가 하나 이상의 블락모듈을 상기 데이타패스로서 포함하고, 탑 모듈을 상기 제어 유닛으로 포함하는 basic-block-by-basic-block 생성 방법, 또는 생성된 상기 HDL 코드가 상기 데이타패스 및 상기 제어유닛 모두를 포함하는 하나의 모듈을 포함하는 function-by-function 생성 방법 중 어느 하나인 것을 특징으로 한다.

[0048] 또한, 상기 HDL 코드의 하드웨어 함수의 상기 데이타 패스는 상기 HDL 백엔드의 기계묘사파일에 의해 상기 저수준 IR 코드로부터 직접 생성되고, 상기 HDL 코드의 하드웨어 함수의 컨트롤 유닛은 상기 HDL 코드 생성기에 의해 상기 저수준 IR 코드의 베이직 블락들 간의 컨트롤 의존성에 기초하여 생성되며, 상기 데이타 패스 및 상기 컨트롤 유닛은 유한상태기계(FSM) 형태인 것을 특징으로 한다.

[0049] 또한, 상기 basic-block-by-basic-block 생성 방법의 경우, 상기 블락모듈은 연속적인 저수준 IR 코드 인스트럭션의 시퀀스로부터 생성된 유한상태기계의 세트(set)이고, 상기 유한상태기계의 스테이트(state) 분할은 상기 저수준 IR 코드 인스트럭션의 딜레이 추정값 및 데이타/컨트롤 의존성에 기초하여 이루어지며, 상기 블락 모듈은 상기 저수준 IR 코드의 베이직 블락과 동일한 기능을 수행하며, 상기 탑 모듈은 상기 블락모듈을 활성화/비활성화 시키고, 상기 블락모듈의 실행순서를 정하는 것을 특징으로 한다.

[0050] 또한, 상기 basic-block-by-basic-block 생성 방법의 경우, 상기 Top 모듈이 한 번에 하나 또는 그 이상의 블락모듈이 동작하도록 상기 하나 이상의 블락모듈 중 하나 또는 그 이상의 블락모듈에 입력을 할당하고 활성화시킨 후, 활성화된 상기 하나 또는 그 이상의 블락모듈의 실행이 끝날때까지 기다리고, 활성화된 상기 하나 또는 그 이상의 블락모듈은 할당된 입력을 가지고 자신의 코드를 실행하고, 자신의 레지스터 세트에 출력을 저장(buffer)한 후, 상기 Top 모듈에 제어흐름출력(Control Flow Outcome)과 함께 레지스터 세트에 저장된 상기 출력을 제공(return)하고, 상기 Top 모듈은 상기 제어흐름출력에 기초하여 상기 하나 이상의 블락모듈 중 다음에 실행될 블락모듈을 결정하는 것을 특징으로 한다.

[0051] 또한, 상기 function-by-function 생성 방법의 경우, 상기 고수준 언어 코드의 서브루틴/함수에 해당하는 생성된 상기 HDL 코드의 상기 데이타 패스 및 상기 컨트롤 유닛은 하나의 모듈에 포함되고, 유한상태기계로 구현(implement)되며, 상기 데이타 패스는 상기 유한상태기계의 출력 로직(output logic)을 생성하고, 상기 컨트롤 유닛은 상기 유한상태기계의 스테이트 변이(transition)를 제어하며, 상기 데이타 패스는 상기 고수준 언어 코드의 서브루틴/함수의 모든 베이직 블락의 저수준 IR 코드 인스트럭션의 연속적인 시퀀스로부터 생성되며, 상기 유한상태기계의 스테이트(state) 분할은 상기 저수준 IR 코드 인스트럭션의 딜레이 추정값 및 데이타/컨트롤 의존성에 기초하여 이루어지는 것을 특징으로 한다.

[0052] 본 발명의 또 다른 실시예에 따르면, HLL-to-HDL 변환기를 이용하여 고수준 언어 코드를 HDL 코드로 변환하는 시스템에 있어서, 상기 HLL-to-HDL 변환기는:

[0053] 메인 컴파일러의 프론트 엔드와 미들 엔드; 및 HDL 백엔드를 포함하는 것을 특징으로 하는 HLL-to-HDL 변환기를 이용하여 고수준 언어 코드를 HDL 코드로 변환하는 시스템이 제공되어질 수 있다.

[0054] 바람직한 실시예에서, 상기 HDL 백엔드는: HDL 신택스와 세만틱스를 따르는 신택스를 가지며, 목표 어셈블리 코드(target assembly code) 생성과 동일한 방식으로 저수준 IR 코드를 HDL 코드로 맵핑하는 변환기; 및 상기 메인 컴파일러의 프론트 엔드 및 미들 엔드에 의해 상기 고수준 언어 코드로부터 상기 목표 어셈블리 코드 생성 바로 직전에 변환된 저수준 IR 코드로부터 상기 HDL 코드를 생성하는 HDL 제네레이터(generator)를 포함하는 것을 특징으로 한다.

[0055] 또한, 상기 HLL-to-HDL 변환기에 의해 상기 고수준 언어 코드의 서브루틴/함수로부터 생성된 상기 HDL 코드는 데이타 패스 및 컨트롤 유닛을 포함하고, 상기 데이타 패스 및 상기 컨트롤 유닛은 유한상태기계 형태로 구현되는 것을 특징으로 한다.

[0056] 또한, basic-block-by-basic-block 생성 방법의 경우, 상기 HDL 코드는 상기 고수준 언어 코드의 서브루틴/함수로부터 생성된 하나 또는 그 이상의 블락 모듈을 상기 데이타 패스로서 포함하고, 탑 모듈을 컨트롤 유닛으로서 포함하며, function-by-function 생성 방법의 경우, 상기 HDL 코드는 데이타 패스 및 상기 컨트롤 유닛 양자 모두를 포함하는 하나의 모듈을 포함하는 것을 특징으로 한다.

[0057] 또한, 상기 basic-block-by-basic-block 생성 방법의 경우, 상기 데이타 패스는 하나 또는 그 이상의 블락 모듈을 포함하고, 상기 블락 모듈은 베이직 블락의 저수준 IR 코드 인스트럭션으로부터 생성되는 유한상태기계 세트이며, 상기 function-by-function 생성 방법의 경우, 상기 데이타 패스는 상기 고수준 언어 코드의 서브루틴/함수의 모든 베이직 블락의 저수준 IR 코드 인스트럭션으로부터 생성되는 유한상태기계 세트이며, 상기 두 가지 방법 모두에 있어, 상기 유한상태기계의 스테이트(state) 분할은 상기 저수준 IR 코드 인스트럭션의 딜레이 추정값 및 데이타 의존성에 기초하여 이루어지는 것을 특징으로 한다.

발명의 효과

[0058] 본 발명에 따른 고수준 언어 코드의 HDL 코드로의 변환 시스템 및 방법은 변환 대상 고수준 언어 코드에 대한 수정없이도 모든 고수준 언어의 syntax 및 semantics를 변환가능하여, 메인 컴파일러가 지원하는 모든 고수준 언어 코드를 HDL 코드로 변환가능한 유리한 효과가 있다.

[0059] 또한, 모든 고수준언어의 syntax 및 semantics에 사용가능하여, 메인 컴파일러가 지원하는 모든 고수준 언어 코드를 HDL 코드로 변환가능하고, 메인 컴파일러를 기본 컴파일러로 사용함으로써 고수준 언어로된 코드에 사용하는 모든 최적화(optimization) 를 사용가능한 유리한 효과가 있다.

[0060] 또한, 전용하드웨어에서 실행되는 변환된 HDL 코드와 메인코어에서 실행되는 소프트웨어 코드 사이의 크로스 콜 및 하드웨어 내부에서의 재귀호출(recursive call)을 통해 HDL코드와 소프트웨어 코드가 연동되도록 함으로써 응용프로그램의 속도를 향상시킬 수 있는 유리한 효과가 있다.

도면의 간단한 설명

[0061] 도 1은 본 발명의 일 실시예에 따른 고수준 언어 코드의 HDL 코드로의 변환 시스템의 구조도.

도 2는 본 발명의 일 실시예에 따른 고수준 언어 코드의 HDL 코드로의 변환하는 과정을 나타내는 순서도.

도 3은 본 발명의 일 실시예에 따른 변환대상 언어가 C 언어인 경우에 있어서, 본 발명의 시스템의 구성을 나타내는 예시도.

도 4는 본 발명의 다른 실시예에 따른 변환대상 언어가 C 언어인 경우의 시스템의 구성을 나타내는 예시도.

도 5는 본 발명의 일 실시예에 따른 HLL-to-HDL 변환기에 의해 하드웨어부분 코드를 HDL 코드로 변환하는 과정을 나타내는 순서도.

도 6은 본 발명의 일 실시예에 따른 basic-block-by-basic-block방법을 이용한 경우의 데이타 패스로서의 HDL 블락 모듈과 컨트롤 유닛으로서의 탑 모듈 사이의 제어 흐름 및 통신을 나타내는 블락 다이어그램.

도 7은 본 발명의 일 실시예에 따른 basic-block-by-basic-block방법을 이용한 경우의 저수준 IR 코드의 베이직 블락으로부터 HDL 코드의 블락 모듈의 코드 생성을 나타내는 블락 다이어그램.

도 8은 본 발명의 일 실시예에 따른 basic-block-by-basic-block방법을 이용한 경우의 컨트롤 유닛으로서의 탑 모듈의 Verilog HDL 코드를 나타내는 예시도.

도 9는 본 발명의 일 실시예에 따른 basic-block-by-basic-block방법을 이용한 경우의 컨트롤 유닛과 블락 모듈의 인스턴스를 포함하는 Verilog HDL 코드의 탑 모듈을 나타내는 예시도.

도 10은 본 발명의 일 실시예에 따른 function-by-function방법을 이용하여 C 함수로부터 생성된 데이타 패스 및 컨트롤 유닛을 나타내는 블락 다이어그램.

도 11은 본 발명의 일 실시예에 따른 메인 코어와 전용 하드웨어 사이의 커뮤니케이션을 나타내는 블락 다이어그램.

도 12는 본 발명의 일 실시예에 따른 메인 코어에서 실행되는 소프트웨어부분 코드와 전용 하드웨어에서 실행되는 하드웨어부분 코드 사이의 콜링 프로시져(calling procedure)를 나타내는 순서도.

도 13은 본 발명의 일 실시예에 따른 메인코어와 전용하드웨어가 소결합인 경우의 소프트웨어부분 코드와 하드웨어부분 코드 사이의 크로스 콜을 지원하기 위한 SW/HW 인터페이스 및 하드웨어 컨트롤러를 나타내는 블락 다이어그램.

도 14는 본 발명의 일 실시예에 따른 메인코어와 전용하드웨어가 소결합인 경우의 메인코어에서 실행되는 소프트웨어부분 코드와 전용하드웨어에서 실행되는 하드웨어부분 코드 사이의 크로스 콜을 수행하는 메카니즘을 나타내는 예시도.

도 15는 본 발명의 일 실시예에 따른 메인코어와 전용하드웨어가 소결합인 경우의 도 14의 메인코어와 전용하드웨어 사이의 크로스 콜의 예시 코드에 대한 크로스 콜 임플리멘테이션을 나타내는 예시도.

도 16은 본 발명의 일 실시예에 따른 메인코어와 전용하드웨어가 밀결합인 경우의 메인코어와 전용하드웨어 사이의 커뮤니케이션 인터페이스를 나타내는 블락 다이어그램.

도 17 및 18은 본 발명의 일 실시예에 따른 메인코어와 전용하드웨어가 밀결합인 경우의 도 16의 크로스 콜 임플리멘테이션을 나타내는 블락 다이어그램.

발명을 실시하기 위한 구체적인 내용

[0062] 이하 첨부한 도면들을 참조하여 본 발명의 바람직한 실시예를 상세히 설명하기로 한다.

[0063] 도 1은 본 발명에 따른 고수준 언어 코드의 HDL 코드로의 변환 시스템의 구조도이다.

[0064] 도 1을 참조하면, 본 발명의 고수준 언어 코드의 HDL 코드로의 변환 시스템은 HLL-to-HLL 소스 변환기(110), GCC(GNU Compiler Collection)와 같은 메인 컴파일러(120), 변환대상 고수준 언어 코드를 Verilog와 같은 HDL로 변환하는 HLL-to-HDL 변환기(130), ARM(Advanced RISC Machine)과 같은 메인코어(140) 및 FPGA(Field Programmable Gate Array)와 같은 전용하드웨어(150)를 포함함을 알 수 있다.

[0065] 상기 HLL-to-HLL 소스 변환기(110)는 고수준 언어 코드의 변환대상 코드 섹션에 프로그래밍 디렉티브(programming directive)에 의한 표시가 있는 경우, 상기 프로그래밍 디렉티브를 읽어 들인 후 상기 변환대상 고수준 언어 코드를 하드웨어 부분 코드와 소프트웨어 부분 코드로 분할(partition)한다. 이때, 상기 하드웨어 부분 코드는 상기 변환대상 고수준 언어 코드 가운데 특정 또는 다수 프로그램에서 되풀이하여 사용되는 독립된 명령군인 서브루틴/함수 부분이고, 상기 소프트웨어 부분 코드는 상기 변환대상 고수준 언어 코드의 나머지 부분 및 상기 하드웨어 부분 코드로의 함수 콜을 포함한다.

[0066] 상기 메인 컴파일러(120)는 상기 HLL-to-HLL 소스 변환기(110)에 의해 파티션된 상기 소프트웨어부분 코드 및 하드웨어부분 코드를 함께 컴파일하여 실행가능한(executable) 코드로 변환하는 역할을 한다.

[0067] 상기 HLL-to-HDL 변환기(130)는 상기 HLL-to-HLL 소스 변환기(110)에 의해 파티션된 상기 하드웨어 부분 코드를 HDL 코드로 변환하는 역할을 한다.

[0068] 상기 메인코어(140) 및 상기 전용하드웨어(150)는 각각 상기 메인 컴파일러(120)에 의해 컴파일된 소프트웨어부분 코드와 상기 변환기(130)에 의해 변환된 HDL 코드를 서로 협력하여 실행하는 역할을 한다.

[0069] 도 2는 본 발명의 일 실시예에 따른 고수준 언어 코드의 HDL 코드로의 변환하는 과정을 나타내는 순서도이다.

[0070] 도 2를 참조하면, 본 발명에 따른 고수준 언어 코드의 HDL 코드로의 변환 방법은 변환대상 고수준 언어 코드를 프로그래밍 디렉티브로 표시하는 단계(S210), 표시된 상기 코드를 하드웨어부분 코드와 소프트웨어부분 코드로 파티션하는 단계(S220), 상기 소프트웨어 부분 코드 및 하드웨어부분 코드를 메인 컴파일러로 함께 컴파일하는 단계(S230), 상기 하드웨어부분 코드를 HDL 코드로 변환하는 단계(S240), 컴파일된 상기 소프트웨어부분 코드를 메인코어에서 실행하는 단계(S250) 및 변환된 상기 HDL 코드를 전용하드웨어에서 실행하는 단계(S260)를 포함함을 알 수 있다. 이때, 상기 메인코어와 상기 전용하드웨어는 서로 연동하여 실행된다.

