論文98-35C-12-1

VHDL 컴파일러 후반부의 VHDL-to-C 사상에 관한 설계 및 구현

(A design and implementation of VHDL-to-C mapping in the VHDL compiler back-end)

孔 鎭 興 *, 高 亨 壹 *

(Jin-Hyeung Kong and Hyung-Il Goh)

요 약

본 논문은 VHDL 컴파일러 시스템에서 후반부의 VHDL-to-C 사상 과정을 설계 및 구현한 연구에 관하여 기술한다. 컴파일러 전반부가 VHDL 설계 프로그램으로부터 발생시킨 중간 형식의 분석 데이터는 컴파일러 후반부의 VHDL-to-C 사상을 통해서 VHDL 어의가 구현된 C 코드 모델로 변환된다. 기본적으로 VHDL 어의를 표현하기 위한 C 코드 모델은 선언부, 구축부, 초기화부 및 실행부의 4개 기능적 템플릿으로 구성된다. 사상 과정에서는 사상 단위와 기능분류에 따른 129개 C 사상 템플릿과 반복적 알고리듬을 통하여 터미널 정보를 이용해서 C 코드를 생성하게 된다. C 프로그램의 구성은 코드를 직접 템플릿으로 출력하거나, 생성된 코드를 데이터큐에 중간 저장시키고 상위사상 결과에 결합시켜서 이루어진다. 설계 및 구현된 VHDL-to-C 사상기는 Validation Suite의 96% VHDL 구문 구조에 대해서 100% C 코드 모델을 완벽하게 사상할 수 있음을 보였다. 또한 VHDL-to-C 사상의 성능에서 생성된 코드의 메모리 오버헤드가 해석기 방식보다는 작고 직접코드 방식보다는 크지만 VHDL 프로그램 크기에 대해서 완만한 증가 경향을 보이고 있으며, 사상처리 시간에서는 사상 메카니즘의 구현에서 최적화 및 개선이 요구됨을 나타내었다.

Abstract

In this paper, a design and implementation of VHDL-to-C mapping in the VHDL compiler back-end is described. The analyzed data in an intermediate format(IF), produced by the compiler front-end, is transformed into a C-code model of VHDL semantics by the VHDL-to-C mapper. The C-code model for VHDL semantics is based on a functional template, including declaration, elaboration, initialization and execution parts. The mapping is carried out by utilizing C mapping templates of 129 types classified by mapping units and functional semantics, and iterative algorithms, which are combined with terminal information, to produce C codes. In order to generate the C program, the C codes are output to the functional template either directly or by combining the higher mapping result with intermediate mapping codes in the data queue. In experiments, it is shown that the VHDL-to-C mapper could completely deal with the VHDL analyzed programs from the compiler front-end, which deal with about 96% of major VHDL syntactic programs in the Validation Suite. As for the performance, it is found that the code size of VHDL-to-C is less than that of interpreter and worse than direct code compiler of which generated code is increased more rapidly with the size of VHDL design, and that the VHDL-to-C timing overhead is needed to be improved by the optimized implementation of mapping mechanism.

---

* 正會員, 光云大學校 컴퓨터工學科
(Kwangwoon Univ., Computer Eng. Dept.)

※ 본 연구는 교육부 반도체분야 학술 연구 조성 (ISRC-95-E-2008) 및 광운대학교 '98 학술 연구, 반도체설계교육센터(IDEC) 사업의 지원을 받아 수행되었음

接受日字:1998年5月26日, 수정완료일:1998年11月23日

Ⅰ. 서 론

기존의 VHDL 설계 검증 시스템은 해석기(interpreter) 방식과 컴파일러(compiler) 방식의 시뮬레이션으로 나누어진다[1,2]. 컴파일러 방식에서는 VHDL 설계 프로그램을 컴파일하여 수행 가능한 코드로 변화시킨 후 시뮬레이터 커널(kernel) 및 라이브러리(library)와 링크시켜 VHDL 설계를 동작시키는 시뮬레이터를 발생하며, 해석기 방식은 VHDL 설계를 해석하여 고유의 의사코드(pseudo code)를 구성한 후 가상기계(virtual machine)인 별도의 해석기에서 라이브러리와 함께 시뮬레이션을 실행한다. 두 방식의 VHDL 시뮬레이션은 절대적인 우열을 판단하기 어렵고, 일반적으로 해석기 방식[3]이 디버깅 기능의 제공에 적합하여 설계 검증 사이클을 단축하는 장점을 가지며, 컴파일러 방식[4]은 빠른 시뮬레이션 속도를 제공하여 실행속도가 요구되는 대형설계에 적합하다고 평가되고 있다. 또한 최근 대형설계 검증에서 요구되는 성능을 위해서 컴파일러 방식에서 직접 원시코드를 생성하는 VHDL 시뮬레이터[5,6]들이 개발되었다. 이 같은 VHDL 검증 시스템의 연구 및 개발결과는 대부분 VHDL의 표준화 과정에 참여한 국외의 업체 및 대학에서 이루어지고 있으며, 국내에서는 표준안의 발표 이후에 연구가 시작되고 있어서 결과가 미흡한 실정이다. 실제로 VHDL 분석기[7,8,9] 및 VHDL 시뮬레이터[10,11]에 관한 연구가 발표되었으나, 개발결과 및 성능이 미흡하고 LRM[12,13] 사양의 지원범위가 제한되는 등의 문제에 대해서 해결의 여지가 많이 남아있는 실정이다.

본 연구에서는 VHDL 설계검증을 위한 VHDL 컴파일러 및 시뮬레이터를 개발하기 위하여, 앞서 개발한 VHDL 컴파일러 전반부(front-end)[9]의 출력결과인 중간형식(intermediate format, IF) 데이터를 입력으로 받아 목적코드로 사상을 수행하는 후반부(back-end)에 관하여 진행하였다. 이러한 후반부는 대형설계에 대해서 충분한 검증 속도를 제공할 수 있는 컴파일러 방식의 구현 및 다양한 호스트 환경에 대한 VHDL 시뮬레이션의 호환성 및 이식성을 유지하고자 C 언어를 실행 목적코드로 선택하였다. 개발된 VHDL 컴파일러의 전반부는 입력된 VHDL 코드에 대하여 어의 분석 및 중간형식 데이터로 저장하며, 컴파일러 후반부는 분석데이터의 VHDL-to-C 사상을 통해서 VHDL 시뮬레이션 어의를 갖는 C 코드 모델로 변환시키고 실행제어 커널 및 라이브러리와 링크시켜서 VHDL 설계를 검증하기 위한 시뮬레이터를 생성시킨다.

따라서 본 연구에서는 VHDL과 C 언어의 차이점을 효과적으로 해결하기 위한 처리방법을 설계 및 구현하였다. VHDL의 동형이의어(homograph) 및 다중정의(overloading) 특성은 C 언어에서 인식자 충돌(identifier conflict)을 발생시킨다. 이를 위하여 VHDL 컴파일러의 전반부는 기본적인 어휘요소(lexical element) 및 구문생성(syntactic production) 검사 이외에 계층구조/선언영역 및 가시성(visibility)/객체와 타입에 대한 어의검사(semantic checking) 기능을 추가하였다. 이러한 분석결과는 후반부에서 새로운 이름규칙(naming rule)을 적용하여 모든 인식자를 구분하도록 사상하였다. 또한 순차수행 언어인 C 언어로 VHDL의 병행수행 및 타이밍특성을 나타내기 위하여 프로세스의 스레드(thread) 및 신호의 타이밍정보를 관리하는 시뮬레이션 커널을 설계 및 구현하였으며, 전반부에서는 프로세스의 실행스택 및 신호의 드라이버(driver)를 분석하는 전처리 기능을 추가하였다. 이러한 분석결과는 후반부에서 각각의 프로세스마다 독립된 실행스택 및 커널과의 인터페이스로 사상되어지도록 설계하였다. 이렇게 구현된 VHDL-to-C 사상기의 동작은 C 코드를 생성시켜 VHDL 시뮬레이션 커널의 실행제어에 따라서 동작함을 확인함으로써 검증되었다.

