論文2001-38SD-3-8

마이크로프로세서를 위한 명령어 집합 시뮬레이터의 자동 생성

(Automatic Generation of Instruction Set Simulators for Microprocessors)

李 晟 旭*, 洪 晩 杓*

(Seong-Uck Lee and Man Pyo Hong)

요 약

새로운 마이크로프로세서의 설계, 최적화, 그리고 완성 후 어플리케이션의 작성 단계에서 칩의 명령어 집합 시뮬레이션은 필수적인 요소이다. 그러나, 기존의 시뮬레이션 툴들은 저 수준의 하드웨어 기술언어와 게이트 레벨 이하의 시뮬레이션으로 인해 시뮬레이터 구성과 실행 시에 상당한 시간적 지연을 초래하고 있다. 본 논문에서는 이러한 문제들을 해소하고 칩 제작과정에서 발생하는 잦은 설계 변경에 유연성 있게 대응할 수 있는 레지스터 전송 수준의 명령어 집합 시뮬레이터 생성기를 제안하며 그 설계 및 구현에 관해 기술한다.

Abstract

Simulation of an instruction set is essential to design and optimize new microprocessors, and to develop application programs. Though many simulation tools are widely used, their low-level description and simulation make users construct simulators difficult and spend a lot of time for simulation. We developed an automatic generator of instruction set simulators that perform register-transfer-level simulation. This automatic generator might be adaptable so as to be suitable for new modification or different conditions in designing microprocessors. In this paper, we describe a structure of automatic generation system and an implementation details.

I. 서 론

최근 컴퓨터 및 관련 기술의 빠른 발전으로 정보화, 자동화의 대상 영역이 확대됨에 따라 새로운 마이크로프로세서들이 지속적으로 출시되고 있으며, 특히 전자통신 장비와 이의 멀티미디어 응용이 보편화되면서 이들 장비의 핵심기술이라 할 수 있는 DSP 관련 제품에 대한 수요는 나날이 급증하고 있다[1~3]. 이러한 칩들은 소수의 업체가 표준을 주도하는 범용 CPU에 비해 제한된 응용 분야를 가지며 고속의 신호 처리를 목적으로 하므로, 속도 향상을 위한 다양한 아키텍처가 적용된 여러 제품군이 존재하며 국내 업체와 대학에서도 개발이 이루어지고 있다[4~5].

새로운 칩의 개발은 하드웨어의 설계, 최적화 그리고 제작 단계를 통하여 이루어진다. 칩의 시뮬레이션은 이러한 각 단계에서 설계된 칩의 동작을 검증하고 칩의 최적화를 위한 벤치마킹을 수행하는데 이용되며, 칩이 제작된 후 어플리케이션 프로그램의 작성에도 이용된다. 특히, 하드웨어 제작 과정 중 내부 기능 유닛을 디자인하고 동작을 검증하는 하드웨어 설계와 최적화 단계는 분석력과 창의력을 요구하는 단계이기 때문에 많은 시간과 노력을 필요로 한다. 이 과정에서는 사용 목

* 正會員, 亞州大學校 情報通信大學
(Collage of Information & communication, Ajou University)

※ 이 논문은 두뇌한국21 사업에 의하여 지원되었음
接受日字:1999年10月1日, 수정완료일:2001年2月6日

적과 비용, 성능 등 여러 환경 요소들을 고려해야 하므로 수정, 검증, 성능 평가 작업이 계속적으로 반복되며, 이 때 기능 유닛들의 동작 검증과 벤치마킹을 위해 시뮬레이션을 수행하게 된다.

일반적으로, 좁은 의미에서의 논리 시뮬레이션(logic simulation)은 게이트 레벨(gate-level)에서 이루어지는 회로 요소(circuit element)의 시뮬레이션을 뜻하며, 넓은 의미에서는 스위치 레벨(switch-level)에서부터 동작 레벨(behavioral-level)까지의 시뮬레이션을 말한다[6 8]. VHDL, Compass, Verilog HDL, Synapsis 등의 EDA(Electronic Design Automation) 툴에는 VHDL, Verilog HDL과 같은 하드웨어 기술 언어를 입력으로 하는 시뮬레이터가 포함되어 있으며, 다소간의 차이는 있으나 대부분 좁은 의미에서의 논리 시뮬레이션과 넓은 의미에서의 논리 시뮬레이션을 대부분 만족하는 시뮬레이션 기능을 제공한다[3,6]. 그러나 이들 작업에 기존의 시뮬레이션 툴을 이용할 경우 다음과 같은 문제들에 직면한다.