[0071] 도 3 및 도 4는 본 발명의 두 가지 실시예에 따른 변환대상 언어가 C 언어인 경우에 있어서, 본 발명의 시스템의 구성을 나타내는 예시도이다.

[0072] 도 3 및 도 4를 참조하면, 앞서 설명한 바와 같이, 먼저 고수준 프로그래밍 언어인 C 언어로 된 변환 대상 코드의 변환 대상 섹션에 프라그마와 같은 프로그래밍 디렉티브를 이용하여 표시한다. 이때, 표시된 코드 부분은 루프(도면 내의 #pragma hardware for 부분), 서브루틴(#pragma hardware 섹션), 또는 #pragma hardware로 표시한 변환대상 코드 내의 어느 부분(섹션)이라도 될 수 있다. 이 후 상기 HLL-to-HLL 소스 변환기(110)는 프로그래밍 디렉티브로 표시된 고수준 언어 코드를 읽은 후 상기 프로그래밍 디렉티브로 표시된 코드 섹션을 서브루틴으로 변환한다. 상기 표시된 코드 섹션 이외의 고수준 언어 코드의 나머지 부분이 상기 소프트웨어부분 코드에 해당한다. 상기 하드웨어부분 코드 및 소프트웨어부분 코드는 하나의 공통된 파일에 저장될 수 있으며, 또는 각각 다른 파일에 저장되어질 수 있다. 생성된 하드웨어 함수는 전위표기(prefix) hw_func_과 HWID 번호(hardware identification number)로 명명된다(도 4의 hw_func_1 참조). 발췌된(extracted) 코드 섹션은 래퍼(wrapper)라고 이름붙여진 API 함수 콜로 대체된다. 하드웨어 코드는 소프트웨어와 달리 어드레스가 없기 때문에, 고유한 하드웨어 아이덴티피케이션 넘버인 HWID가 각각의 하드웨어 함수에 부여되며, 부여된 HWID는 생성된 하드웨어 함수를 식별하는데 사용된다.

[0073] 상기 코드에 대한 전처리 과정이 완료된 후, 상기 소프트웨어부분 코드와 상기 하드웨어부분 코드는 GCC 컴파일러와 같은 메인 컴파일러 120에 의해 함께 컴파일되며, 상기 하드웨어 함수는 상기 HLL-to-HDL 변환기(130)에 의해 상응하는 Verilog 코드로 변환된다.

[0074] 일례로, 일반적인 변수 및 함수인 arr[10] 및 printf가 소프트웨어코드에 선언(declare)된 경우, 이들은 hw_func_1로 하드웨어에서 엑세스(access)된다. 이러한 경우, 소프트웨어부분과 하드웨어부분이 함께 구성되지 않는다면, 하드웨어는 상기 일반적인 변수(global variable) 심벌의 어드레스를 알 수 없다. 심벌 테이블은 모든 글로벌 변수 및 함수의 어드레스뿐만 아니라 데이타 섹션의 베이스 어드레스도 포함한다. 그러므로, 상기 글로벌 심벌의 어드레스 레솔루션(resolution)을 위해 상기 심벌 테이블을 이용하여 상기 하드웨어부분 코드와 소프트웨어부분 코드를 함께 컴파일한다.

[0075] 그러나, 상기 하드웨어부분 코드의 "arr[%d]=%d"와 같은 컨스턴트 스트링(constant string)은 변수가 아니고 심벌 테이블에 포함되지 않기 때문에, 이들은 ".rodata" 섹션에 저장된다. 컨스턴트의 어드레스는 3 변수의 합으로 계산된다. 즉, 컨스턴트의 어드레스는 상기 심벌 테이블의 ".rodata" 섹션의 시작 어드레스, 상기 HLL-to-HDL 변환기의 변환대상인 인풋 소스파일의 컨스턴트 오프셋 어드레스(constant's offset address) 및 본 발명에 따라 수정된 GCC 링커의 파일 오프셋 어드레스를 통해 정해진다.

[0076] 소프트웨어의 함수가 소프트웨어가 아닌 하드웨어에 의해 실행되기 때문에 메인컴파일러는 상기 소프트웨어부분 코드와 하드웨어부분 코드를 함께 컴파일한다. 또한, 상기 소프트웨어부분 코드의 컴필레이션(compilation) 과정의 코드 최적화(optimization)에서 소프트웨어에서 실행될 필요가 없는 경우 상기 함수 부분은 제외되어질 수 있다. 즉, 데드코드 제외(deadcode elimination)가 이루어질 수 있다. 일례로, 라이브러리 함수인 "printf"는 하드웨어부분 코드의 "hw_func_1" 함수로부터의 호출(call)을 위해 소프트웨어부분 코드에 포함되어 질 수

있다. 또한, 하드웨어 컴필레이션도 컨스턴트 오프셋 정보를 올바르게 계산하기 위해 전체 소스 코드를 컴파일한다. 그러나, 오직 하드웨어 함수만이 Verilog와 같은 HDL 코드로 변환되며, 나머지 섹션은 HLL-to-HDL 변환기에 의해 무시되며 변환되지 않는다.

[0077] 생성된 HDL 코드는 하드웨어 컴파일러에 의해 합성되며(synthesized) FPGA와 같은 전용 하드웨어(150)에서 실행되며, 반면에 컴파일된 소프트웨어부분 코드는 ARM과 같은 메인코어(140)에서 실행된다.

[0078] 도 5는 본 발명의 일 실시예에 따른 HLL-to-HDL 변환기에 의해 하드웨어부분 코드를 HDL 코드로 변환하는 과정을 나타내는 순서도이다.

[0079] 도 5를 참조하면, 고수준 언어로된 하드웨어 부분 코드를 Verilog와 같은 HDL 코드로 변환하는 과정은 고수준 언어 코드를 RTL 코드와 같은 저수준 IR(Intermediate Representation)로 변환하는 단계(S410) 및 목표 어셈블리 코드(targeted assembly code) 생성 바로 이전 단계의 상기 RTL과 같은 저수준 IR 코드로 부터 Verilog와 같은 HDL 코드를 생성하는 단계(S420)로 이루어 짐을 알 수 있다.

[0080] 상기 저수준 IR(Intermediate Representation)로 변환하는 단계(S410) 및 목표 어셈블리 코드(targeted assembly code) 생성 바로 이전 단계의 상기 RTL과 같은 저수준 IR 코드로 부터 Verilog와 같은 HDL 코드를 생성하는 단계(S420)는 GCC와 같은 메인 컴파일러 프론트 엔드 및 미들 엔드와 HDL 백엔드를 포함하는 HLL-to-HDL 변환기에 의해 수행된다.

[0081] 상기 하드웨어부분 코드를 GCC의 RTL과 같은 저수준 IR 코드로 변환하는 단계(S410)는 상기 HLL-to-HDL 변환기의 메인 컴파일러 프론트 엔드 및 미들 엔드에 의해 이루어지며, 상기 저수준 IR 코드로부터 Verilog와 같은 HDL코드를 생성하는 단계는 상기 HLL-to-HDL 변환기의 HDL 백엔드에 의해 이루어진다. 이때, 상기 HLL-to-HDL 변환기는 다른 백엔드에 의해 타겟 어셈블리 코드 생성에 사용되기 바로 직전의 저수준 IR 코드를 활용하여 상기 HDL 코드를 생성한다. 이때 저수준 IR 코드는 입력 고수준 언어와는 무관하기 때문에, 상기 HLL-to-HDL 변환기는 메인 컴파일러가 지원하는 모든 고수준 언어로된 코드를 HDL 언어로 변환할 수 있는 것이다.

[0082] 이하 상기 RTL과 같은 저수준 IR로부터 Verilog와 같은 HDL 코드가 생성되는 방법을 상세히 설명하기로 한다.

[0083] 메인 컴파일러로서 GCC 컴파일러가 사용되고, 변환된 HDL이 Verilog인 경우를 예로 설명하면, GCC 컴파일러의 미들엔드 부분은 알고리즘을 어셈블리 언어로 변환하기 전에 상기 알고리즘을 최적화하기 위해 RTL로 불려지는 IR(Intermediate Representation)을 사용한다. GCC의 RTL은 "*.md" 와 같은 기계묘사파일(machine description file)에 의해 매개변수로 나타내어지는(parameterized) 추상적 묘사(abstract description)이다. GCC의 상기 기계묘사파일 "*.md"는 함수 유닛(functional unit), 레지스터 파일 및 인스트럭션 세트 아키텍쳐(instruction set architecture)와 같은 목적 프로세서에 대해 정의한다. 본 발명에 따른 HDL 백엔드는 상기 저수준 RTL의 Verilog 코드로의 변환을 용이하게 하기 위해 인스트럭션 패턴이 Verilog 신택스를 따르도록 구성되어진다.

[0084] RTL의 베이직 블락을 면밀히 검토하면, RTL과 Verilog가 많은 유사점을 가지고 있음을 알 수 있다. RTL의 경우, C 포인터 및 복잡한 데이타 구조에 대한 모든 메모리 액세스는 단순한 메모리 액세스로 단순화되고(convert) 모든 종류의 컨트롤 스테이트먼트는 브랜치(branches) 및 점프 인스트럭션으로 변환된다. 또한, RTL은 three-address 코드를 따르며, 이는 Verilog 스테이트먼트로 쉽게 변환이 가능하다. 모든 RTL의 오퍼레이터(operator) 또한 Verilog의 오퍼레이터에의해 표현이 가능하고, RTL 베이직 블락의 live-in 및 live-out 레지스터는 Verilog 모듈의 input/output 데이타 형식으로 변환이 가능하다. Verilog의 블락 모듈은 input/output 데이타를 통해 서로서로 통신(communication)이 가능하다. 나아가, RTL에서의 레지스터의 사용은 Verilog의 reg 데이타 변수(reg data variables)에 대응한다.

[0085] 소프트웨어 RTL의 부재타이밍 컨트롤(absent timing control) 및 병행성(concurrency) 문제는 유한 상태 기계(FSM, finite state machine)의 도입으로 해결되어질 수 있다. 유한 상태 기계는 하드웨어에서 병행 실행(concurrent execution)을 활용(exploit)하면서도 소프트웨어의 실행 시퀀스를 유지할 수 있다. 유한 상태 기계의 각각의 스테이트는 프로그램 어드레스와 같이 고유하게 번호 붙여지고, 하나의 스테이트는 한 클락 싸이클(one clock cycle)내의 실행을 나타낸다. 하나의 유한 상태 기계 스테이트에는 하나 또는 그 이상의 RTL 인스트럭션이 실행될 수 있으며, 또는 시간이 걸리는 RTL 인스트럭션(time-consuming RTL instruction)의 경우에는 복수의 유한 상태 기계 스테이트에 걸쳐 실행되어질 수 있다. 또한, 다수 개의 인스트럭션이 서로 독립적인 경

우에는 하나의 스테이트 내에 다수 개의 인스트럭션이 병렬적으로 실행되어질 수 있다. 병렬적으로 실행되는 인스트럭션의 수가 많을수록 FPGA와 같은 전용하드웨어의 실행속도는 더 빨라진다. 그러나, 함수 콜, 곱셈, 나눗셈 및 메모리 엑세스와 같은 몇몇 인스트럭션은 하나의 스테이트 내에 실행될 수 없으며, 따라서, HDL 백엔드는 다수 개의 유한상태기계의 스테이트에 걸쳐 실행될 수 있도록 코드를 생성한다. 실행 중인 스테이트에서의 모든 인스트럭션이 실행된 후, 스테이트 변수(예로써, 프로세서의 PC, program counter 등)는 다음 스테이트 번호로 맞추어 진다. non-blocking 어싸인먼트인 "<="가 생성된 코드에 사용된다.

[0086] I/O기기 및 메모리 시스템을 제외한 일반적인 하드웨어 시스템은 데이타 패스(레지스터, 함수 유닛 등)와 상기 데이타 패스를 제어하기 위한 컨트롤 유닛을 포함한다. 따라서, 변환된 HDL 코드의 서브루틴은 하나의 데이타 패스 및 하나의 컨트롤 유닛을 포함하도록 HLL-to-HDL 변환기에 의해 변환된다. 상기 데이타 패스는 하나 이상의 Verilog 블락 모듈로써 구현되고, 상기 컨트롤 유닛은 하나의 Verilog 컨트롤 모듈로 구현된다. 하나의 베릴로그 블락 모듈은 하나의 RTL 베이직 블락으로부터 구성되어진다.

[0087] 도 6은 본 발명의 일 실시예에 따른 basic-block-by-basic-block방법을 이용한 경우의 데이타 패스로서의 HDL 블락 모듈과 컨트롤 유닛으로서의 탑 모듈 사이의 제어 흐름 및 통신을 나타내는 블락 다이어그램이다.

[0088] 도 6을 참조하면, GCC에서 함수의 RTL 코드는 수 개의 기본 블락들(basic blocks)로 나누어지고, 각각의 기본 블락은 단지 하나의 엔트리 포인트(entry point) 및 하나의 엑시트 포인트(exit point)를 갖는다. 따라서 코드의 제어의 흐름은 기본블락의 엔드 부분에서 정하여 진다. 즉, 현재 실행 중인 기본블락 내지 전처리 블락이 자신 다음에 실행될 기본블락 내지 후처리 블락을 지정하는 것이다. 전처리기와 후처리기 사이의 입력 및 출력 정보는 레지스트 파일이다. 이때, 후처리기의 실행에 있어서는 전처리기에 의해 업데이트된 레지스터 파일이 사용된다.

[0089] 또한, 도 6을 참조하면, 본 발명의 HDL 모듈은 RTL과 같은 저수준 IR 코드의 기본블락의 역할과 동일한 역할을 수행하는 블락모듈들(560 내지 590)과 블락모듈의 실행을 제어하고 그 실행 순서를 정하기 위한 Top 모듈(550)을 포함할 수 있음을 알 수 있다. 저수준 IR 코드의 레지스터 파일은 상기 블락모듈의 입력(input) 및 출력(output) 시그널로 구현(implement) 된다. 상기 Top 모듈(550)은 복수의 여러 블락모듈 사이의 실행을 제어한다. 즉, 상기 Top 모듈은 블락 모듈에 입력을 할당하여 블락모듈을 활성화시키며, 상기 활성화된 블락모듈은 할당된 입력을 가지고 자신의 코드를 실행하고, 자신의 레지스터 세트 또는 다른 블락모듈과 공유하는 레지스터 세트에 출력을 저장(buffer)한 후 상기 Top 모듈(550)에 제어흐름출력(control flow outcome)과 함께 레지스터 세트에 저장된 출력을 제공(return)한다. 이후 상기 Top 모듈(550)은 상기 제어흐름출력에 기초하여 다음에 실행될 블락모듈을 결정한다. 이와 같은 방식(mechanism)을 이용하여 블락모듈에 의한 데이터경로와 Top 모듈에 의한 제어경로가 분리될 수 있다.