본 논문은 VHDL 컴파일러 후반부에서 VHDL-to-C 사상과정의 설계 및 구현에 관해서 논한다. 먼저 VHDL의 어의 특성을 C 언어에 대해서 비교 분석하여 VHDL-to-C 사상에서 처리되어야 할 시뮬레이션 어의를 정리한다. 다음에 분석데이터로부터 C 코드 모델을 생성하기 위한 사상방법을 제안하고, VHDL 컴파일러 후반부의 VHDL-to-C 사상기를 설계 및 구현한 과정을 기술한다.

구현된 VHDL-to-C 사상기의 기능 및 성능을 검증하기 위하여 사상지원의 범위를 확인하고, 사상된 C 코드 모델의 메모리 오버헤드와 사상과정의 타이밍 오버헤드를 해석기 및 직접코드 생성방식의 상용 VHDL 검증 툴들과 비교해서 보인다. 마지막으로 본 연구에 대한 결론 및 추후에 해결해야 할 문제에 대해서 정리한다.

Ⅱ. VHDL과 C 언어

VHDL과 C 언어는 계층적 기술, 실행 메커니즘, 선언영역과 가시성(visibility), 타입 및 객체, 실행어 등의 큰 차이점을 갖고 있다. 이러한 차이점을 VHDL-to-C 사상에 적합한 데이터 구조로 표현하기 위하여, 전반부는 계층적 기술 및 선언영역과 가시성을 표현하기 위한 논터미널(nonterminal)과 객체 및 타입 등의 어의를 표현하기 위한 터미널(terminal)을 이용하여 AST(Abstract Syntax Tree)를 표현하였다. 또한 전반부는 어의정보를 효율적으로 검색하기 위하여 트리구조의 심볼테이블 및 타입테이블을 구성하며, 사상과정에서 검색을 최소화하기 위하여 어의정보를 논터미널을 이용하여 상향식으로 전달하는 어의전파(semantic propagation)를 포함시켰다. 본 연구에서는 전처리된 분석데이터를 바탕으로 사상과정에서 구현되어야 할 VHDL 시뮬레이션 어의를 C 언어로 모델링하였다.

1. 계층적 기술

LRM에 정의된 VHDL 설계단위의 구성 및 상호관계는 하드웨어에 대한 계층적인 모델링을 가능하게 하며, 객체 지향적인 모델링으로 캡슐화(encapsulation)와 분류(classification) 및 사례화(instantiation)등을 지원한다. 이러한 계층적 및 객체 지향적 모델링을 지원하는 VHDL에 비해서 C언어는 두가지 프로그램 파일(*.h, *.c)의 관계 설정만을 직접적으로 지원하고 있다. 따라서, VHDL의 계층성 및 상호 연관성에 대해서 C 언어는 두가지 파일의 상호포함 관계를 이용하여 VHDL 설계의 구성 어의를 기술해야 한다. 이를 위해서 컴파일러 전반부는 VHDL 설계단위의 구성을 분석하고 계층성 및 상호 연관성을 표현하는 데이터구조를 분석데이터에 구축하며, 후반부의 VHDL-to-C 사상에서는 VHDL 설계 및 사상단위에 대응된 C 파일들 사이의 상호관계를 설정하게 된다.

2. 선언영역 및 가시성

VHDL과 C 언어는 선언된 심볼에 대해서 블록 구조(block-structured) 언어에서와 같은 영역(scope) 및 가시성(visibility)의 기본규칙을 갖는다. 그러나 VHDL은 각 설계단위마다 독립적인 선언영역을 정의하며, 설계단위간의 종속관계가 선언영역의 상하위 계층성을 결정한다. 또한 VHDL 설계단위 안에서 블록 선언에 대한 중첩(nesting)과 서브프로그램 사이에 선언영역을 중첩시키는 것이 가능하며, 선언영역의 확장에서도 선택적으로 임의의 위치에 원하는 영역을 추가시킬 수 있다[9]. 이외에도 VHDL은 상·하위 선언영역에서 같은 인식자를 갖는 동형이의어(homograph) 선언을 이용할 수 있으며, 서브프로그램 및 열거형리터럴(literal), 수식표현(operator)에 대하여 다중정의(overloading)를 지원한다.

실제로 VHDL의 영역 및 가시성에 관한 모든 정보는 구문분석의 결과인 AST(Abstract Syntax Tree)에 포함되어야 한다. 컴파일러 후반부의 VHDL-to-C 사상에서는 이와 같은 분석데이터로부터 심볼의 영역 및 가시성의 계층 정보와 심볼의 동형이의어 및 다중정의 등의 이름 규칙(naming rule) 정보를 효과적으로 검색해서 C 언어의 프로그램에 적합한 심볼 형태로 변환시키게 된다.

3. 객체와 타입

VHDL은 하드웨어 설계를 효과적으로 기술할 수 있도록 C 언어에 비해서 다양한 객체(object), 타입(type) 및 리터럴(literal), 속성(attribute)과 연산자(operator)를 지원한다. VHDL의 객체는 식별가능한 인식자 및 타입이 허용하는 값을 갖으며, 신호(signal), 변수(variable) 및 상수(constant)의 3 종류로써 구성된다. 변수 및 상수는 C 언어와 같이 데이터 처리를 표현하는데 이용되나, 신호는 값의 지정과정에 타이밍 동작을 포함할 수 있어서 C 언어에서는 지원되지 않는 객체이다. VHDL의 변수와 상수 객체는 C 언어가 지원하는 객체를 그대로 이용하나, 신호 객체는 타입 및 속성, 타이밍과 값의 결정을 위한 항목 등으로 구성된 구조체로써 표현된다.

객체(object) 특성 및 연산(operation) 특성을 표현하는 VHDL 타입은 미리 정의된(predefined) 타입과 사용자 정의(user-defined) 타입으로 구분되며, 하드웨어 객체의 특성을 정확하게 표현하기 위해서 기본타입(basetype)에 범위(range) 및 색인(index)의 제약을 설정하여 종속타입(subtype)으로 세분화시키는 상하위 계층관계를 지원하고 있다. 이같은 VHDL의 타입을 표현하기 위해서는 타입의 기본정보, 제약정보 및 상하 종속관계 등을 나타낼 수 있는 트리 데이터구조가 이용된다. 분석된 타입 정보는 VHDL-to-C 사상 과정에서 심볼의 타입을 검색하거나 연산문(expression)의 타입을 결정하는데 사용한다. 특히 VHDL의 연산문에 대해서 연산자(operator) 이름과

피연산자(operand) 타입을 이용하여 리턴타입을 결정하며, 다중정의된 연산자를 식별하기 위한 타입추적 기능을 수행한다. 또한 다수의 연산자 및 피연산자들로 구성된 연산문의 경우, 리턴타입을 결정하는 과정에서 일어나는 모든 피연산자에 대한 타입 검색을 위해서 타입전파(type propagation)를 처리한다.