먼저, 프로그래밍의 어려움이 있다. 대부분의 시뮬레이션 툴에서 제공하는 하드웨어 기술 언어(Hardware Description Language; HDL)가 낮은 레벨의 언어이고, 논리합성 가능한 함수만을 제공하기 때문에 기능 유닛들의 동작 모델(behavior model)을 구현하는데 많은 제약을 받는다.

다음으로는, 시뮬레이션 시간의 지연을 들 수 있다. 하드웨어를 최적화 하는 단계에서는 벤치마킹을 통해 수행 클럭 사이클이나 명령어의 사용빈도 등 여러 측정 요소들의 결과를 얻어 이들을 분석하는 것이 목적인데, 기존 시뮬레이션 툴들은 함수나 모듈들이 게이트 레벨 이하에서 게이트 딜레이와 와이어 딜레이 등의 요소를 감안하여 동작하기 때문에 결과를 얻는데 상당한 시간적 지연을 초래하게 된다.

따라서, 본 논문에서는 기존의 시뮬레이션 툴이 하드웨어 설계와 최적화 과정에서 가지고 있는 문제들을 해소하고, 칩 제작과정에서 발생하는 잦은 설계 변경에 유연성 있게 대응할 수 있는 레지스터 전송 수준(Register Transfer Level; RTL)의 명령어 집합 시뮬레이터 생성기 구현에 관해 기술한다.

본 논문의 구성은 다음과 같다. 2장에서는 본 논문에서 제안하는 시뮬레이터 생성기의 기본 개념을 살펴보고, 3장에서는 시뮬레이터 생성기의 설계 및 구현 방식을 기술한다. 4장에서는 구현된 시뮬레이터 생성기를 이용하여 다양한 아키텍처에 대한 시뮬레이터를 구성하는 방식에 대해 설명하고, 5장에서는 결론 및 향후의 연구방향을 제시한다.

Ⅱ. 시스템 개요

상술한 바와 같이 기존의 시뮬레이터를 이용한 시뮬레이션 작업은 시뮬레이터 구성과 실행 시에 상당한 시간을 요구하므로, 하드웨어 설계 및 최적화 단계에서의 동작 모델 시뮬레이션에 시간적 지연을 초래하며, 어플리케이션 개발을 위한 시뮬레이터로서는 부적합하다는 문제를 안고 있다.

이에 반해 본 논문에서 제안하는 시뮬레이터 생성기는 동작 모델이 레지스터 전송 수준으로 기술된 스크립트를 입력받아 C++로 작성된 명령어 집합 시뮬레이터의 소스코드를 자동으로 작성하고 컴파일 하여 시뮬레이터를 생성하므로, 상술된 단점들을 극복하는 유연한 시뮬레이션 플랫폼을 제공한다. 이러한 시뮬레이터 생성기의 개괄적인 구조는 그림 1과 같다.

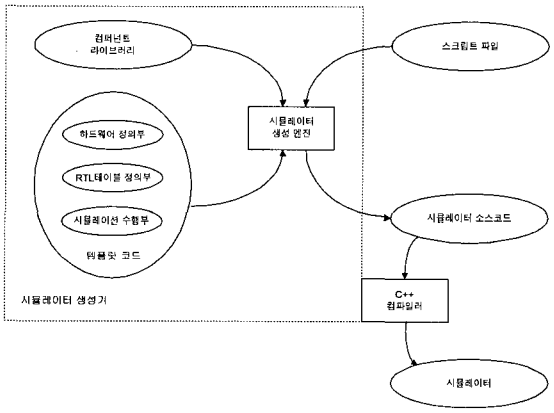


그림 1. 시뮬레이터 생성기의 구조 및 동작
Fig. 1. Architecture and behavior of the proposed simulator generator.

제안하는 시뮬레이터 생성기는 시뮬레이터의 자동생성을 위해 사전에 작성되어 있는 C++ 소스 코드를 이용하는데, 이것이 컴퍼넌트 라이브러리(component library)와 템플릿 코드(template code)이다. 컴퍼넌트 라이브러리는 각종 하드웨어 컴퍼넌트들을 C++ 클래스들로 구현하여 라이브러리화 한 것으로서, 레지스터, 메모리, 제어 신호 비트, 버스 등의 보편적인 컴퍼넌트들은 변경 없이 시뮬레이터에 포함되며, 하드웨어마다 다

른 구조를 가지는 컴퍼넌트들은 내장된(built-in) 컴퍼넌트 클래스들의 상속 또는 포함을 통해 대상 하드웨어의 기능 유닛(functional unit)을 구성하게 된다. 템플릿 코드는 실제 시뮬레이션을 진행시키는 시뮬레이션 수행부의 소스 코드로, 대부분이 이미 코딩되어 있으나 그 일부는 확정되지 않고 특별한 표식(mark)을 가지고 있다. 즉, 템플릿 코드 중 하드웨어 구성 요소나 동작 등에 따라 가변적인 부분은 공란으로 비워져 있으며, 이러한 공란에는 이 부분이 스크립트 파일의 해석에 의해 추후에 채워질 부분임을 표시하는 표식이 붙여지는 것이다. 따라서, 시뮬레이터 생성기는 입력으로 주어진 스크립트 파일에서 이들 가변적인 요소에 대한 내용을 읽어들여 템플릿 코드의 비어있는 부분을 채워 넣음으로써 시뮬레이터를 완성한다.