[0090] 일례로, 도 6의 블락 모듈 0(560) 또는 블락 모듈 2(580)와 같이 브랜치 인스트럭션을 포함하는 블락모듈은 다음에 실행될 블락 모듈의 결정을 위해 branch_result(예로, 비교명령의 결과)를 컨트롤 모듈에 해당하는 탑 모듈(550)에 전송한다. 이후 상기 탑 모듈은 enable_bb 와 같은 신호를 통해 다음에 실행될 블락 모듈을 활성화시키고 활성화된 블락 모듈이 input_select 시그널을 통해 자신 이전에 실행된 블락 모듈로부터 올바를 입력을 선택하는 것을 돕는다.

[0091] 상기 블락모듈의 바디코드(body code)는 본 발명의 기계묘사파일(machine description file)에 의해 RTL과 같은 저수준 IR 코드로부터 직접(directly) 생성된다. 이와 달리 Top 모듈의 바디코드는 저수준 IR 코드의 기본블락들의 정보에 기초한 본 발명의 HDL 제어 생산부(HDL control generator)에 의해 생성되며, 본 발명의 HLL-to-HDL 변환기의 출력은 수 개의 블락모듈과 하나의 Top 모듈을 하나의 포함한다.

[0092] 이하 본 발명의 basic-block-by-basic-block 방법의 블락모듈을 위한 HDL 코드 생성에 대해 설명하기로 한다.

[0093] 도 7은 본 발명의 일 실시예에 따른 basic-block-by-basic-block방법을 이용한 경우의 저수준 IR 코드의 베이직 블락으로부터 HDL 코드의 블락 모듈의 코드 생성을 나타내는 블락 다이어그램이다. 참조 부호 ① 내지 ⑥은 어셈블리 코드, RTL 인스트럭션 및 이에 상응하는 Verilog 코드의 일대일 대응을 표현하기 위해 사용되었다. 유한상태기계의 각 스테이트들에는 번호가 부여되어있으며, 그들은 case 스테이트먼트에 보여지고 있다.

[0094] 모든 블락 모듈 및 탑 모듈은 Verilog 컴비네이션 "always" 블락을 이용한 유한상태기계(FSM, Finite State

Machine)의 형태로 생성된다. 도 7을 참조하면, 인스트럭션 시퀀스는 각각이 유한상태기계의 하나의 단계에 해당하도록 수 개의 부분으로 분리되었으며, 현재 실행 중인 단계의 모든 명령어가 실행된 후 단계 변수가 이후 실행될 단계 번호가 되도록 하였음을 알 수 있다.

[0095] 각각의 FSM 단계는 하나의 클락 사이클 이내에 실행되며, 따라서 하나의 FSM 단계는 하나 또는 다수 개의 저수준 IR 명령어를 실행할 수 있다. 하나의 단계에서 실행되는 명령어의 수가 많을 수록 FPGA와 같은 전용하드웨어의 실행속도는 더 빨라진다. 명령어들이 하나의 단계 내에서 실행될 수 있기 위한 조건은 바로 명령어들이 병렬적이어야 한다는 것이다. 즉, 명령어들이 서로 독립적이어야 하는 것이다. 본 발명의 HLL-to-HDL 변환기는 명령어들의 딜레이 추정값(delay estimation) 및 데이터 의존성에 기초하여 저수준 IR 명령어들을 어떻게 분할할 것인지를 결정한다. 덧셈, 뺄셈, 배분(assignment), 행 이동(shift) 및 연산(logic operation)은 시간이 걸리는 작업이 아니므로 하나의 주기(cycle) 안에 실행될 수 있는 것으로 고려되었으며, 이러한 시간이 걸리지 않는 명령어들은 병렬적이기만 하다면 하나의 FSM 단계 내에 다수 개가 포함될 수 있다. 그러나, 곱셈, 나눗셈 및 메모리 엑세스는 시간이 걸리는 작업에 해당하고, 따라서 이들에 기초한 명령어들은 곱셈, 나눗셈 및 메모리 엑세스 등이 끝날 때 까지 기다린 후 다음 주기에 실행된다.

[0096] 예로써 도 7의 HDL 블락 모듈이 활성화된 경우(코드의 시작부분에서 활성화됨), 스테이트 변수 "pc"는 스테이트 0으로 세트되며 레지스터 regs9 및 regs10은 전처리기로부터 입력받은 입력 값(live-in value) "*_in_4"로 초기화된다. 상기 변수 "pc"는 다음에 실행될 스테이트를 나타내며, 이는 일반 프로세서의 프로그램 카운터(program counter)와 유사하다.

[0097] 도 7을 참조하면, 메모리 로드(load) 인스트럭션은 두 스테이트에 걸쳐 실행된다. 즉, 스테이트 0에서 메모리 서비스를 요청하고, 스테이트 1에서 메모리 스테이터스(status)를 체크하고 메모리로부터 데이타를 수령한다. 3번째 스테이트의 명령은 덧셈으로 이는 4번째 스테이트의 명령인 메모리 저장과 병렬적으로 실행되어질 수 있다. 이는 이들 명령이 서로 독립적이기 때문이다. 상기 메모리 저장 인스트럭션은 스테이트 3(4번째 스테이트)의 메모리 초기화(initialization)과 스테이트 4의 메모리 스테이터스 체크의 두 개의 스테이트로 분할되어 있다.

[0098] 마지막으로 스테이트 5에서 브랜치 출력(outcome) 신호는 "regs10"의 값으로 결정되며 종료 신호(finish signal)는 컨트롤 모듈인 탑 모듈에게 실행이 종료되었음을 알리기 위해 세트(set)된다.

[0099] 다음으로, basic-block-by-basic-block 방법의 탑 모듈의 HDL 코드 생성에 대해 설명하기로 한다.

[0100] 도 8은 본 발명의 일 실시예에 따른 basic-block-by-basic-block방법을 이용한 경우의 컨트롤 유닛으로서의 탑 모듈의 Verilog HDL 코드를 나타내는 예시도이다. 탑 모듈은 본 발명의 HLL-to-HDL 변환기에 의해 블락 모듈의 제어와 실행 순서(schedule)를 정하기 위해 생성된다. 상기 탑 모듈도 고유한 번호를 갖는 스테이트들로 이루어진 유한상태기계의 형식으로 이루어져 있다. 탑 모듈은 블락 모듈을 활성화(enable)/비활성화(disable) 시키기 위해 enable_bb 시그널을 사용한다. 블락 모듈은 자신의 HWID(hardware identification number)에 상응하는 비트 포지션(bit position)이 세트되면 활성화된다. 예를 들어, "enable_bb <= 4'b0001"는 4개의 블락 모듈이 이에 의해 제어되며 블락 모듈 0이 활성화됨을 나타낸다. 블락 모듈이 활성화된 경우, 블락 모듈은 자신의 전처리기인 블락 모듈로부터의 입력 값을 가지고 자신의 바디코드(body code)를 실행하고, 자신의 다음에 실행될 블락 모듈의 입력 값에 해당하는 출력 값을 레지스터 세트에 저장(buffer)한다. 만일, 블락 모듈이 조건문 브랜치(conditional branch)로 완료되는 경우, 실행완료 시에 석세서(successor) 신호를 통해 브랜치 출력을 탑 모듈에 리턴한다. 탑 모듈은 상기 브랜치 출력을 고려하여 다음에 실행될 블락 모듈(successor block module)을 결정한다.

[0101] 도 9는 본 발명의 일 실시예에 따른 basic-block-by-basic-block방법을 이용한 경우의 컨트롤 유닛과 블락 모듈의 인스턴스를 포함하는 Verilog HDL 코드의 탑 모듈을 나타내는 예시도이다.

[0102] Top 모듈의 주요 임무 두 가지는 블락모듈에게 입력(input)/출력(output)을 할당하는 것과 블락모듈의 실행 순서(schedule)를 정하는 것이다. 입력/출력 신호를 할당하기 위해 Top 모듈은 블락모듈의 인스턴스(instance)를 생성한다. 이는, q블락 모듈은 모듈을 정의하고 바로 사용할 수 없고, 단지 구조적 기능적 정의일 뿐이므로, 정의된 모듈을 사용하기 위해서는 instantiation이 필요하기 때문이다.

[0103] Top 모듈은 현재 실행된 블락모듈의 출력을 다음에 실행될 블락모듈에게 입력으로 할당한다. 블락모듈들 간의 전처리리와 후처리기(predecessor/successor) 사이 관계는 RTL 기본블락들 간의 전처리기와 후처리기 사이의 관

계와 같다.

[0104] Top 모듈에는 블락모듈들의 실행 순서를 정하기 위한 제어부가 구비된다. 상기 제어부도 FSM 형식으로 구성되며(implement), 따라서 다음과 같은 기능을 하게 된다. 즉, 작동활성화 신호(enable signals)를 세팅(setting)함으로써 블락모듈을 활성화/비활성화(enable/disable) 시키며, 중단신호(interrupt signal)를 생성하여 프로그램(소프트웨어)으로 하여금 실행종료를 나타내도록 한다.

[0105] 도 9를 참조하면, 상기 설명한 바와 같이, Top 모듈은 제어부를 포함하고 있으며, 변수를 선언하고, 블락모듈에 입력신호를 할당하며, 블락모듈 인스턴스를 생성하는 것을 알 수 있다. 또한, 상기 제어부는 FSM 형식을 취하고 있음을 알 수 있다.

[0106] 다음으로, 본 발명의 일 실시예에 따른 function-by-function 방법의 고수준 언어 코드의 서브루틴의 Verilog 코드 생성에 대해 설명하기로 한다.

[0107] 도 10은 본 발명의 일 실시예에 따른 function-by-function방법을 이용하여 C 함수로부터 생성된 데이타 패스 및 컨트롤 유닛을 나타내는 블락 다이어그램이다. 도 6 내지 8을 참조하여 앞서 설명한 저수준 IR 코드의 각각의 베이직 블락을 HDL 블락 모듈로 생성하는 basic-block-by-basic-block 코드 생성의 경우, 생성된 HDL 코드를 쉽게 제어하고 디버그(debug)할 수 있다. 그러나, basic-block-by-basic-block 방법에 의해 생성된 HDL 코드는 각각의 블락 모듈에 대한 별개의 레지스터 파일이 임플리먼트되어야 하므로 많은 리소스(resource)를 소비하게된다. 따라서, 하드웨어 함수의 베이직 블락의 수가 많아지는 경우, 생성된 HDL 코드는 큰 수자의 레지스터를 필요로하게되며, 이에 따라 FPGA와 같은 전용 하드웨어의 리소스를 고갈되게 할 수 있다.

[0108] *따라서, 리소스의 소비를 줄이기 위해, 고수준 언어 코드의 전체 저수준 IR 코드를 제네럴 데이타패스(general datapath)로 변환하는 function-by-function 방법이 활용될 수 있다. 즉, 전체 고수준 언어 코드의 함수는 오직 하나의 일반 레지스터 파일(global register file)을 활용한다. C 코드 및 RTL 제어 흐름이 도 10의 (a) 및 (b)에 나타내어져 있으며, 컨트롤 유닛과 데이타 패스를 포함하는 C 코드 및 RTL 제어 흐름에 해당하는 변환된 Verilog 코드는 도 10의 (c) 및 (d)에 각각 나타내어져 있다.

[0109] 도 10의 (c) 및 (d)를 참조하면, 상기 데이타 패스 및 컨트롤 유닛은 하나의 유한상태기계(FSM, finite state machine)으로 구성됨을 알 수 있다. 상기 컨트롤 유닛은 유한상태기계의 스테이트 변수 "nx_pc" 값을 변경시키며, 상기 데이타 패스는 유한상태기계의 출력을 생성하기 위한 레지스터 파일과 함수 유닛(functional unit)을 포함하고 있다. 본 발명에 따른 HLL-to-HDL 변환기는 하나의 베이직 블락을 유한상태기계의 스테이트 블락으로 변환시키고, 모든 베이직 블락의 스테이트는 글로벌 데이타 패스를 형상하기 위해 조합되어 진다. 저수준 IR 코드의 베이직 블락은 하나의 엔트리(entry) 포인트와 하나의 엑시트(exit) 포인트만을 가지고 있으므로 이에 상응하는 HDL 코드의 하나의 스테이트 블락도 하나의 엔트리 포인트와 엑시트 포인트를 갖는다. 스테이트 블락의 엔트리 스테이트는 이에 상응하는 저수준 IR 코드의 베이직 블락의 넘버로 번호붙여진다. 이러한 번호부여 방식은 블락 스테이트들 사이의 제어 흐름의 변화를 매우 용이하게 가능하도록 한다. 예를 들어, 상기 컨트롤 유닛이 베이직 블락 3의 엔트리 코드로의 점프를 원하는 경우, 이는 스테이트 블락 3으로 점프하면 된다는 것을 즉각적으로 알 수 잇는 것이다. 딸서, 베이직 블락의 가장 큰 번호가 n인 경우, n까지의 수는 블락 스테이트들의 엔트리 스테이트의 번호를 부여하기 위해 사용되며, 숫자 n+1은 함수의 엑시트 스테이트 블락을 위해 사용된다. n+2 이후의 수는 베이직 블락 내부의 스테이트들을 위해 사용된다. 상기 데이타 패스는 일반적인 소프트웨어의 함수와 마찬가지로 링크와 피호출자 저장 레지스터(link and callee-saved registers)를 시작시의 스택에 푸시(push)하고, 완료(end)시에 파핑(popping)한다. 비록 각각의 하드웨어 함수가 별도의 레지스터 파일을 가지고 있는 경우라도 푸시와 파핑은 재귀적 함수(recursive function)를 지원하기 위해 여전히 필요하다.

[0110] 데이타 패스 구성(implementation)의 일례가 도 10(d)에 나타내어있다. 하드웨어 함수의 RTL 코드는 2부터 6까지의 5개의 베이직 블락을 포함하고 있다. 링크 레지스터 "reg0"은 스테이트 2에서 스택으로 푸쉬(push)되고 스테이트 블락 5의 스테이트 16 및 블락 6의 스테이트 19에서 재저장(restore)된다. C 함수는 레지스터 집약적(register intensive)이지 않기 때문에 컴파일러는 호출자 저장 레지스터(caller-saved registers)인 "reg7 <= reg14"만을 실행을 위해 사용하며, 따라서, 링크 레지스터를 제외한 기타 피호출자 저장 레지스터(callee-saved register)를 저장(save)할 필요가 없다. C의 일반 어레이(global array C)의 기본 어드레스(based address)는 스테이트 3에서 "reg9"에 할당되었으며, 루프의 바디 계산(loop body calculation)은 블락 4의 스테이트 4,

12, 13, 14, 15에서 실행되었다. 도 10 (a) 루프의 어레이 요소(array element) "c[i]"는 스테이트 도 10 (d)의 스테이트 4, 12 및 13부터 "reg7"에 쓰여진다. 상기 어레이 요소는 스테이트 14에서 "reg14"에 "reg14 <= reg7+reg14;"와 같이 축적된다. 도 10 (a) 루프의 쓰레숄드(threshold) "n"은 첫 번째 아규먼트 레지스터인 "reg12"에 저장되고, 따라서 상기 컨트롤 유닛의 스테이트 15에서 브랜치 디시젼(branch decision)을 정하기 위해 "reg8"에 저장된 루프의 도입(induction) 변수 "i"는 "successor[4] = (reg8 !=reg12);"와 같이 "reg12"와 비교된다.