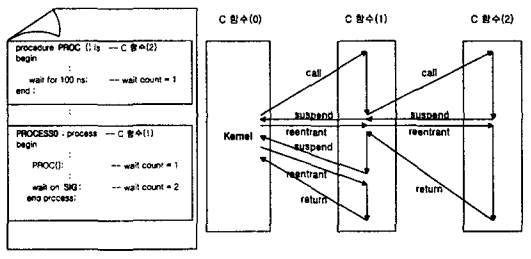
VHDL 값의 표기에는 정수(integer), 실수(floating point), 문자(character), 스트링(string), 비트스트링(bit string), 물리적(physical) 리터럴(literal) 등이 사용된다. 실제로 이와 같은 VHDL 리터럴의 사상은 컴파일러 전반부에서 처리하는 것이 효과적이다. 컴파일러 분석과정에서 정수와 실수 리터럴은 10진수 값으로 표현되며, 비트스트링 리터럴은 기수(base)를 2진법으로 통일시켜서 C 언어의 스트링으로 변환시킨다. 또한 물리적 리터럴은 단위를 고려한 10진수의 실수값으로 표현되고, 스트링 리터럴은 C 언어에서 그대로 이용된다. 다중정의가 허용되는 문자 리터럴은 열거타입(enumeration type) 리터럴 집합에서의 순서를 정수값으로 표현한다.

또한 VHDL은 객체의 이용과 관련되는 속성(attribute)과 연산자를 지원한다. VHDL의 속성은 객체에 대한 동작이나 상태를 표현하는 특성으로서 C 언어의 상수/변수/함수로써 표현될 수 있다. 값을 계산하는 연산문(expression)을 위해서 VHDL과 C는 각각 정의된 연산자(predefined operator)를 제공한다. VHDL은 C 언어보다 다양한 연산자를 지원하며, 사용자가 연산자(user-defined operator)를 정의하는 것도 가능하다. 또한 VHDL은 연산자 및 서브프로그램에 대해서 하나 이상의 기능을 정의하는 다중정의를 제공한다. 이와 같은 VHDL 연산자는 VHDL-to-C 사상에서 C 함수 호출로써 변환되는데, 다중정의된 함수들을 구분하기 위하여 리턴타입, 연산자 이름 및 피연산자들의 타입을 이용해서 고유의 함수 이름을 구성하게 된다.

4. 실행어의 및 스택관리

VHDL은 디지털 하드웨어를 기술하기 위해서 행위(behavior)/타이밍(timing)/구조(structure)의 세가지 모델링을 지원한다. 이에 비해서 C 언어는 하드웨어 행위를 모델링하는 것이 가능하나 시간(timing) 어의를 갖고 있지 않으며, 하드웨어 구조의 직접적인 기술 방법을 지원하지 않는다. 따라서 VHDL의 구조 기술에 대한 정적 구조체를 C 모델링하는 것이 요구되며, 시간 어의를 위해서 VHDL의 병행 수행 의사프로세스(pseudo process), 다중스택 구조 및 신호 객체를 관리할 시뮬레이션 커널 등을 C 모델링하여야 한다. 이와 같은 C 모델링을 위해서 컴파일러 전반부에서는 VHDL 설계 문장의 어의를 행위/구조 및 병행/순차 측면에서 식별하고, 병행 수행에서 제어흐름을 분석하며, 각 수행 스레드(thread)마다 독립적인 스택 구조를 구현하기 위한 심볼 데이터구조를 구성하여야 한다[9].

VHDL의 병행문과 순차문은 서로 다른 실행 메커니즘을 갖기 때문에 VHDL-to-C 사상 과정에서는 별도로 취급된다. VHDL의 각 병행문은 시뮬레이션의 수행 과정에서 커널에 대한 독립적인 실행단위로써 사상되며, 순차문은 각 병행 실행단위에서 순서대로 처리되는 과정을 기술하게 된다. VHDL의 전반적인 병행동작은 C 의사프로세스 함수에 대한 시뮬레이션 커널의 실행제어에 의해서 이루어지나, 병행 프로세스문 실행과정에서 동기화를 기술하는 순차 대기문(wait statement)은 C 의사프로세스 함수의 병행 실행과정 중에 정지(suspend) 및 재진입(reentrant) 과정을 수행하도록 그림 1의 제어 흐름이 VHDL-to-C 사상에서 구현되어야 한다. 대기문을 포함한 VHDL 문장에 대응된 C 함수는 대기문을 처리하기 위해서 수행을 중지하고 커널로 문맥 전환(context-switching)을 하게 되며 대기문의 조건이 처리된 후에 수행이 중지되었던 반대 순서대로 재진입하게 된다. 이때 수행 중에 상위 C 함수로의 문맥 전환에서는 수행 상태를 저장시키기 위한 별도의 동적 스택이 요구되며, 대기문의 처리 후에 재진입을 위해서는 중지되었던 위치로 수행 상태를 복구하는 것이 필요하게 된다.



(a) VHDL 대기문 (b) C 커널과 의사 프로세스 실행 과정

그림 1. VHDL 대기문에 대한 C 의사 프로세스 함수의 정지 및 재진입 과정

Fig. 1. Suspend and reentrant procedures of C pseudo processes for VHDL wait statements.

계층적인 VHDL 설계단위 안의 모든 병행문들은 다중 스레드(multi-thread)의 제어 흐름을 가지며, 각 수행 스레드는 객체 및 상태의 관리를 위해서 독립된 스택을 유지 및 관리해야 한다. 반면에 C 프로그램은 운영체제의 프로세스 단위로 처리되며, 순차적 단일 스레드(single-thread)의 제어 흐름을 갖고, 데이터 관리를 위한 단일 스택 구조만을 동적으로 제공한다. 본 연구에서는 병행적인 C 의사 프로세스 함수들의 모든 심볼들을 실질적으로 시뮬레이션 커널에서 유지 및 관리할 수 있도록 전역(global) 선언 영역에서 구분시켜 관리하는 방법을 이용하고자 한다. 이를 위해서는 인식자(identifier)들 사이의 충돌을 방지해야 하는데, VHDL 심볼의 선언영역 및 계층성을 이용하면 모든 인식자를 구분하는 것이 가능하다. 즉 심볼의 계층적인 선언영역의 정보를 인식자의 접두어로써 나타내면 인식자들의 충돌을 방지할 수 있다. 예를 들어, 그림 2의 VHDL 프로그램에서 Process1의 S와 Process2의 S는 A_S 및 A_B_S로 구분된다.

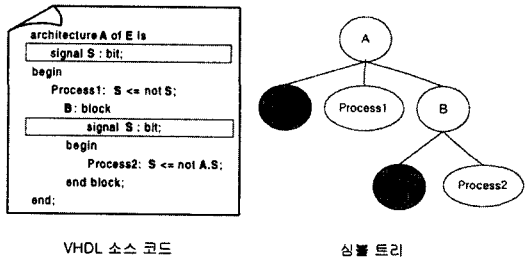


그림 2. 인식자의 충돌을 방지하기 위한 심볼 관리
Fig. 2. Symbol management to avoid the collision of identifiers.

5. 신호 및 타이밍

VHDL에서 신호는 타이밍 정보 및 결과값으로 구성된 드라이버(driver)를 통해서 값이 변화되는 특성을 갖고 있다. 또한 드라이버는 다수의 프로세스에서 병행적으로 발생될 수 있으며, 이렇게 발생된 드라이버들은 분해 함수(resolution function)에 의해 값을 결정하게 된다. 최종적으로 값의 변화라는 이벤트가 발생한 신호들은 감지신호(sensitivity list)로 등록된 프로세스들을 활성화(activation)시키게 된다.