[0111] 도 10 (c)를 참조하면, 생성된 컨트롤 유닛은 두 개의 "always" 블락을 포함함을 알 수 있다. 즉, 스테이트 변수 "pc"를 위한 것과 조합된 다음 스테이트 "nx_pc"의 두 블락을 포함함을 알 수 있다. 비교(comparison) 및 점프 명령을 제외한 대부분의 저수준 IR 인스트럭션은 데이타 패스에서 실행된다. 점프 명령은 베이직 블락들 사이의 제어 흐름을 변경시킨다. 즉, 스테이트 변수인 "pc" 값을 변경시키는 것이다. 따라서, 이는 데이타 패스가 아닌 컨트롤 유닛 내에 구현된다. 비교 명령은 점프 명령을 수행하기 위한 조건을 설정하기위해 사용되며, 따라서 점프명령과 마찬가지로 데이타 패스에서 제외된다. 비교 동작은 연속적인 할당(continuous assignment)를 통해 체크되며, 따라서, 수행에 의한 딜레이를 최소화할 수 있다. 예를 들어, 베이직 블락 2의 레지스터 12인 "reg12" 값이 0 보다 큰지에 대한 비교(comparison) 체크는 도 10 (c)의 스테이트먼트 "assign successor[2]=(reg12 <= 0)"으로 구현된다. 제어 흐름이 베이직 블락 2의 마지막에 도달한 경우, "successor[2]"의 값은 올바르게 설정되고, 이는 컨트롤 유닛에 의해 실행 스테이트를 변경하기 위한 점프 명령을 수행하기 위해 사용된다. 도 10의 베이직 블락 2는 두 개의 계승자(successor)를 가지며, 이는 베이직 블락 3과 6이다. 즉, "10: nx_pc <= successor[2] ? 6 : 3;" 스테이트먼트에 의해 브랜치의 조건을 만족하는 경우에는 베이직 블락 6으로 제어 흐름이 진행되며, 그렇지 않은 경우 베이직 블락 3이 다음 실행을 위해 선택되어 진다.

[0112] 함수의 마지막 부분에 제어 흐름은 엑시트 스테이트로 진행되며(도 10 (c)의 스테이트 7 참조), 실행 중인 함수의 엑시트 상태(status)를 나타내기 위해 도 10 (c)의 "assign finish = (pc==7);" 과 같은 스테이트먼트에 의해 연속적인 할당(continuous assignment)으로 종료 신호가 설정된다. 이러한 엑시트 스테이트는 RTL 베이직 블락의 엑시트와 동일하다. 이 후 스테이트 변수는 도 10 (c)의 "22: nx_pc<=22;"과 같은 스테이트먼트에 의해 스톨 상태(stall status)로 변경된다.

[0113] 다음으로, 임베디드 마이크로 프로세서인 ARM과 같은 메인코어와 FPGA와 같은 전용하드웨어 사이의 정보교환에 대해 설명하기로 한다.

[0114] 도 11은 본 발명의 일 실시예에 따른 메인 코어와 전용 하드웨어 사이의 커뮤니케이션을 나타내는 블락 다이어그램이다. 도 11을 참조하면, 메인 코어와 전용 하드웨어는 메모리와 활성화 신호(enable signal)와 같은 제어 신호(control signal)와 아규먼트(argument)를 통하여 정보 교환을 하는 것을 알 수 있다.

[0115] 다음으로, 메인 코어와 전용 하드웨어 사이의 콜링 컨벤션(calling convention)에 대해 설명하기로 한다.

[0116] 도 12는 본 발명의 일 실시예에 따른 메인 코어에서 실행되는 소프트웨어부분 코드와 전용 하드웨어에서 실행되는 하드웨어부분 코드 사이의 콜링 프로시져(calling procedure)를 나타내는 순서도이다. 도 12를 참조하면, 시작점에서 메인코어에서 컴파일된 소프트웨어 부분 코드가 실행되고(S1010), 상기 소프트웨어 부분 코드가 전용 하드웨어의 HWIP(Hardware Intellectual Property)의 실행이 필요한지 여부에 대해 판단하고(S1020), 전용하드웨어의 실행이 필요한 경우 작동 활성화 신호(enable signal) 및 파라미터를 전용하드웨어에 전송하여 전용 하드웨어를 호출하고(S1030), 활성화된 전용하드웨어는 호출된 HWIP를 실행하고(S1040), 실행 완료 후 실행 결과로 데이타를 업데이트하고 실행완료에 대해 상기 메인 코어에 통지하며(S1050), 전용하드웨어의 실행완료를 통지받은 메인 코어는 업데이트된 데이타로 소프트웨어 부분 코드의 실행을 재개(S1060)함을 알 수 있다. 이후 더 이상 전용하드웨어의 HWIP의 실행이 필요없는 경우 프로그램은 종료되는 것이다.

[0117] 도 13은 본 발명의 일 실시예에 따른 메인코어와 전용하드웨어가 소결합인 경우의 소프트웨어부분 코드와 하드웨어부분 코드 사이의 크로스 콜을 지원하기 위한 소프트웨어(SW)/하드웨어(HW) 인터페이스 및 하드웨어 컨트롤러를 나타내는 블락 다이어그램이다.

[0118] 도 13에 나타난 바와 같이, 메인 코어와 전용 하드웨어가 소결합인 경우의 크로스 콜을 지원하기 위해 두 개의 추가적인 하드웨어 구성요소(hardware components)인 SW/HW 인터페이스 및 하드웨어 컨트롤러가 추가된다. 전용 하드웨어의 상기 SW/HW 인터페이스 모듈은 크로스 콜 시에 일정한 인터페이스가 제공되도록 메인 코어에 해당하는 소프트웨어를 포장(wrap)하기 위해 구현된다. 상기 SW/HW 인터페이스 모듈은 전용 하드웨어로부터 메인 코어로의 커뮤니케이션의 세부사항을 숨기며, 하드웨어 컨트롤러로 하여금 기타 다른 하드웨어 컴포넌트와 동일한 방식으로 소프트웨어인 메인 코어와 통신할 수 있도록 한다. 상기 SW/HW의 HWID(hardware identification number)는 도 13에 나타난 바와 같이 디폴트(default) 값 0으로 설정되어 있으며, SW/HW 인터페이스는 HWID의 하나일 수 있는 것이다.

[0119] 상기 하드웨어 컨트롤러는 상기 SW/HW 인터페이스 및 하드웨어 구성요소(components)를 제어하고 스케줄링(scheduling)하기 위해 구현되었다. 이를 위해, 상기 하드웨어 컨트롤러는 하드웨어 모듈 사이에 공유되는 공유 레지스터 세트를 포함하며, 상기 공유 레지스터 세트는 4개의 아규먼트 리지스터, 하드웨어 스택 포인터 및 링크 레지스터를 포함한다. 상기 하드웨어 컨트롤러는 한 번에 하나의 HWIP를 활성화시키며, 활성화된 HWIP만이 모든 시스템 리소스에 액세스 가능하며, 나머지 모듈들은 액세스하지 못하도록 블락된다(도 13의 HW module 1 참조).

[0120] 도 14는 본 발명의 일 실시예에 따른 메인코어와 전용하드웨어가 소결합인 경우의 메인코어에서 실행되는 소프트웨어부분 코드와 전용하드웨어에서 실행되는 하드웨어부분 코드 사이의 크로스 콜을 수행하는 메카니즘을 나타내는 예시도이다.

[0121] 도 14를 참조하면, 변환 대상 C 코드에 프로그래밍 디렉티브가 표시되어 있음을 알 수 있다. 도 14에서는 "min 함수"와 "for 루프" 함수가 HDL 코드로의 변환대상으로 표시되어 있다. C-to-C 소스 변환기는 도 14의 Preprocessed code에 나타난 바와 같이, "for 루프"에 해당하는 "hw_func_1" 및 "min 함수"에 해당하는 "hw_func_2"의 두 개의 하드웨어 서브루틴을 생성한다. 변환대상 오리지널 코드의 for 루프는 전용 하드웨어와의 커뮤니케이션을 수행하기 위해 API 랩퍼(wrapper) 함수로의 호출(call)로 대체되었다. 이러한 랩퍼펑션의 첫 번째 아규먼트는 호출된 하드웨어 함수의 HWID를 나타내며(예로서, 1의 경우 하드웨어 함수 hw_func_1 을 나타냄), 나머지 아규먼트는 피호출자에게 전달될 변수 파라미터를 나타낸다. 랩퍼 함수의 세부 사항은 도 15 (d)에 나타내어져 있다.

[0122] 도 15는 본 발명의 일 실시예에 따른 메인코어와 전용하드웨어가 소결합인 경우의 도 14의 메인코어와 전용하드웨어 사이의 크로스 콜의 예시 코드에 대한 크로스 콜 임플리멘테이션을 나타내는 예시도이다.

[0123] 메인 코어와 전용 하드웨어가 소결합인 경우, 전용 하드웨어는 메인 코어와 별도의 시스템 레지스터를 사용하며, 메인 코어와 인터럽트(interrupting)를 통해 커뮤니케이션한다. 또한, 상기 메인 코어는 커뮤니케이션을 위해 랩퍼라는 API를 사용한다.

[0124] 도 15를 참조하면, 도 15 (a)는 소프트웨어의 API 랩퍼 함수가 HWIP1(hardware intellectual property 1), 즉, "hw_func_1"을 호출하는 경우의 실행 흐름(execution flow)을 나타내며, 실행 단계는 (1) 내지 (5)로 번호가 부여되었다.

[0125] 단계 (1)에서, 상기 랩퍼 함수는 HWIP1의 첫 4개 아규먼트(a, b, c 및 d)를 아규먼트 레지스터에 저장하고, 마지막 아규먼트 e를 자신의 스택에 저장하며, 상기 SW/HW 인터페이스를 통해 전용 하드웨어의 하드웨어 컨트롤러의 링크 레지스터에 리턴 어드레스를 저장한다. 이후, 상기 랩퍼 함수는 HWIP1의 실행을 위해 활성화 신호(enable signal)인 "HWID=1"을 컨트롤러에 전송한다.

[0126] 단계 (2)에서, 상기 하드웨어 컨트롤러는 HWIP1 모듈을 활성화시킨다. HWIP1의 실행 시작점에서, HWIP1은 상기 하드웨어 컨트롤러의 스택 포인터 값을 읽어들이고 자신의 스택 스페이스를 할당한다. HWIP1은 상기 하드웨어 컨트롤러의 링크 레지스터로부터 자신의 리턴 어드레스를 자신의 스택에 저장하며, 자신의 바디코드의 실행을 시작한다.

[0127] 단계 (3)에서, HWIP1의 실행이 완료된 경우, HWIP1은 상기 스택 스페이스를 재저장하고 리턴 어드레스와 함께 종료 신호(finish signal)를 상기 하드웨어 컨트롤러에 전송한다.

[0128] 단계 (4)에서, 상기 하드웨어 컨트롤러는 상기 랩퍼 함수를 활성화시키기 위해 상기 SW/HW 인터페이스를 통해

메인 코어에게 인터럽트 시그널을 전송한다.

[0129] 단계 (5)에서, HWIP1의 실행 완료를 랩퍼 함수에 알리기 위해 인터럽트 핸들러는 일반 종료 변수를 설정하고, 상기 랩퍼 함수는 종료된다.

[0130] 도 15 (d)는 상기 랩퍼 함수의 의사(擬似) 코드(pseudo code)를 나타낸다. 하드웨어 함수를 호출(call)한 후, 상기 랩퍼 함수는 호출된 하드웨어 함수의 실행 완료(소프트웨어가 하드웨어를 호출한 경우) 또는 호출된 하드웨어가 다른 소프트웨어 함수를 호출(소프트웨어가 하드웨어를 호출하고, 다시 하드웨어가 소프트웨어를 호출하는 경우)하기를 기다린다. 호출된 하드웨어 함수가 소프트웨어로 리턴하는 경우 상기 랩퍼는 종료하며, 하드웨어 함수가 다른 소프트웨어 함수를 호출하는 경우 상기 랩퍼는 호출을 돕게된다.

[0131] 하드웨어 함수로부터의 소프트웨어 함수로의 호출도 반대의 경우와 유사하다. 도 14의 "hw_func_1"이 "max" 함수를 호출하는 경우를 살펴보면, 실행 시퀀스는 도 15 (b)에 나타나 있으며, 실행 순서는 다음과 같다.

[0132] (1) HWIP1이 "max" 함수의 첫 4개 아규먼트인 "a, b, c 및 d"를 하드웨어 컨트롤러의 아규먼트 레지스터에 저장하고, 마지막 아규먼트 "e"를 자신의 스택의 바닥(bottom)에 저장한다. 또한, HWIP1은 상기 하드웨어 컨트롤러의 링크 레지스터에 자신의 리턴 스테이트를 저장한다. 이후, HWIP1은 hwid=0과 함께 "max" 함수의 실제 어드레스를 sw_addr 신호로 상기 하드웨어 컨트롤러에 전송하며 이때 호출되는 함수를 위해 hwid=0도 함께 전송한다.

[0133] (2) 상기 하드웨어 컨트롤러는 HWIP1 모듈을 비활성화(disable) 시키며, "max" 함수 호출을 위해 SW/HW 인터페이스를 통해 메인 코어를 인터럽트한다.

[0134] (3) 인터럽트 핸들러는 글로벌 변수를 통해 랩퍼 함수에게 상기 호출(call)에 대해 통지한다(도 15 (d)의 hw_called 참조). 상기 랩퍼 함수는 상기 메인 코어의 스택 포인터를 업데이트하고, 아규먼트 레지스터를 상기 하드웨어 컨트롤러의 값으로 업데이트하고, 자신의 로컬 스택에 상기 HWIP1의 리턴 스테이트를 저장하고, 상기 "max" 함수를 호출한다.

[0135] (4) 상기 "max" 함수의 실행이 완료되면, 상기 랩퍼 함수는 상기 "max" 함수의 리턴 값과 종료 신호를 상기 하드웨어 컨트롤러에 전송하고 상기 SW/HW 인터페이스를 통해 리턴 스테이트를 재저장한다.

[0136] (5) 상기 하드웨어 컨트롤러는 링크 레지스터로부터 리턴 스테이트를 제공함으로써 HWIP1을 활성화시킨다.

[0137] 도 15 (c)는 도 14의 HWIP1인 "hw_func_1"이 HWIP2인 "hw_func_2"를 호출하는 경우를 나타내며, 호출 프로세스는 다음과 같다.

[0138] (1) HWIP1은 상기 하드웨어 컨트롤러의 아규먼트 레지스터와 스택에 아규먼트를 저장하고, 상기 하드웨어 컨트롤러의 링크 레지스터에 리턴 스테이트를 저장한다. 이후, HWIP1은 호출 대상 함수를 나타내는 "hwid=2"와 함께 호출 신호(call signal)를 상기 하드웨어 컨트롤러에 전송한다.

[0139] (2) 상기 하드웨어 컨트롤러는 HWIP2를 활성화시키고, HWIP2는 HWIP1의 리턴 스테이트를 자신의 스택에 저장한다.

[0140] (3) HWIP2의 실행완료시, HWIP2는 호출한 함수인 HWIP1의 리턴 스테이트와 함께 종료 신호를 상기 하드웨어 컨트롤러에 전송한다.

[0141] (4) 상기 하드웨어 컨트롤러는 HWIP1을 다시 활성화 시키고, HWIP1은 자신의 원래의 스테이트에서 실행을 재개한다.

[0142] 다음으로, 메인 코어와 전용 하드웨어가 밀결합(tight coupling)인 경우의 메인 코어와 전용하드웨어 사이의 콜링 컨벤션에 대해 설명하기로 한다.

[0143] 도 16은 본 발명의 일 실시예에 따른 메인코어와 전용하드웨어가 밀결합인 경우의 메인코어와 전용하드웨어 사이의 커뮤니케이션 인터페이스를 나타내는 블락 다이어그램이다.

[0144] 도 16을 참조하면, 상기 메인 코어와 상기 전용 하드웨어의 HWIP들 사이의 커뮤니케이션은 인터커넥션 네트워크(interconnection network)를 통해 이루어짐을 알 수 있다. 상기 인터커넥션 네트워크는 버스 시스템 및 상기 버스 시스템을 제어하기 위한 하드웨어 컨트롤러를 포함한다. 모든 HWIP 및 메인 코어는 상기 버스 시스템을 통해 다른 컴포넌트가 자신들의 레지스터 파일을 읽을 수 있도록 구성된다. 퍼블릭 레지스터(public register)는 스택 포인터, 링크 레지스터, 아규먼트 레지스터 및 함수(function) 리턴 레지스터를 포함한다. 따라서, 상기

하드웨어 컨트롤러 내에 별도의 시스템 레지스터 세트를 필요로 하지 않는다.

[0145] 상기 메인 코어의 인터페이스는 상기 하드웨어 컨트롤러가 인터럽트 신호를 사용하는 대신 직접 제어 신호를 전송하거나 수신할 수 있도록 하고 있으며, 상기 메인 코어와 HWIP들은 all-to-all 네트워크 토폴로지(topology)에 의해 연결되어 있지 않으므로, 하나의 컴포넌트가 다른 컴포넌트와 통신하기를 원하는 경우 상기 하드웨어 컨트롤러를 통해서 통신(communicate)하여야 한다. 상기 하드웨어 컨트롤러는 상기 메인 코어와 HWIP의 호출제어신호(call control signal)를 검사(examine)하고, 결정에 따라, 호출 시퀀스(calling sequence)에 기초하여 대상이되는 컴포넌트를 활성화 시키고, 컴포넌트가 버스로 데이타를 전송할 수 있도록 한다. 메인 코어 내지 HWIP가 호출을 원하는 경우, 호출제어신호 및 피호출자의 어드레스를 상기 하드웨어 컨트롤러에 전송하고, 상기 하드웨어 컨트롤러는 호출 대상을 확인하고 호출자를 비활성화시키고, 피호출자를 활성화시키며, 호출자의 레지스터를 버스가 읽을 수 있도록 한다. 피호출자가 호출자에게 리턴하는 경우, 피호출자도 마찬가지로 종료 신호(finish signal)와 리턴 어드레스를 상기 하드웨어 컨트롤러에 전송한다. 상기 하드웨어 컨트롤러는 리턴 어드레스를 체크하여 호출자를 확인하고, 피호출자의 리턴 데이타를 버스가 읽을 수 있도록 하며, 피호출자를 비활성화시키고 호출자를 다시 활성화시킨다.

[0146] 도 17 및 18은 본 발명의 일 실시예에 따른 메인코어와 전용하드웨어가 밀결합인 경우의 도 16의 크로스 콜 임플리멘테이션을 나타내는 블락 다이어그램이다.

[0147] 커뮤니케이션 오버헤드를 감소시키기 위해 상기 메인코어의 인터페이스는 상기 전용 하드웨어와 보다 나은 상호협조가 가능하도록 모디파이(modify)된다. 즉, software-to-hardcore 호출을 수행하기 위한 중간(intermediate) 단계를 감소시키기 위해 "hcall"이라고 불리는 특수 명령(special instruction)이 상기 메인 코어의 ISA에 더해진다. 컴파일링하는 경우, 메인 컴파일러는 피호출자가 하드웨어부분 코드롤 구현되는지 아니면 소프트웨어부분 코드로 구현되는지 체크하고, 하드웨어 함수인 경우 메인 컴파일러는 일반 펑션 콜 인스트럭션 대신 "hcall" 인스트럭션을 자동적으로 생성한다. "hcall" 인스트럭션의 사용은 상기 메인 코어로 하여금 software-to-hardware 크로스 콜을 하나의 기계 명령(machine instruction)만으로 직접 수행할 수 있도록 한다. 인스트럭션 세트를 수정(modify)하지 않는다면, 상기 메인 코어는 전용 하드웨어와의 통신(communicate)을 위해 도 14의 랩퍼 함수와 같은 중간 API 함수를 사용하여야 하며, 이 경우 높은 오버헤드를 면치 못한다.

[0148] 도 17을 참조하면, 본 발명에 따른 software-to-hardware cross call 수행은 ① 내지 ⑥의 6 단계로 이루어짐을 알 수 있다.

[0149] ① "hcall" 인스트럭션이 실행 단계인 경우, "hcall" 인스트럭션은 메인 코어 파이프라인을 스톨(stall)하기 위해 "nx_pc_unready" 신호를 개시되도록(trigger) 한다.

[0150] ② 상기 신호는 페치(Fetch) 컨트롤러 컴포넌트로 하여금 인스트럭션 페칭(fetching) 프로세스를 스톨(stall)하도록 하고, 상기 메인 코어가 "hcall" 인스트럭션 이후의 모든 인스트럭션을 상기 파이프라인과 인스트럭션 큐(instruction queue)에서 플러쉬(flush)하도록 한다.

[0151] ③ 상기 "hcall" 이 write back 단계에서 실행되는 경우, 호출에 대해 통지하기 위해 "hw_enable" 신호가 피호출자(callee)의 HWID와 함께 하드웨어 컨트롤러에게 전송되며, 또한, 일반적인 호출과 같이 리턴 어드레스를 위한 다음 실행 어드레스가 링크 레지스터에 저장된다.

[0152] ④ 하드웨어측에서, 상기 하드웨어 컨트롤러는 전송받은 HWID 값에 기초하여 "hw_enable" 신호 및 컨트롤 시그널 "called"를 사용하여 호출된 HWIP1을 활성화시킨다.

[0153] ⑤ 활성화된 HWIP1은 메인 코어의 스택 포인터, 아규먼트 레지스터, 링크 레지스터 등의 출력 값(output value)을 이들에 해당하는 자신의 레지스터에 복사하고 실행을 시작한다. HWIP1의 실행완료시, HWIP1은 종료 신호를 상기 하드웨어 컨트롤러에 전송하고, 상기 하드웨어 컨트롤러는 링크 레지스터 내의 HWIP1의 리턴 어드레스를 체크하여 호출자(caller)가 상기 메인 코어임을 확인한다.

[0154] ⑥ 확인 후, 상기 메인 컨트롤러는 피호출자(callee)인 HWIP1의 종료 스테이터스(status)를 알리기 위해 "hw_fin" 신호를 전송한다.

[0155] 상기 메인 코어는 페칭 프로세스를 활성화시키고, 스택 포인터, 링크 및 리턴 레지스터 등을 피호출자의 값으로 조정(set)한다. 이후, 상기 메이 코어는 링크 레지스터의 하드웨어 함수의 리턴 어드레스로부터 실행을 재개한

다.

[0156] 도 18을 참조하면, 본 발명에 따른 hardware-to-software cross call 수행은 ① 내지 ⑤의 5 단계로 이루어짐을 알 수 있다.

[0157] ① HWIP1은 호출 신호(call signal) 및 호출대상 함수의 어드레스를 설명하는 "target_addr" 신호를 포함한 제어 신호를 하드웨어 컨트롤러에게 전송한다. 또한, HWIP1은 자신의 링크 레지스터에 자신의 다음 스테이트의 어드레스를 저장하고, 아규먼트를 자신의 아규먼트 레지스터와 스택 스페이스에 저장한다.

[0158] ② 상기 하드웨어 컨트롤러는 "target_addr" 신호를 검사하여 호출되는 함수가 메인 코어의 소프트웨어 함수임을 확인하고, 하드웨어 함수로부터의 호출에 대해 상기 메인 코어에게 통지하기 위해 "sw_enable" 신호를 전송한다. 호출된 경우, 호출된 소프트웨어 피호출자(callee)를 실행하기 위해, 상기 메인 코어는 호출자(caller)의 데이타를 버스를 통해 자신의 레지스터에 복사하고, 자신의 프로그램 카운터 레지스터를 "target_addr" 신호 값으로 설정하며, 페칭 인스트럭션을 활성화 시킨다.

[0159] ③ 피호출자의 실행 완료시, 상기 메인 코어는 호출자인 HWIP1에 리턴한다. 호출한 하드웨어 함수(function)로 리턴하는 경우, 상기 메인 코어는 일반적인 소프트웨어 리턴과 다르게 대상 어드레스로 점프하지 않고, 페칭 인스트럭션을 스톨(stall)하고 파이프라인을 플러쉬하기 위해 "nx_pc_unready" 신호를 설정한다.

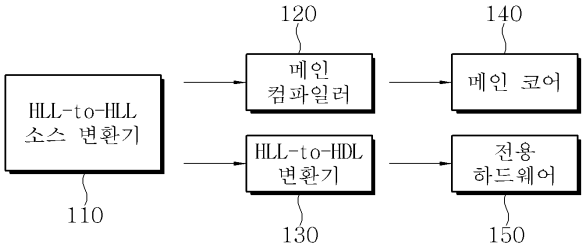
[0160] ④ Write back 단계에 하드웨어 리턴 인스트럭션은 "sw_fin" 신호를 상기 하드웨어 컨트롤러에게 전송한다.

[0161] ⑤ 상기 하드웨어 컨트롤러는 링크 레지스터 값을 검사하여 호출자(caller)가 HWIP1임을 확인하고, "hw_enable" 신호로 HWIP1을 활성화 시킨다. 재활성화된 경우, HWIP1은 피호출자(callee)의 리턴 값을 읽은 후 링크 레지스터 내의 자신의 스테이트 변수를 업데이트하고 해당 스테이트로부터 실행을 재개한다.

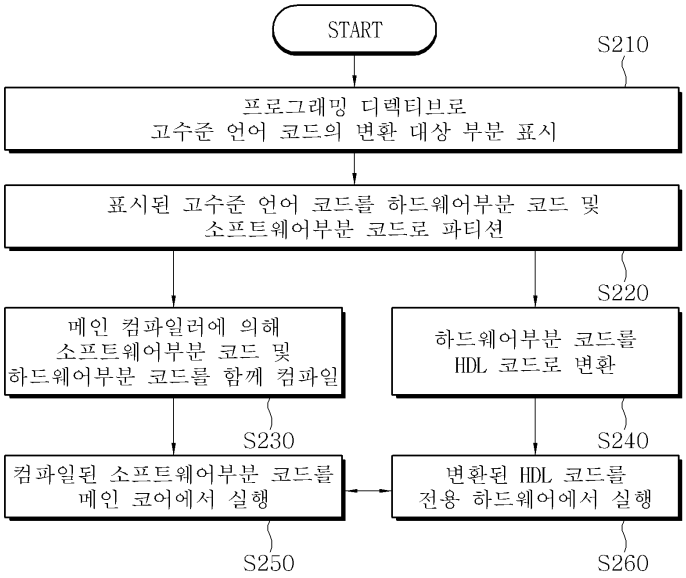
[0162] 이상 본 발명의 실시예에 따른 고수준 언어 코드의 HDL 코드로의 변환 방법 및 시스템에 대하여 설명하였으며, 본 발명은 상기 실시예에 한정되지 않으며, 많은 변형이 본 발명의 사상 내에서 본 발명이 속하는 기술분야에서 통상의 지식을 가진 자에 의하여 가능함은 물론이다.