그러나 C언어는 신호를 위한 타이밍 정보와 동기화가 요구되는 값의 결정 메카니즘 및 활성화 기능을 지원하지 않는 차이점을 갖고 있다. 이러한 문제점을 해결하기 위해서 드라이버들의 스케줄링 및 이벤트 신호와 관련된 프로세스들의 활성화를 제어하는 시뮬레이션 커널의 설계가 요구된다. 또한 VHDL 설계에서는 시뮬레이션 커널과의 인터페이스 사상을 통하여 드라이버의 스케줄러 등록 및 개별 프로세스마다 자신의 실행스택 및 재진입 포인트를 커널에 등록하는 기능이 필요하다.

VHDL에서 신호 값의 결정 메커니즘을 해결하기 위하여 그림 3과 같이 이벤트구동(event-driven) 방식으로 처리하는 시뮬레이션 커널을 설계하였다. 이러한 커널은 각 프로세스에 대한 드라이버 리스트, 신호의 분해 함수 및 신호가 구동시킬 프로세스 리스트를 구축(elaboration) 과정에서 요구한다. 또한 커널은 프로세스 수행 과정에서 발생한 신호대입(signal assignment) 요구를 입력으로 받아서 해당 신호에 대한 드라이버를 갱신하고 타임 휠(wheel)에 등록한다. 시뮬레이션 시간 t에서 커널은 타임 휠의 타임 슬롯에 등록되어 있는 드라이버들을 검색하여 신호의 트랜잭션을 수행하며, 이 과정에서 이벤트가 발생된 신호들은 등록된 C 코드 모델의 의사프로세스를 실행시킴으로써 시뮬레이션이 진행되어진다.

이러한 커널과의 인터페이스를 사상을 위하여 전반부는 신호대입문(signal assignment)을 분석하여 각각의 프로세스를 구동시킬 감지 신호(sensitivity list) 유도 및 타이밍 정보와 결과값을 드라이버로 표현하는 전처리 과정을 수행하게 된다.

후반부는 구축(elaboration)과정에서 해당 프로세스와 신호의 관계 설정 및 드라이버 생성을 수행하고, 실행(execution)과정에서 신호대입문(signal assignment)에 의해 생성된 드라이버를 커널에 등록하도록 모델링하였다. 또한 활성화된 프로세스를 위하여, 실행스택에 재진입 포인트를 설정하고 커널로 복귀하기 위한 메커니즘과 재진입 과정에서 실행포인트 위치로 이동하기 위한 메커니즘을 의사 프로세스마다 사상하도록 설계하였다.

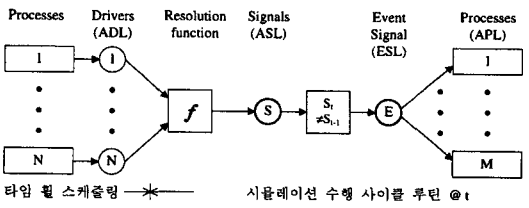


그림 3. 신호 값을 결정하는 메커니즘
Fig. 3. Deterministic mechanism for signal values.

Ⅲ. VHDL-to-C 사상기 설계

VHDL 컴파일러 후반부의 VHDL-to-C 사상기는 VHDL 설계모델을 분석한 결과인 중간 형태의 AST, 심볼트리와 타입트리, 속성리스트를 입력받아 C 코드 모델로 변환하는 역할을 수행한다. 본 장에서는 VHDL-to-C 사상 방법의 설계에 대해서 기술하고, VHDL 컴파일러 후반부에 구현된 VHDL-to-C 사상기의 동작 및 구성에 대하여 설명한다.

1. 사상 방법의 설계

VHDL-to-C 사상의 설계는 사상 단위의 분류, C 코드 모델의 설계와 사상 및 코드 생성 방법의 설계로 나누어진다.

a. 사상단위와 데이터큐

VHDL-to-C의 사상단위는 표 1과 같이 VHDL LRM[12,13]에 기술되어 있는 구축과 실행의 기능을 수행하는 VHDL 문장들과 특수한 기능의 문장 및 구문들로서 6개군의 총 43개 단위로써 이루어진다. 각 사상 단위는 시뮬레이션 어의에 따라 C 코드 모델의 선언/구축/실행의 기능으로 분류되며, 고유의 사상 및 코드생성 방법이 적용된다.

표 1. VHDL 문장의 사상 단위 분류
Table 1. Classification of mapping units for VHDL statements.

사상단위	VHDL 문장
선언문군	AliasDeclaration, AttributeDeclaration, ComponentDeclaration, ConstantDeclaration, VariableDeclaration, FileDeclaration, SignalDeclaration, TypeDeclaration, SubtypeDeclaration
설계단위군	DesignUnit, EntityDeclaration, ArchitectureBody, PackageDeclaration, PackageBodyDeclaration, ConfigurationDeclaration, BlockConfiguration, ComponentConfiguration, ConfigurationMap
병행문군	BlockStatement, BlockHeader, BlockStatement01, ProcessStatement ConcurrentAssertionStatement, ConcurrentSignalAssignStatement, ConcurrentProcedureCall, ConcurrentConfiguration, GenerateStatement,
순차문군	AssertionStatement, CaseStatement, ExitStatement, IfStatement, LoopStatement, NextStatement, ReturnStatement, SequentialProcedureCall, SignalAssignmentStatement, VariableAssignmentStatement, WaitStatement
서브프로그램군	SubprogramSpecification, SubprogramBody
특수 처리군	Allocator, AttributeSpecification, DisconnectionSpecification, Expression

b. C 코드 모델의 기능적 템플릿

VHDL 하드웨어 구조 및 동작의 어의를 C 언어로써 기술하기 위하여 그림 4와 같은 C 코드 모델의 기능적 템플릿(template)이 설계되었다. 기능적 템플릿은 선언부(declaration)와 초기화부(initialization), 구축부(elaboration) 및 실행부(execution)의 4 부분으로 구성되어 VHDL 설계의 어의를 표현한다.

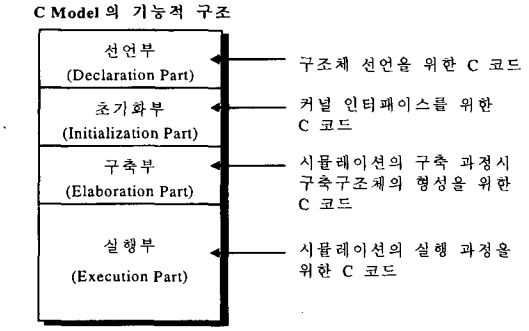


그림 4. 생성된 C 모델의 기능적 구조
Fig. 4. Functional template of C code model.

선언부에서는 모든 객체(object)와 타입(type) 및 속성(attribute)의 구조체를 선언한다. 객체들은 작업 라이브러리에서부터 선언되어진 위치까지의 범위(scope)를 포함한 이름으로 표현된다. 실제로 선언부의 구조체는 구조적 병행문들에서 사용되는 객체를 위한 구축 구조체와 동작적 병행문들에서 사용되는 객체를 위한 구축 구조체, 그리고 순차문의 서브프로그램에서 사용되는 호출스택 구조체로 구분된다.

초기화부는 사상된 C 코드 모델과 시뮬레이션 커널간의 인터페이싱 C 코드로 구성된다. VHDL-to-C 사상기는 초기화부에 시뮬레이션 과정에서 구축(elaboration)이 일어날 수 있도록 최상위 설계단위 구축 함수의 이름을 등록한다. 따라서 커널은 시뮬레이션할 때 최상위 구축함수를 호출함으로써 별도의 관리 없이 하위 구축함수를 순차 호출함으로써 모든 구축을 수행하게 되며, 이와 함께 프로세스와 신호 객체의 초기화가 함께 이루어진다.

구축부는 설계단위, 선언문, 병행문에 대한 정적 구축(static elaboration)과 순차문중 서브프로그램에 대한 동적 구축(dynamic elaboration)으로 이루어진다. 정적 구축과 동적 구축은 상위 구축함수가 하위 구축함수를 호출함으로써 이루어지며 진행은 다음과 같다[9,14].