도면

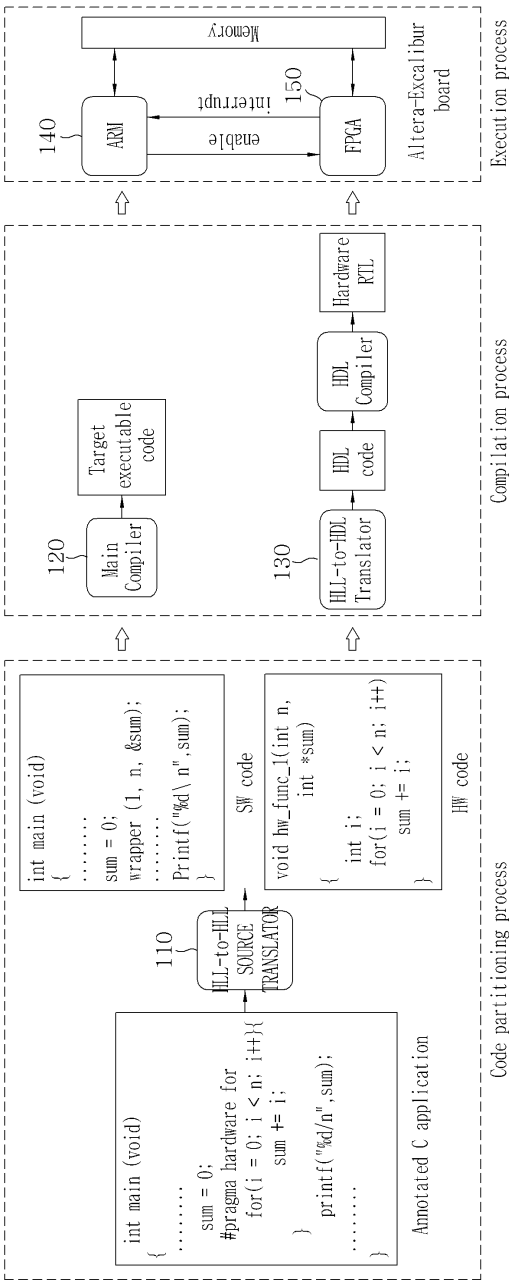
도면1



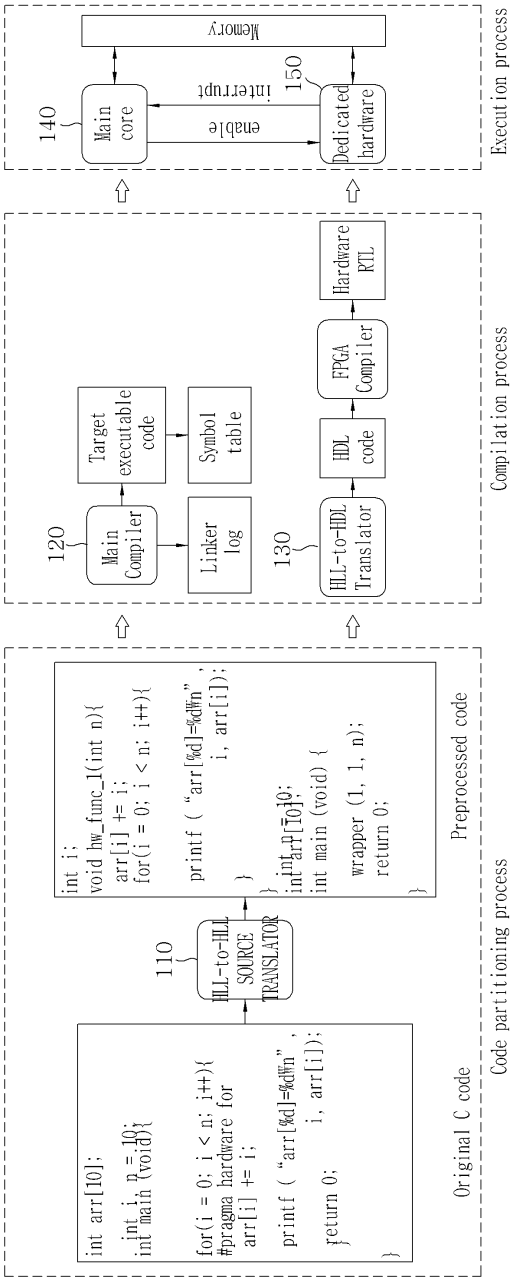
도면2


START
S210
프로그래밍 디렉티브로
고수준 언어 코드의 변환 대상 부분 표시
표시된 고수준 언어 코드를 하드웨어부분 코드 및
소프트웨어부분 코드로 파티션
S220
메인 컴파일러에 의해
소프트웨어부분 코드 및
하드웨어부분 코드를 함께 컴파일
하드웨어부분 코드를
HDL 코드로 변환
S230
S240
컴파일된 소프트웨어부분 코드를
메인 코어에서 실행
변환된 HDL 코드를
전용 하드웨어에서 실행
S250
S260

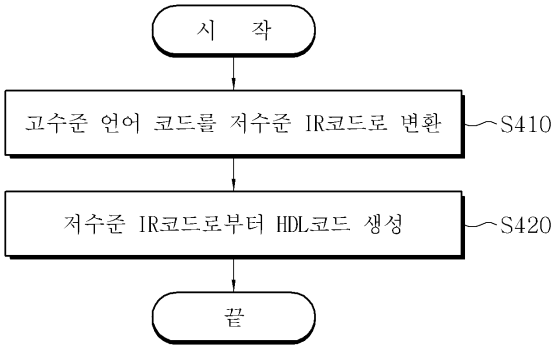
도면3


Code partitioning process
Annotated C application
int main (void)
{
.....
sum = 0;
#pragma hardware for
for(i = 0; i < n; i++){
sum += i;
}
printf("%d/n",sum);
.....
}
110
HLL-to-HLL SOURCE TRANSLATOR
SW code
int main (void)
{
.....
sum = 0;
wrapper (1, n, &sum);
.....
Printf("%d\ n",sum);
}
HW code
void hw_func_1(int n,
int *sum)
{
int i;
for(i = 0; i < n; i++)
sum += i;
}
Compilation process
120
Main Compiler
Target executable code
130
HLL-to-HDL Translator
HDL code
HDL Compiler
Hardware RTL
Execution process
Altera-Excalibur board
140
ARM
enable
interrupt
150
FPGA
Memory

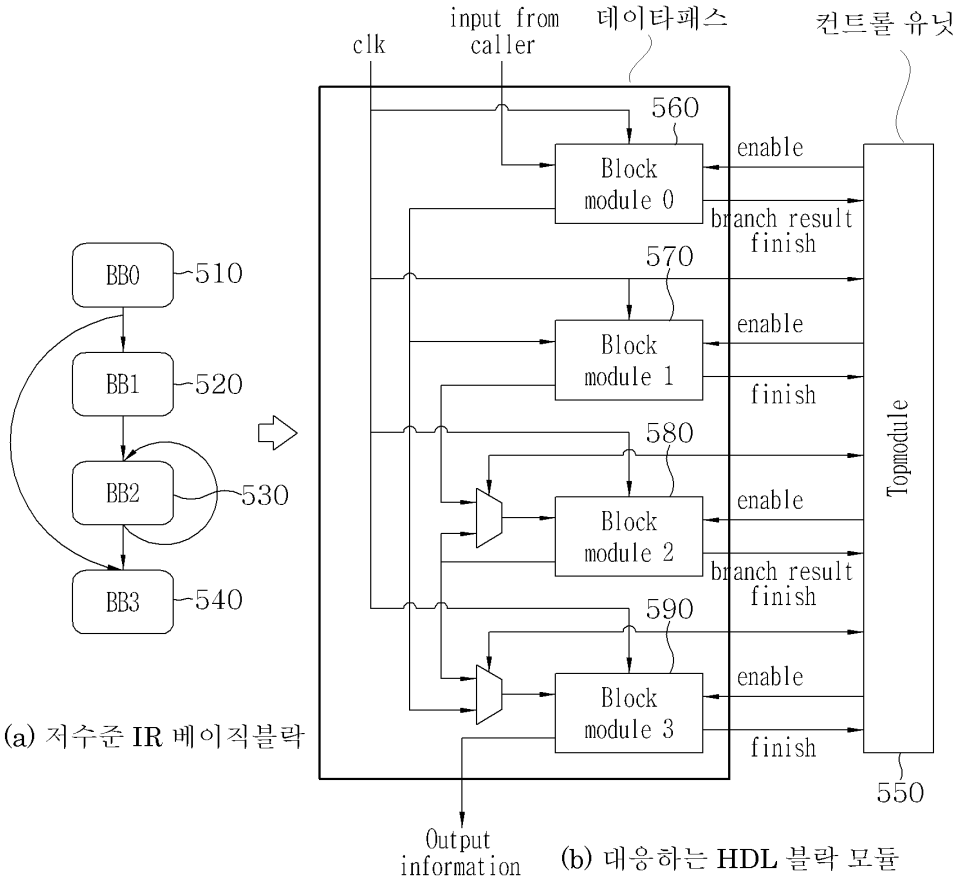
도면4


Execution process
Memory
140
Main core
interrupt
enable
150
Dedicated hardware
Compilation process
120
Main Compiler
Target executable code
Symbol table
Linker log
130
HLL-to-HDL Translator
HDL code
FPGA Compiler
Hardware RTL
Code partitioning process
Original C code
int arr[10];
int i, n=10;
int main (void){
for(i = 0; i < n; i++){
#pragma hardware for
arr[i] += i;
printf ("arr[%d]=%d\n" ,
i, arr[i]);
return 0;
}
}
110
HLL-to-HLL SOURCE TRANSLATOR
Preprocessed code
int i;
void hw_func_1(int n){
arr[i] += i;
for(i = 0; i < n; i++){
printf ("arr[%d]=%d\n" ,
i, arr[i]);
}
}
int arr[10];
int main (void) {
wrapper (1, 1, n);
return 0;
}

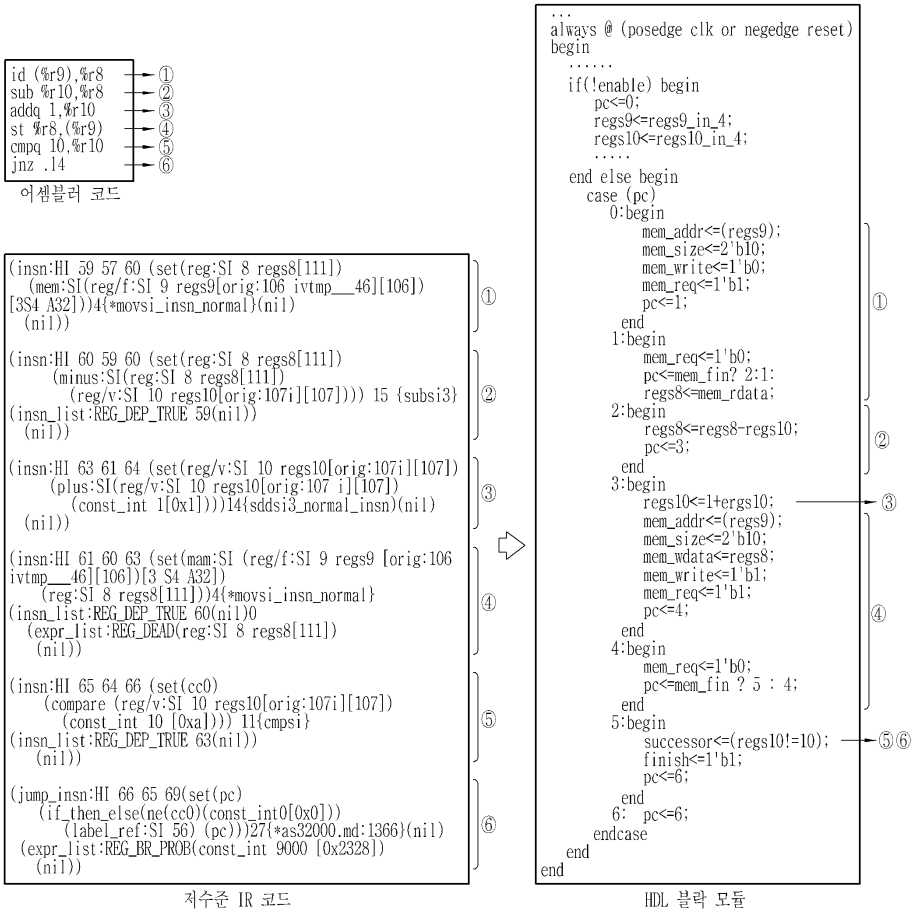
도면5


시 작
고수준 언어 코드를 저수준 IR코드로 변환
S410
저수준 IR코드로부터 HDL코드 생성
S420
끝

도면6


BB0
510
BB1
520
BB2
530
BB3
540
(a) 저수준 IR 베이직블락
clk
input from caller
데이타패스
컨트롤 유닛
560
Block module 0
enable
branch result
finish
570
Block module 1
enable
finish
580
Block module 2
enable
branch result
finish
590
Block module 3
enable
finish
Topmodule
550
Output information
(b) 대응하는 HDL 블락 모듈

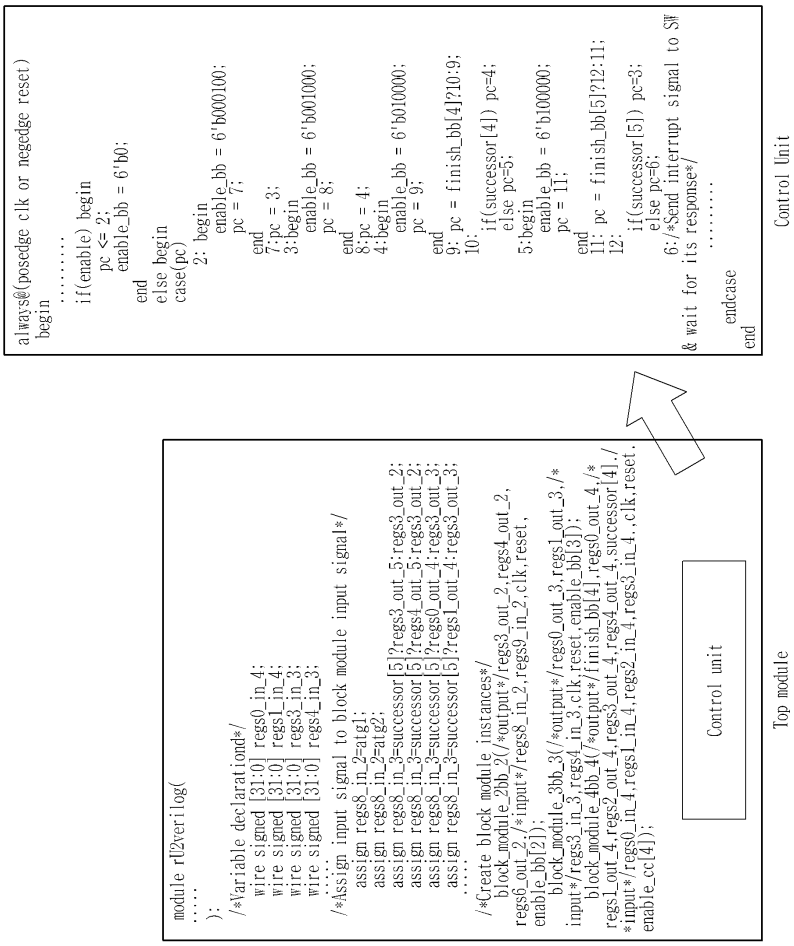
도면7


id (%r9),%r8 → ①
sub %r10,%r8 → ②
addq 1,%r10 → ③
st %r8,(%r9) → ④
cmpq 10,%r10 → ⑤
jnz .14 → ⑥
어셈블러 코드
(insn:HI 59 57 60 (set(reg:SI 8 regs8[111])
(mem:SI(reg/f:SI 9 regs9[orig:106 ivtmp___46][106])
[3S4 A32]))4{*movsi_insn_normal}(nil)
(nil))
(insn:HI 60 59 60 (set(reg:SI 8 regs8[111])
(minus:SI(reg:SI 8 regs8[111])
(reg/v:SI 10 regs10[orig:107i][107])) 15 {subsi3}
(insn_list:REG_DEP_TRUE 59(nil))
(nil))
(insn:HI 63 61 64 (set(reg/v:SI 10 regs10[orig:107i][107])
(plus:SI(reg/v:SI 10 regs10[orig:107 i][107])
(const_int 1[0x1])))14{sddsi3_normal_insn)(nil)
(nil))
(insn:HI 61 60 63 (set(mam:SI (reg/f:SI 9 regs9 [orig:106
ivtmp___46][106])[3 S4 A32])
(reg:SI 8 regs8[111]))4{*movsi_insn_normal}
(insn_list:REG_DEP_TRUE 60(nil)0
(expr_list:REG_DEAD(reg:SI 8 regs8[111])
(nil))
(insn:HI 65 64 66 (set(cc0)
(compare (reg/v:SI 10 regs10[orig:107i][107])
(const_int 10 [0xa])))) 11{cmpsi}
(insn_list:REG_DEP_TRUE 63(nil))
(nil))
(jump_insn:HI 66 65 69(set(pc)
(if_then_else(ne(cc0)(const_int0[0x0]))
(label_ref:SI 56) (pc)))27{*as32000.md:1366}(nil)
(expr_list:REG_BR_PROB(const_int 9000 [0x2328])
(nil))
①
②
③
④
⑤
⑥
지수준 IR 코드
...
always @ (posedge clk or negedge reset)
begin
.....
if(!enable) begin
pc<=0;
regs9<=regs9_in_4;
regs10<=regs10_in_4;
.....
end else begin
case (pc)
0:begin
mem_addr<=(regs9);
mem_size<=2'b10;
mem_write<=1'b0;
mem_req<=1'b1;
pc<=1;
end
1:begin
mem_req<=1'b0;
pc<=mem_fin? 2:1;
regs8<=mem_rdata;
2:begin
regs8<=regs8-regs10;
pc<=3;
end
3:begin
regs10<=1+ergs10;
mem_addr<=(regs9);
mem_size<=2'b10;
mem_wdata<=regs8;
mem_write<=1'b1;
mem_req<=1'b1;
pc<=4;
end
4:begin
mem_req<=1'b0;
pc<=mem_fin ? 5 : 4;
end
5:begin
successor<=(regs10!=10);
finish<=1'b1;
pc<=6;
end
6: pc<=6;
endcase
end
end
①
②
③
④
⑤⑥
HDL 블락 모듈