1) 설계 계층성(design hierarchy) 정적 구축

VHDL 설계단위의 계층성 및 상호 관계는 그림 5와 같은 C 코드 파일간의 포함 관계를 통해서 표현되며, 시뮬레이션 과정에서는 각 설계단위간의 계층성 구축은 상위 구축 함수가 하위 구축 함수를 순차적으로 호출하여 구축 구조체를 형성 및 참조함으로써 이

루어진다.

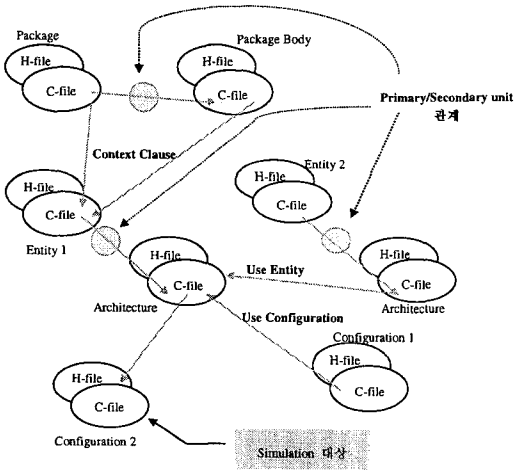


그림 5. VHDL 설계단위의 C 파일간 포함 관계
Fig. 5. Relationship between VHDL design units in C.

2) 선언문의 정적 구축

선언문 구축은 타입 및 객체 선언문에 대하여 이루어진다. 타입에 대한 구축은 각 타입에 대해서 객체(object)를 구축할 때 이용될 구축함수를 미리 정의한다. 객체의 구축은 객체 부류(object class), 파일 선언(file declaration) 및 연결 선언(interface declaration)에 대하여 이루어진다.

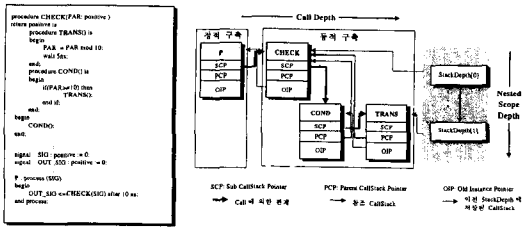


그림 6. 호출스택 구조체를 이용한 동적 구축
Fig. 6. Dynamic elaboration with the call stack structure.

3) 병행문 및 순차문 구축

순차문 서브프로그램은 실행부에서 동적 구축을 수행하기 위하여 호출스택(call stack) 구조체로 이루어진다. 병행문은 동작 병행문과 구조 병행문, 블록문으로 구분되며 정적구축 구조체를 갖는다. 프로세스 구축 구조체 형태로 사상된 C 모델은 선언부에 위치하며, 프로세스 함수의 형태로 사상된 C 모델은 실행부에 위치한다. 그리고 구축 함수 형태로 사상된 C 모델은 구축부에 위치하게 된다.

시뮬레이션을 위한 실행부는 병행문에 대해 사상된 프로세스 함수와 순차문중 서브프로그램에 대해 사상된 서브프로그램 함수로 구성되어 있다. 또한 프로세스가 서브프로그램를 호출하면, 서브프로그램 함수는 선언부에 있는 호출스택 구조체를 참조하여 그림 6과 같이 동적 구조체를 형성 및 참조하면서 서브프로그램 함수내의 순차문이 실행되어지도록 설계되었다.

c. 사상 및 코드생성 설계

VHDL 문장 및 구문으로부터 분류된 표 1의 각 사상 단위를 C 코드 모델의 프로그램으로 변환하기 위해서는 사상단위의 시뮬레이션 어의를 C 코드로 기술하는 사상 방법이 요구되며 하나 이상의 사상 단위들을 C 코드 모델로 변환하기 위한 생성 방법이 필요하게 된다. 본 연구에서는 사상 단위의 시뮬레이션 어의를 C 코드 모델의 기능별로 구분하여 표 2와 같은 사상 및 코드생성 방법을 설계하였다.

표 2. 사상 메카니즘의 분류
Table 2. Classification of mapping mechanism.

			기능 모델 단위											
			선언부			구축부			실행부			초기화부				
사상 단위	선언문	타입 선언	O		O		O							Template 적용	사상 방법
		객체 선언		O			O								
		기타 선언		O			O								
	설계 단위	패키지 선언			O			O					O		
		패키지 몸체			O			O					O		
		엔터티 선언			O			O					O		
		아키텍쳐 몸체			O			O					O		
		구성 선언			O			O					O		
	병행문	블록문	O	O		O	O								
		프로세스문	O	O		O	O		O						
		구조적 병행문	O	O		O	O								
		행위적 병행문	O	O		O	O		O						
	순차문	신호 지정문				O			O						
		대기문				O			O						
		프로시져 호출문	O			O			O						
		반복문	O			O			O						
		기타 순차문				O			O						
	서브프로그램	부프로그램 선언		O											
		부프로그램 몸체					O			O					
	특수 처리	내재적 선언	O			O									
		명세문				O									
		Expression				O			O					알고리즘적 반복	
		Name	O			O			O			O			
			직접	중간	결합	직접	중간	결합	직접	중간	결합	직접	중간	결합	
			코드 생성 방법												

VHDL 시뮬레이션 어의를 C 코드로 사상하는 과정은 사상 단위의 어의가 정형적(formal)인가 비정형적(unformal)인가로 구분하여 설계되었다. VHDL 이름 및 연산표현과 같이 정형적인 어의를 갖는 사상 단위에 대해서는 반복적인 알고리듬을 이용한 사상 방법을 적용하였으며, 비정형적인 VHDL 문장 및 구문들 각각에 대해서는 C 코드 모델의 템플릿을 구성하였다. 실제로 선언문에서는 객체 구조체의 선언과 객체의 구축과정을 모델링하였으며, 설계단위에서는 구축 구조체의 선언, 구축 구조체 및 선언 객체들의 구축 및 병행문 구축함수의 호출을 통한 초기화과정을 모델링하였다. 프로세스를 비롯한 행위적 병행문들은 시뮬레이션 동안 무한반복 수행되어야 하기 때문에, 프로세스 구조체의 구축과 대기/복귀를 위한 수행지점으로의 이동 기능을 모델링하였다. 또한 블록문과 구조적 병행문들은 구축구조체 선언 및 구축과정을 수행함으로써 상하위 설계단위간의 연결관계가 유지되도록 모델링되었다. 서브프로그램은 구축함수, 실행함수 및 소멸함수의 기능 단위로써 설계되었다. 구축함수는 구축 구조체를 생성하여 상위 호출함수의 구축 구조체와 연결하는 과정을 수행하며, 실행함수는 VHDL 프로그램에 설계된 동작 과정을 수행하게 된다. 또한 실행함수가 종료된 후에는 해당 구조체의 소멸 및 상위 호출함수로의 복귀를 처리하기 위해 소멸함수가 수행된다. 순차문은 프로세스 및 서브프로그램의 수행과정에 포함될 수 있도록 독립적으로 모델링되었다. 순차 신호대입문(signal assignment)은 신호 객체의 시간지연 특성을 커널이 관리할 수 있도록 지원함수를 사용하였으며, 반복문(loop statement)의 경우 루프변수의 내재적(implicit) 선언 및 다중분기(multi-branch) 기능을 모델링하였다. 대기문의 경우는 감지신호의 사건 발생 여부를 확인하여 재진입/복귀의 판단을 결정하기 위해서, 대기문에 의한 종료 상태를 상위 호출함수에게 알리는 기능과 재진입에서는 수행위치로의 분기 및 수행기능을 모델링하였다. 또한 기타 순차문인 변수대입문(variable assignment), IF 조건문, Case 조건문 등은 각각의 특성을 고려하여 템플릿이 설계되었다.

사상 단위에 대한 C 코드생성 방법으로는 결합(combined) 사상과 중간(intermediate) 사상 그리고 직접(direct) 사상이 설계되었다. 중간사상은 상위 사상 단위가 하위 사상 단위의 정보를 필요로 하는 경우를 위해서 하위 사상 단위를 데이터 큐에 미리 C 코드로 사상시키는 것을 말한다. 중간사상의 경우는 표현이나 범위 값을 갖거나 상위의 사상 단위에 포함되어야 할 경우에 행하여지며, 최상위 사상 단위인 설계단위(design unit) 문장들을 제외한 모든 문장들에 대해서 중간사상을 수행한다. 결합사상은 상위 사상 단위의 처리에서 하위 사상 단위의 정보가 요구되는 경우에 행해지며, 상위 사상 단위의 사상된 C 코드에 데이터 큐에 있는 하위 사상 단위의 C 코드 중 필요한 정보를 호출해서 사상시키는 것을 말한다. 실제로 결합사상은 최상위 설계 단위 문장들과 순차문을 포함한 병행문 등과 같이 하위 사상을 포함해야 하는 경우의 사상 단위에서 이루어진다. 그리고 직접사상은 현재의 사상 단위만으로 완전한 C 코드가 구성될 수 있는 경우에 행하여진다. 따라서 선언의 의미를 가지고 C 모델의 기능별 구조 중 선언부(declaration part)에 사상되는 경우나 실행부에 있는 서브프로그램 함수와 같이 하위 사상 단위의 정보가 필요 없이 사상 단위만으로 완전한 사상이 이루어질 수 있는 경우에 행해진다. 표 2에서는 각 사상 단위에 대해서 직접/중간/결합 코드생성 방법이 C 코드 모델의 기능별로 어떻게 이용되는가를 나타내고 있다.

2. 사상기의 구성 및 동작

VHDL-to-C 사상기는 VHDL 컴파일러 전반부가 설계기술을 분석한 중간(IF) 데이터를 입력으로 C 코드 모델로 변환하는 역할을 수행한다. 설계된 VHDL-to-C 사상기의 구성 및 동작이 그림 7에 나타나있다. CodeGen은 전반부에서 생성한 IF 데이터를 설계단위별로 구분하여 Mapper로 보내고, MapSupport에서 생성된 C 코드를 구분하여 *.c 와 *.h 파일로 발생시키는 역할을 담당한다. 또한 입력받은 설계단위가 설계의 최상위 모델인 경우에는 시뮬레이션에서 커널 인터페이스를 통한 구축이 일어나도록 초기화부에 최상위 설계단위의 구축 함수를 호출하는 C 코드를 사상한다. Mapper는 CodeGen에서 입력받은 각 설계단위의 AST를 상향식 방식으로 순회한다. Mapper는 134개의 VHDL 논터미널에 대하여 트리검색 함수를 가지고 하위 터미널을 찾아서 데이터큐에 포인터로 연결시키는 기능을 수행하며, 사상 단위를 만나면 MapSupport에 있는 사상 함수를 호출하여 실제 VHDL-to-C 사상이 이루어지도록 한다. 데이터큐에는 Mapper의 논터미널 트리검색 함수가 검색한 터미널의 주소, 하위 논터미널이 중간 사상된 C-코드, 그

리고 데이터의 분류 번호가 저장되어 있다. 실제로 데이터의 분류는 터미널의 종류와 중간 사상된 C-코드의 기능별 구분에 따라서 64개로 이루어진다. Mapper의 호출을 받은 MapSupport는 입력된 사상 단위를 43개 단위 중에서 구분하고, 각 사상 단위의 시뮬레이션 어의를 C의 기능적 템플릿 모델이나 반복적 알고리듬에 따라서 선언과 구축 및 실행으로 분류하여 C 코드로 기술하고 사상 단위의 상호연관성을 고려하여 직접과 중간 및 결합 방법 중에서 코드생성 방법을 선택한다. 사상단위 구분과 C 모델의 기능별 분류에서는 사상 템플릿 모델로서 준비되어 있는 129개 C 코드 형식 중 하나를 결정하여 C 코드 사상을 수행하며, 사상 과정에서 데이터 큐를 참조하여 Mapper가 저장한 터미널 정보나 임시 저장되어 있는 중간 사상의 C 코드를 검색하여 결합시킨다. 또한 직접 및 결합 사상 방법이 적용된 C 코드는 Codegen의 출력부로 보내지나, 중간 사상 방법으로 생성된 C 코드는 데이터 큐에 저장된다. 실제로 VHDL-to-C 사상기가 AST 분석 데이터에 대하여 사상을 수행한 구체적인 예가 그림 8에 나타나있다. [ ] 는 구조체의 형태로 사상됨을 의미하며 { }는 함수 형태로 사상됨을 나타낸다.

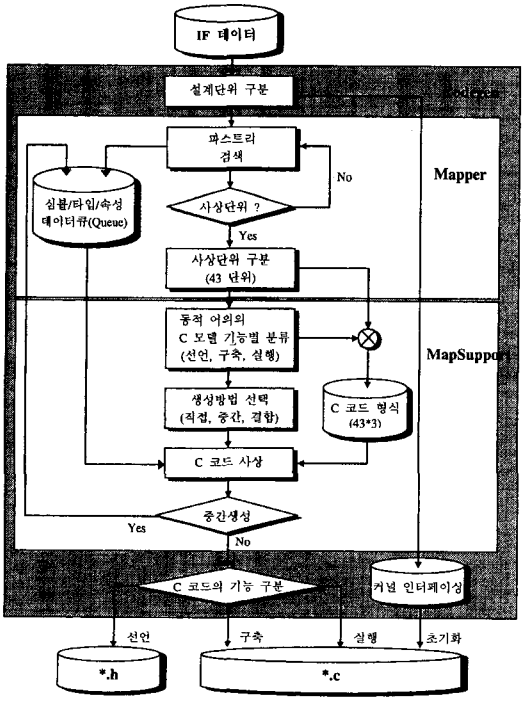


그림 7. VHDL-to-C 사상기의 구성 및 동작
Fig. 7. Organization and operation of VHDL-to-C mapper.

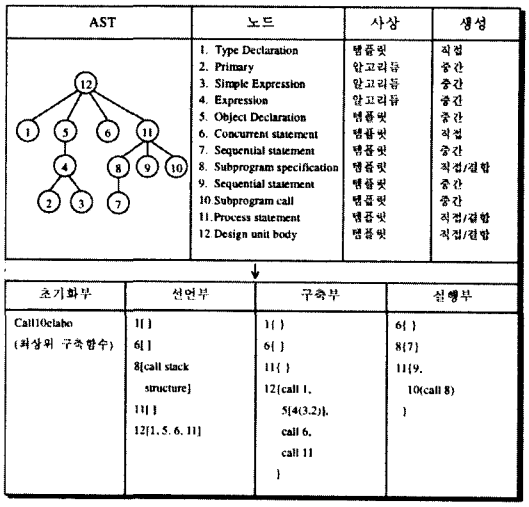


그림 8. 사상단위 AST에 대한 사상 결과
Fig. 8. Mapping results for AST of design unit.