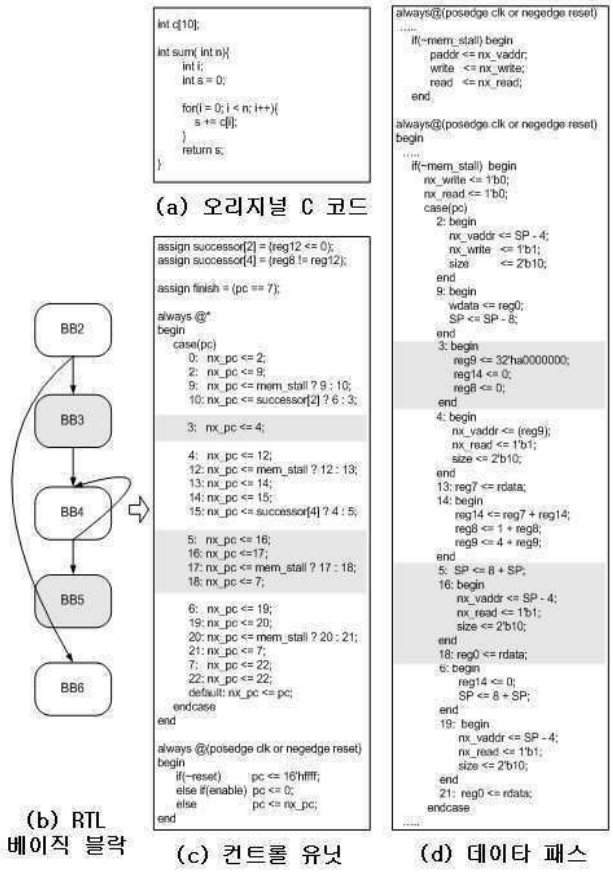
도면8

```
//enable_bb[i]=1:module i is enabled
//finish_bb[i]=1:module i finished its execution
//successor[i]:the taken or not taken value of module i branch
always@(posedge clk or negedge reset)
begin
    ...
  case(pc)
      0 begin
                enable_bb<=4'b0001; // enable module 0
                pc <= 1;
      end
      1 pc <= 2;
      2   //decide the next module after module 0 branch
        if(successor[0])
        begin
                enable_bb <= 4'b1000;
                pc <= 6;
        end
        else begin
                enable_bb <= 4'b0001;
                pc <= 3;
        end
      3 pc <= 4; // wait for block module 1
      4 begin
                enable_bb <= 4'b0100;
                pc <= 5;
      end
      5 if(finish_bb[2])
        begin
            if(successor[2])
            begin
                enable_bb<=4'b0100; // backward jump
                pc <= 5;
            end
            else begin
                enable_bb <= 4'b1000;
                pc <= 6;
            end
        end
        else pc <=5;
       6 if(finish_bb[3])
        begin
                pc <= 7;
                finish <=1;             //the function finished
                enable_bb <= 4'b0000;  //stop all block modules
        end
        else begin
                pc <= 6;
                enable_bb <= 4'b1000;
        end
       7 begin
                pc <= 7;
                finish <=0;
      end
  endcase
end
```

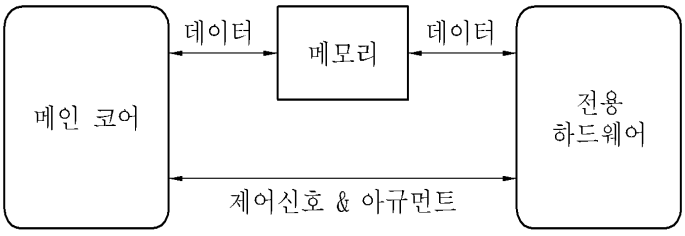
도면9


always@(posedge clk or negedge reset)
begin
.....
if(enable) begin
pc <= 2;
enable_bb = 6'b0;
end
else begin
case(pc)
2: begin
enable_bb = 6'b000100;
pc = 7;
end
7:pc = 3;
3:begin
enable_bb = 6'b001000;
pc = 8;
end
8:pc = 4;
4:begin
enable_bb = 6'b010000;
pc = 9;
end
9: pc = finish_bb[4]?10:9;
10:
if(successor[4]) pc=4;
else pc=5;
5:begin
enable_bb = 6'b100000;
pc = 11;
end
11: pc = finish_bb[5]?12:11;
12:
if(successor[5]) pc=3;
else pc=6;
6://*Send interrupt signal to SW
& wait for its response*/
.....
endcase
end
Control Unit
module rl2verilog(
.....
);
/*\variable declaration@*/
wire signed [31:0] regs0_in_4;
wire signed [31:0] regs1_in_4;
wire signed [31:0] regs3_in_3;
wire signed [31:0] regs4_in_3;
/*\assign input signal to block module input signal*/
assign regs8_in_2=arg1;
assign regs8_in_2=arg2;
assign regs8_in_3=successor[5]?regs3_out_5:regs3_out_2;
assign regs8_in_3=successor[5]?regs4_out_5:regs3_out_2;
assign regs8_in_3=successor[5]?regs0_out_4:regs3_out_3;
assign regs8_in_3=successor[5]?regs1_out_4:regs3_out_3;
.....
/*Create block module instances*/
block_module_2bb_2(/*output*/regs3_out_2,regs4_out_2,
regs6_out_2,/*input*/regs8_in_2,regs9_in_2,clk,reset,
enable_bb[2]);
block_module_3bb_3(/*output*/regs0_out_3,regs1_out_3,/*
input*/regs3_in_3,regs4_in_3,clk,reset,enable_bb[3]);
block_module_4bb_4(/*output*/finish_bb[4],regs0_out_4,/*
regs1_out_4,regs2_out_4,regs3_out_4,regs4_out_4,successor[4]../
input/regs0_in_4,regs1_in_4,regs2_in_4,regs3_in_4,.clk,reset,
enable_cc[4]);
Control unit
Top module

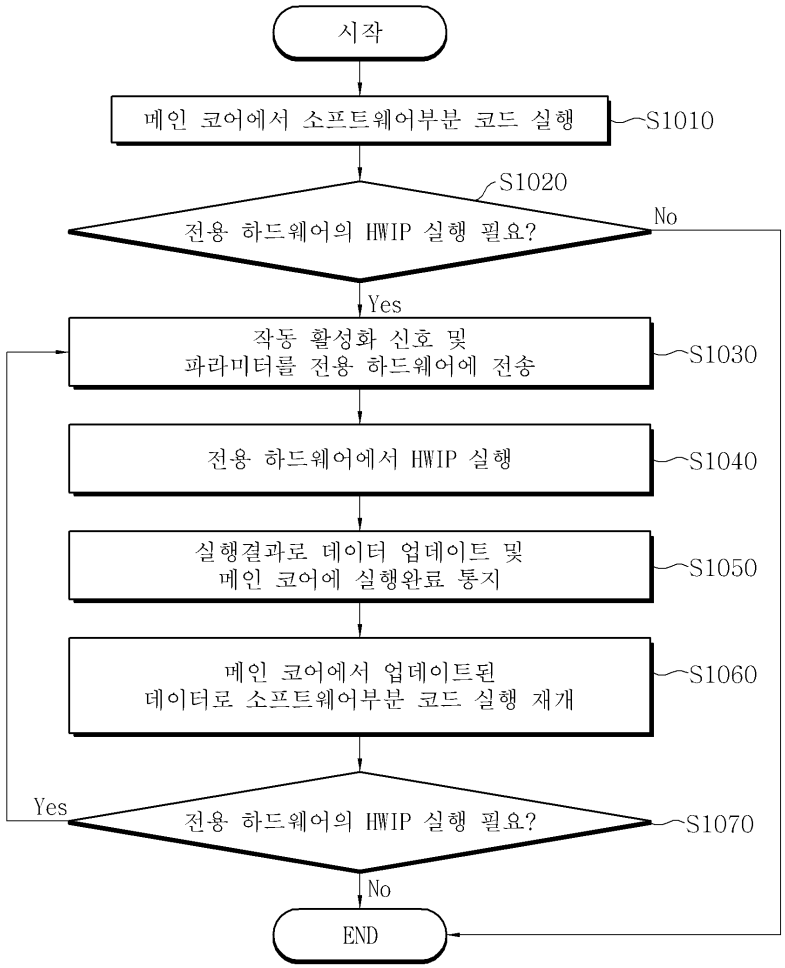
도면10


int c[10];
int sum(int n){
int i;
int s = 0;
for(i = 0; i < n; i++){
s += c[i];
}
return s;
}
(a) 오리지널 C 코드
BB2
BB3
BB4
BB5
BB6
(b) RTL 베이직 블락
assign successor[2] = (reg12 <= 0);
assign successor[4] = (reg8 != reg12);
assign finish = (pc == 7);
always @*
begin
case(pc)
0: nx_pc <= 2;
2: nx_pc <= 9;
9: nx_pc <= mem_stall ? 9 : 10;
10: nx_pc <= successor[2] ? 6 : 3;
3: nx_pc <= 4;
4: nx_pc <= 12;
12: nx_pc <= mem_stall ? 12 : 13;
13: nx_pc <= 14;
14: nx_pc <= 15;
15: nx_pc <= successor[4] ? 4 : 5;
5: nx_pc <= 16;
16: nx_pc <=17;
17: nx_pc <= mem_stall ? 17 : 18;
18: nx_pc <= 7;
6: nx_pc <= 19;
19: nx_pc <= 20;
20: nx_pc <= mem_stall ? 20 : 21;
21: nx_pc <= 7;
7: nx_pc <= 22;
22: nx_pc <= 22;
default: nx_pc <= pc;
endcase
end
always @(posedge clk or negedge reset)
begin
if(~reset) pc <= 16'hffff;
else if(enable) pc <= 0;
else pc <= nx_pc;
end
(c) 컨트롤 유닛
always@(posedge clk or negedge reset)
.....
if(~mem_stall) begin
paddr <= nx_vaddr;
write <= nx_write;
read <= nx_read;
end
always@(posedge clk or negedge reset)
begin
.....
if(~mem_stall) begin
nx_write <= 1'b0;
nx_read <= 1'b0;
case(pc)
2: begin
nx_vaddr <= SP - 4;
nx_write <= 1'b1;
size <= 2'b10;
end
9: begin
wdata <= reg0;
SP <= SP - 8;
end
3: begin
reg9 <= 32'ha0000000;
reg14 <= 0;
reg8 <= 0;
end
4: begin
nx_vaddr <= (reg9);
nx_read <= 1'b1;
size <= 2'b10;
end
13: reg7 <= rdata;
14: begin
reg14 <= reg7 + reg14;
reg8 <= 1 + reg8;
reg9 <= 4 + reg9;
end
5: SP <= 8 + SP;
16: begin
nx_vaddr <= SP - 4;
nx_read <= 1'b1;
size <= 2'b10;
end
18: reg0 <= rdata;
6: begin
reg14 <= 0;
SP <= 8 + SP;
end
19: begin
nx_vaddr <= SP - 4;
nx_read <= 1'b1;
size <= 2'b10;
end
21: reg0 <= rdata;
endcase
.....
(d) 데이터 패스

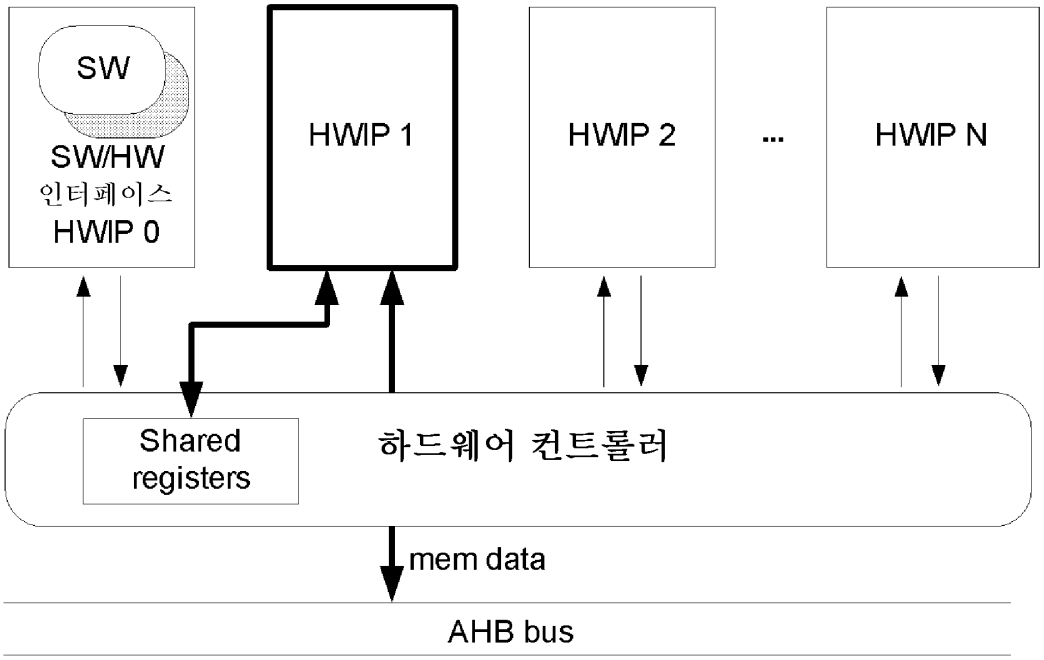
도면11


메인 코어
데이터
메모리
데이터
전용
하드웨어
제어신호 & 아규먼트

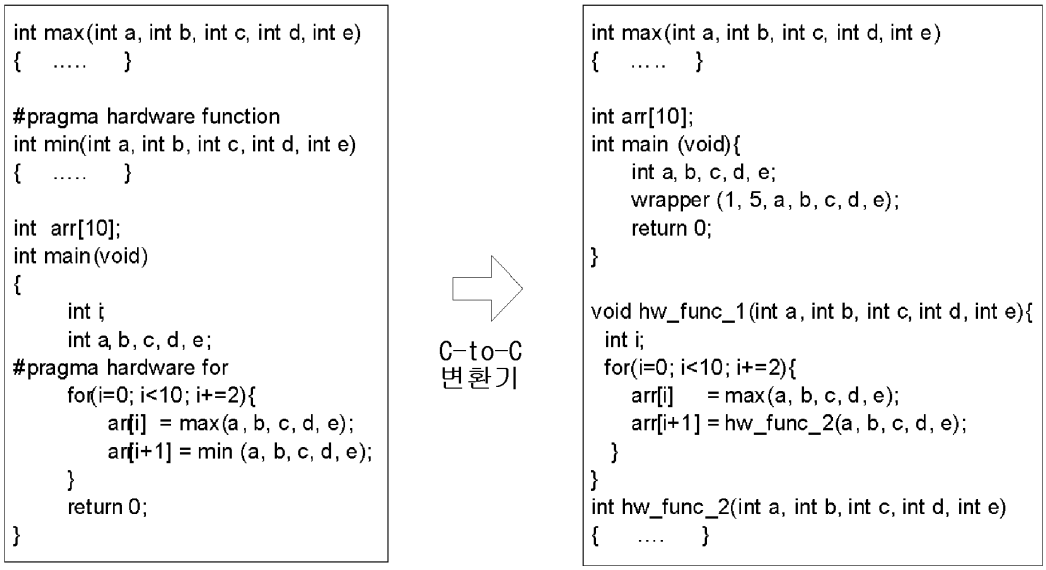
도면12


시작
메인 코어에서 소프트웨어부분 코드 실행
S1010
S1020
전용 하드웨어의 HWIP 실행 필요?
No
Yes
작동 활성화 신호 및
파라미터를 전용 하드웨어에 전송
S1030
전용 하드웨어에서 HWIP 실행
S1040
실행결과로 데이터 업데이트 및
메인 코어에 실행완료 통지
S1050
메인 코어에서 업데이트된
데이터로 소프트웨어부분 코드 실행 재개
S1060
Yes
전용 하드웨어의 HWIP 실행 필요?
S1070
No
END

도면13


SW
SW/HW
인터페이스
HWIP 0
HWIP 1
HWIP 2
...
HWIP N
Shared
registers
하드웨어 컨트롤러
mem data
AHB bus

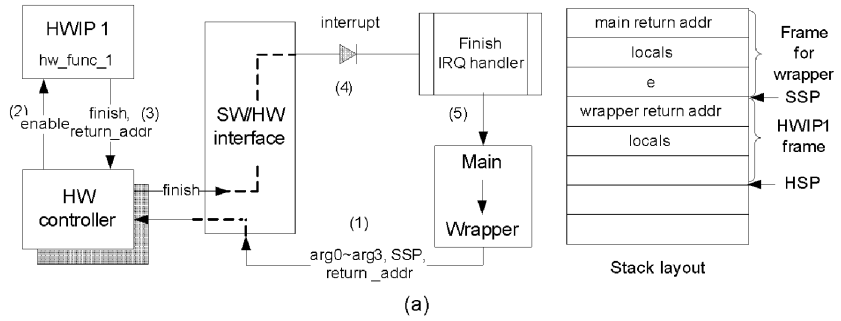
도면14



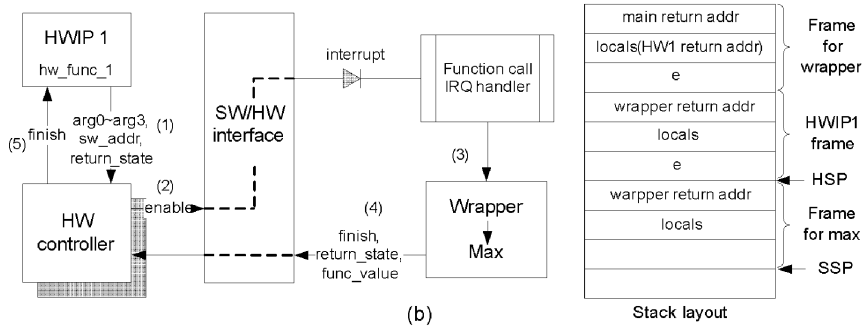
표시된 변환대상 C 코드

전처리된 코드

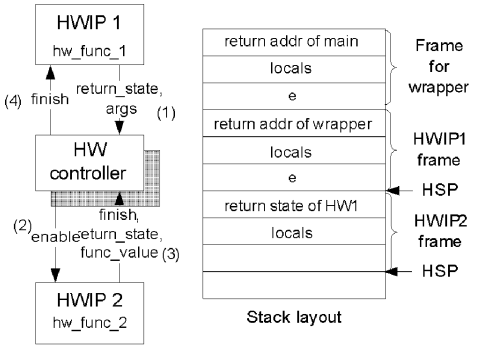
도면15



(a)



(b)

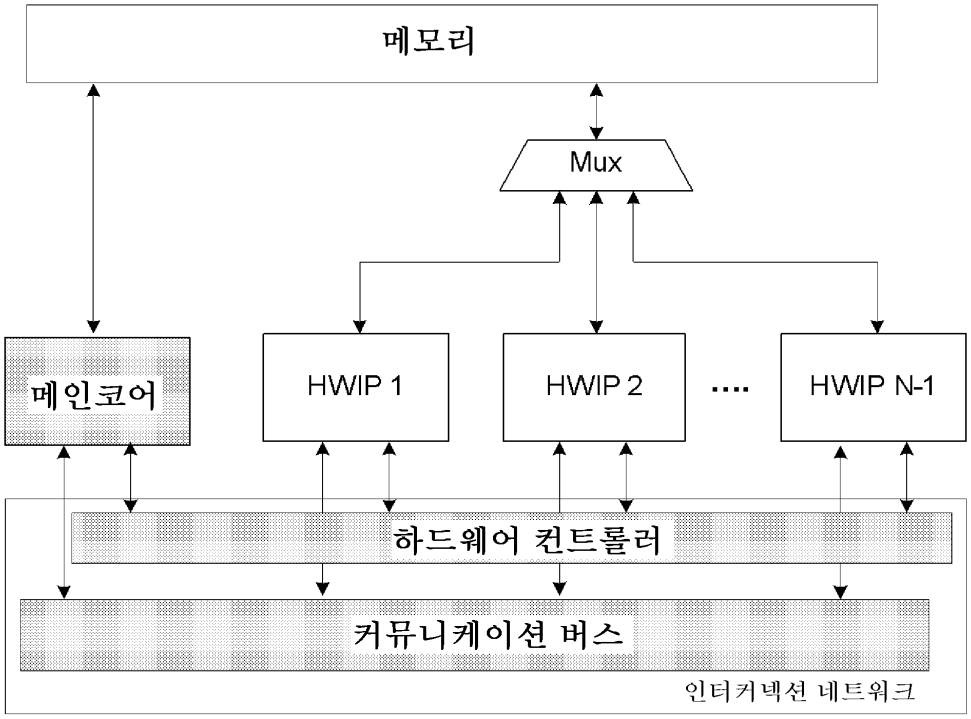


(c)

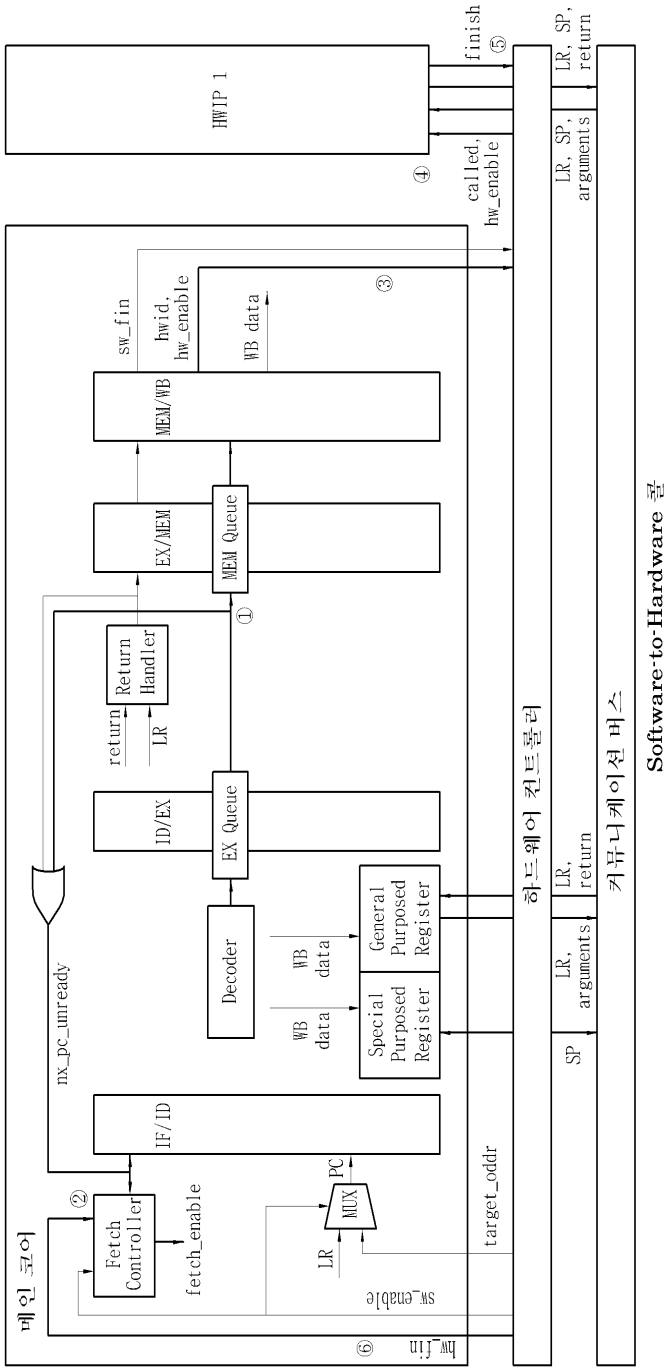
```
void wrapper (int id, int no_arg, void* arg0,. ...){
    write information to HWIP1 ;
    enable the HWIP1;

wait:     // wait for function call or finish
    while (!finish && !hw_called);
    if (finish)
        return;              // finished -> return

    // Process call to max() from HWIP1
    SSP = HSP;           // copy HSP to SSP
    jump to sw_addr;   // max's address
    hw_called = 0;       // Clear the func call
    copy max return value to HW controller;
    reset SSP to original value;
    announce HWIP1 that max() finished;
    goto wait;              // continue to wait
}
```

(d)

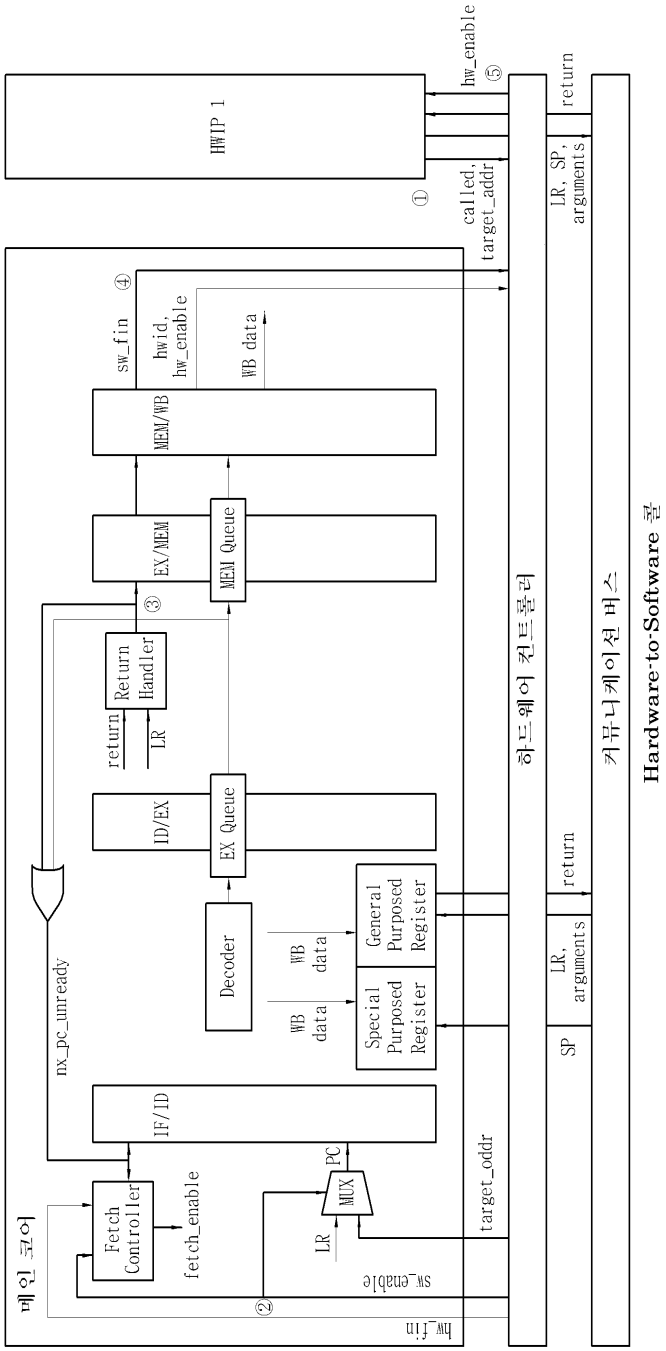
도면16


메모리
Mux
메인코어
HWIP 1
HWIP 2
....
HWIP N-1
하드웨어 컨트롤러
커뮤니케이션 버스
인터커넥션 네트워크

도면17


HWIP 1
finish
⑤
called,
hw_enable
④
LR, SP,
return
LR, SP,
arguments
③
sw_fin
hwid,
hw_enable
WB data
MEM/WB
EX/MEM
MEM Queue
①
return
Return
Handler
LR
ID/EX
EX Queue
Decoder
WB
data
General
Purposed
Register
WB
data
Special
Purposed
Register
nx_pc_unready
IF/ID
②
Fetch
Controller
fetch_enable
LR
PC
MUX
target_oddr
SW_enable
HW_fin
⑥
메인 코어
하드웨어 컨트롤러
LR,
return
LR,
arguments
SP
커뮤니케이션 버스
Software-to-Hardware 콜

도면18


HWTP 1
hw_enable
⑤
called,
target_addr
①
return
LR, SP,
arguments
sw_fin
④
hwid,
hw_enable
WB data
MEM/WB
EX/MEM
MEM Queue
③
return
LR
Return
Handler
ID/EX
EX Queue
Decoder
WB
data
WB
data
General
Purposed
Register
Special
Purposed
Register
nx_pc_unready
IF/ID
PC
MUX
LR
Fetch
Controller
fetch_enable
sw_enable
②
hw_fin
target_oddr
return
LR,
arguments
SP
메인 코어
하드웨어 컨트롤러
커뮤니케이션 버스
Hardware-to-Software 큐