Ⅳ. 실험 및 고찰

본 논문의 VHDL-to-C 사상기는 개발되고 있는 VHDL 컴파일러 후반부에 포함되어 SuperSparc 호환 기종(Axil 320)의 운영체제 SunOS 5.4 환경에서 디버깅되고 있다. 현재 각종 VHDL 설계 benchmark를 통해서 디버깅이 진행되고 있는데, VHDL-to-C 사상기의 실험에서는 VHDL 컴파일러 전반부와 시뮬레이션 커널이 요구된다. 본 연구에서는 VHDL 프로그램을 분석하여 VHDL-to-C 사상기의 입력 데이터를 생성하는 컴파일러 전반부[9]와 분석 데이터로부터 생성된 C 코드 모델을 실행 제어하기 위한 컴파일러 후반부의 시뮬레이션 커널[14]을 이용하여 VHDL-to-C 사상기에 대한 검증을 진행하였다. 설계 및 구현된 VHDL-to-C 사상기의 실험은 LRM[12,13]에 대한 사상 지원범위(coverage)를 확인하기 위해서 분석된 VHDL 설계데이터를 C 코드로 사상하고 컴파일된 VHDL 시뮬레이터의 동작을 검증하였으며, 사상기의 성능을 평가하기 위해서 사상결과인 C 코드 모델의 생성시간 및 메모리 오버헤드를 상용 VHDL 컴파일러의 결과에 대해서 비교하여 진행되었다.

VHDL-to-C 사상의 지원 범위를 확인하기 위해서 VHDL Validation Suite[15]를 이용하였다. Validation Suite를 활용한 이유는 VHDL에 정의된 모든 구문 구조를 포함한 테스트프로그램으로 구성되어 있기 때문이다. 실제로 사상기에 suite의 테스트오브젝트

를 입력시켜서 사상결과를 VHDL 시뮬레이터로 생성시킴으로써 발생된 C 코드 모델과 VHDL-to-C 사상의 기능을 검증하게 된다. VHDL Validation Suite에는 3225개의 VHDL 소스 코드가 포함되어 있으며, 3225개의 소스 코드 중에서 1546개는 각종 오류를 포함하고 있는 코드이다. 본 실험에서는 1606개의 VHDL 코드에 대해서 VHDL-to-C 사상 실험을 수행하였는데, 이때 1606개의 코드는 VHDL 분석기[9]가 중간형태 AST 데이터를 성공적으로 생성한 올바른 코드에서의 1587개 및 오류 코드에서의 19개 테스트 suite로 구성된다. 표 3은 VHDL Validation Suite에 대한 VHDL-to-C 사상의 실험 결과를 나타내고 있다. 이 실험을 통해서 본 논문의 VHDL-to-C 사상기가 Validation Suite의 분석된 VHDL 구문 구조를 C 언어로 완벽하게 처리할 수 있음을 확인하였다.

표 3. VHDL Validation Suite에 대한 실험 결과

Table 3. Experimental results for VHDL Validation Suite.

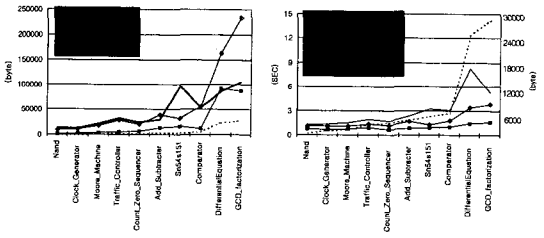
VHDL Validation Suite (VHDL'87)		올바른 C 코드 생성	비 율
올바른 코드[1]	1587	1587	100 %
실행 시간 오류를 포함한 코드[2]	19	19	100 %
정적 오류를 포함한 코드[3]	1523	-	-
계	3129[4] (3225)[6]	1606[5]	100 % (96%)[6]

1) : 올바른 코드 중에서 전반부에서 오류가 검출된 92개를 제외한 나머지 코드
2) : 실행 시간 오류가 포함된 코드 중에서 전반부에서 오류가 검출된 4개를 제외한 코드
3) : 정적 오류가 포함된 코드는 실험에서 제외
4) : 3225개의 코드 중에서 AST 생성에 실패한 96개의 코드(올바른 코드 중에서 92개, 실행 시간 오류 중에서 4개)를 제외한 나머지 코드
5) : 정적 오류가 포함되어 중간 형태의 AST가 없는 코드와 전반부에서 오류가 검출된 코드를 제외한 VHDL 코드에 대한 실험 결과
6) : 전반부의 지원범위를 함께 고려했을 때의 VHDL-to-C 사상 범위

또한 컴파일러 후반부의 VHDL-to-C 사상기에 대한 성능 평가를 위해서 사상 처리 시간 및 사상된 C 모델의 크기를 측정하였다. 성능평가 실험에 사용된 VHDL 입력은 기능 회로(function circuit)를 모델링한 코드로써 여러 VHDL 관련 도서의 예제 중에서 벤치마크 코드 10개를 이용하였다. 표 4는 10개 VHDL 테스트 프로그램에 대한 본 연구의 VHDL-to-C 사상기, Mentor사의 QuickSimII[5] 및 Synopsys사의 vhdlsim[3]의 실험 결과를 나타내고 있다. 실험에서는 직접코드 사상방식을 이용한 QuickSim과 해석기 방식의 vhdlsim에 대해서 C 언어를 목적코드로 컴파일링하는 본 연구의 VHDL-to-C 사상기를 사상시간과 메모리 오버헤드 측면에서 비교하였다. 또한 그림 8은 VHDL 코드의 크기 증가에 따른 세가지 컴파일러의 성능 변화를 사상코드 크기 및 코드 사상시간에 대해서 각각 나타내고 있다. 표 4와 그림 9를 통해서 본 연구에 대한 다음의 결과를 얻을 수 있었다.

- 사상된 코드의 메모리 오버헤드 면에서 본 연구의 VHDL-to-C 사상기는 직접코드 방식의 QuickSim보다는 크나 해석기 방식의 vhdlsim에 비해서는 다소 작은 코드생성을 가져올 수 있었다.
- 사상된 코드의 메모리 오버헤드에서 해석기 방식의 vhdlsim은 VHDL 설계가 커짐에 따라서 코드의 크기가 급격히 증가하고 있으나, 직접코드 방식의 QuickSim 및 VHDL-to-C 사상기에서는 완만한 크기의 증가를 보이고 있다.
- 직접코드 사상방식에 비해서 VHDL-to-C 사상방식이 매우 비효율적이라는 일반적인 인식은 VHDL 설계의 크기가 작은 경우에 명확하게 나타나고 있으나, VHDL 설계가 커짐에 따라서 VHDL-to-C 사상의 효율성은 점차 개선되는 경향을 보이고 있어서 비효율성이 C 사상 템플릿 구성에 요구되는 기본 오버헤드에 따른 것으로 판단된다.
- 목적코드를 생성하기 위한 처리시간에서 본 연구의 VHDL-to-C 사상방식은 해석기 및 직접코드 생성방식에 비해서 큰 오버헤드를 요구하며, VHDL 프로그램의 크기가 증가함에 따라서 처리시간의 증가 속도가 커지는 경향이 발견되고 있다. 특히 "Differential Equation"과 같이 연산처리에 대한 사상에서는 급격한 사상시간의 증가를 보이고 있다. 이는 본 연구의 사상 메카니즘인 AST 트리 검색을 통한 사상방법이 효과적이지 못한 한계성을 나타내는 것으로써 이에 대한 보완이 필요할 것으로 판단된다.

- 사상시간에서 직접코드 방식이 해석기 방식 및 VHDL-to-C 사상방식에 비해서 우수한 것으로 나타나고 있다. 그러나 VHDL 코드의 크기 증가에 대한 사상시간의 증가율에서는 직접코드 방식, 해석기 방식, VHDL-to-C 사상방식의 순서를 보이고 있다. 즉 VHDL-to-C 사상방식은 목적코드의 메모리 오버헤드에서와 같이 사상시간에서도 VHDL 코드 크기가 증가하는 경우에 대해서 완만한 증가 경향을 나타내고 있다.
- 실험에 이용된 VHDL 프로그램이 충분하지 못한 것은 상용 VHDL 컴파일러의 지원범위가 LRM의 subset으로써 제한적이고, 본 연구의 실험환경인 분석기, VHDL-to-C 사상기 및 시뮬레이션 커널에 대한 디버깅이 진행되고 있기 때문으로써 이에 대한 추가 실험을 위한 보완작업이 현재 진행 중에 있다.



a) 사상코드 크기 b) 코드 사상시간

그림 9. C 코드 크기 및 사상시간의 변화 경향
Fig. 9. Trends of C code size and timing overhead.

표 4. C 코드 및 VHDL 시뮬레이터의 비교
Table 4. Comparisons of C code and VHDL simulator in size.

Test Suites	VHDL code size(byte)	VHDL-to-C1)		QuickSim II(mentor)2)		vhdlsim (synopsys)3)	
		time 4) (sec)	C code 5) (byte)	time 4) (sec)	native code 5) (byte)	time 4) (sec)	pseudo code 5) (byte)
Nand	228	1.23	13430	0.76	1456	1.10	9172
Clock_Generator	605	1.34	12041	0.71	2640	1.11	10793
Moore_Machine	1389	1.49	22202	0.79	4104	1.12	18253
Traffic_Controller	2677	1.98	33640	0.89	5472	1.37	29024
Count_Zero_Sequencer	3029	1.67	25048	0.70	6440	1.19	19833
Add_Subtracter	3684	2.39	30430	0.86	13600	1.72	39018
Src54s151	4682	3.20	96675	0.92	16928	1.25	32829
Comparator	5343	3.02	53710	1.00	11560	1.82	56237
Differential_Equation	24673	8.32	85899	1.44	92456	3.39	162865
GCD_factorization	28484	5.26	106011	1.58	86656	3.85	234372

1) VHDL-to-C : Axil 320, Solaris 2.5 환경에서의 수행시간
2) QuickSim II : HP 715, HP-UX 9.05 환경에서의 수행시간
3) vhdlsim : Ultra 1, Solaris 2.5.1 환경에서의 수행시간
4) 시뮬레이션 코드 생성 시간
5) 시뮬레이션 커널을 제외한 시뮬레이션 코드 크기

V. 결 론

본 논문에서는 VHDL 컴파일러 개발을 위하여 VHDL 컴파일러 전반부가 분석한 중간 형식 데이터를 C 프로그램으로 사상하는 컴파일러 후반부의 VHDL-to-C 사상 과정을 설계 및 구현한 연구 결과에 대해서 기술하였다.

본 연구에서 설계된 템플릿 방식의 VHDL-to-C 사상 과정은 62,461 라인의 C 프로그램으로 구현되었으며, 디버깅이 진행되고 있는 컴파일러 전반부[9]와 VHDL 시뮬레이션 커널[14]과 함께 동작, 기능 및 성능에 대한 실험이 진행되었다. 실험을 통해서 VHDL-to-C 사상기는 컴파일러 전반부가 분석한 Validation Suite의 96% VHDL 구문에 대해서 완벽한 사상처리 능력을 보였다. 또한 생성된 코드의 크기에서는 해석기 방식에 비해서 작은 메모리 오버헤드를 요구하였으며, 직접코드 사상방식에 대해서는 VHDL 설계가 커짐에 따라서 사상의 효율성이 상대적으로 개선됨을 나타내었다. 그러나 사상 메카니즘의 처리시간에서는 상용 해석기 및 직접코드 사상방식의 VHDL 컴파일러와의 비교를 통하여 오버헤드가 존재함을 확인할 수 있었다. 이러한 문제점은 전반부에서의 라이브러리 파일 처리시간과 후반부에서의 AST트리 검색시간 및 C 템플릿 구성에 대한 개선의 필요성이 있음을 보였다.

VHDL 컴파일러의 일부분인 VHDL-to-C 사상 기능에 대해서 이루어진 본 연구는 컴파일러 전반부와 시뮬레이션 커널 등에 대한 상호연관성이 매우 밀접하게 진행되었다. 따라서 VHDL-to-C 사상기뿐만 아니라 VHDL 컴파일러 다른 부분과 전체에 대한 동작 및 기능, 성능 등에 관한 디버깅을 통해서 보완 작업이 계속될 것이다.

참 고 문 헌

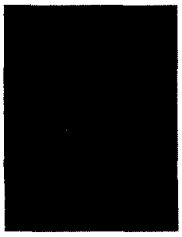
[1] M.C. Markowitz, "Simulating the function," EDN Asia, Apr 1993, pp.71-82.
[2] D.M. Lewis, "A Hierarchical Compiled Code Event-Driven Logic Simulator," *IEEE TCAD*, vol. 10, no. 6, June 1991, pp. 726-737.
[3] *VHDL System Simulation Workshop*, Synopsys Inc., 1995.

[4] *Vantage Spreadsheet User's Guide Volume 1,* Vantage Analysis Systems Inc., Dec 1990.
[5] *Getting Started with QuickSim II,* Mentor Graphics Corp., 1995.
[6] *Leapfrog VHDL Simulator Reference Manual 1.0,* Cadence, June 1993.
[7] 최기영, "IVDT: VHDL Tool 개발 환경," *Proceedings of the 1st Korea VHDL User's Group Workshop,* pp. 1-30, 1993년 5월
[8] 이영희, 김헌철, 황선영, "다층 레벨 VHDL 시뮬레이터 설계," 대한 전자공학회 논문지, 제 30-A권 제 10호, 1993년 10월, pp. 67-76
[9] 공진흥, 고형일, "VHDL-to-C 사상을 위한 VHDL 컴파일러 전반부의 설계," 한국통신학회 논문지, vol. 22 no. 12 , pp. 2834-2851, Dec. 1997
[10] 이영희, 황선영, "VHDL 시뮬레이터의 설계와 구현," *대한 전자공학회지,* vol. 22 no. 8, pp. 90-924, Aug. 1995
[11] *MyVHDL Station,* 서두로직, 1994
[12] *IEEE Standard VHDL Language Reference Manual,* IEEE Std. 1076-1987, IEEE, Mar. 1988.
[13] *ANSI/IEEE Standard VHDL Language Reference Manual,* IEEE Std. 1076-1993, IEEE, June 1994.
[14] 공진흥, "VHDL 컴파일러/시뮬레이터의 설계 및 구현," 반도체공동연구소 연구보고서, Sep. 1997
[15] *VHDL Validation Testsuite Users Manual,* VHDL Technology Group, 1995.

저 자 소 개

孔 鎭 興(正會員)

1976년 3월 ~ 1980년 2월 서울대학교 전자공학과 공학사. 1980년 3월 ~ 1982년 2월 한국과학기술원 통신공학과 공학석사. 1986년 9월 ~ 1989년 12월 텍사스주립대학교(Austin) 컴퓨터공학과 공학박사. 1979년 11월 ~ 1986년 8월 삼성전자(반도체 연구소) 과장. 1989년 9월 ~ 현재 광운대학교 컴퓨터공학과 부교수



高 亨 壹(正會員)

1991년 3월 ~ 1995년 2월 광운대학교 전자계산기공학과 공학사. 1995년 3월 ~ 1997년 2월 광운대학교 대학원 전자계산기공학과 공학석사. 1997년 3월 ~ 현재 광운대학교 대학원 컴퓨터공학과 박사과정