Ⅲ. 시뮬레이터 생성기

1. 입력 스크립트

스크립트 파일에는 대상 하드웨어의 동작 레벨 모델인 가상기계의 구조에 관한 기술과, 명령어의 동작을 묘사한 RTL 기술이 담겨있다. 시뮬레이터 생성기의 스크립트 해석기는 이를 읽어들여 가상기계의 생성과 동작 수행을 위한 C++ 소스 코드를 만들어 낸다. 그림 2는 스크립트 파일의 구조를 확장 BNF(Backus-Naur Form)로 기술한 것이며, 그림 3은 스크립트 파일의 예를 제시한 것이다.

제시된 바와 같이 스크립트 파일은 5개의 섹션(Section)으로 구성되어 있다. User Defined Component 섹션은 대상 하드웨어에만 존재하는 특수한 하드웨어 컴퍼넌트를 정의하는 부분으로, 특수한 연산 유닛 등을 구현할 때 이용하며 C++의 클래스 정의 문법(cpp_class_definition)에 따라 기술된다. 대부분의 컴퍼넌트는 내장된 기본적인 컴퍼넌트들의 상속 또는 조합으로 구성 가능하므로, C++ 구문에 익숙하지 않은 사용자도 손쉽게 작성할 수 있다.

Machine 섹션은 대상 하드웨어를 구성하는 컴퍼넌트들을 기술하는 부분으로, 타이머와 파이프라인의 기술, 그리고 기타 컴퍼넌트 정의가 이루어진다. 이 섹션의 첫 행에는 타이머의 명칭과 함께 상위 타이머와 최대값이 기술되며, 특별히 글로벌 클럭(Global Clock)과 함께 1 씩 증가하는 타이머는 상위 타이머 대신 Sync-Clock이 지정된다. 즉, 예제의 타이머 T는 글로벌 클럭

```
hardware_description_script ::= user_defined_component_section opt
                                machine_section
                                instruction_description_section opt
                                user_defined_function_section opt
                                rtl_description_section

user_defined_component_section ::= %User Defined Component { cpp_class_definition }

machine_section ::= %Machine {
                        timer_definition pipeline_description { component_definition }0+
                    }
timer_definition ::= Timer : timer_name ( super_timer_name , max_value )
                           { , timer_name ( super_timer_name , max_value ) }0+
timer_name ::= identifier
super_timer_name ::= identifier
max_value ::= number
pipeline_description ::= Pipeline : pipeline_depth { { { component_definition }1+ } }opt
pipeline_depth ::= number
component_definition ::= class_name : component_name ( { parameter_value }0+ )
                                    { , component_name ( { parameter_value }0+ ) }0+
parameter_value ::= number

instruction_description_section ::= %Instruction Description { { instruction_definition }1+
}
instruction_definition ::= instruction_no : { 0 | 1 | x }1+
instruction_no ::= number

user_defined_function_section ::= %User Defined Function { cpp_function_definition }

rtl_description_section ::= %RTL Description { { rtl_description }1+ }
rtl_description ::= issue_condition : cpp_expression
issue_condition ::= component_name { ( parameter_value ) }opt
                  { , component_name { ( parameter_value ) }opt }0+
```

그림 2. 스크립트 형식
Fig. 2. Grammar of the script.

과 동기되는 8비트 로컬 타이머(Local Timer)의 역할을 수행하며, 타이머 t는 상위 타이머 T가 1 증가하는 동안 4번 증가하게 된다. 이러한 계층 관계를 가지는 타이머 집합은 하나의 클럭 내에서 정확한 타이밍을 유지하며 수행되어야 하는 동작의 기술에 이용될 수 있다.

다음 행은 파이프라인의 깊이를 정의하며, 뒤따르는 "{}" 내에 기술된 컴퍼넌트들은 파이프라인 되는 제어 신호를 정의한다. 따라서, 이 컴퍼넌트들은 파이프라인 깊이와 같은 수의 인스턴스(instance)를 가지게 되며 그 상태 값이 파이프라인의 진행에 따라 자동적으로 이동된다. 이는 파이프라인을 따라 전송되는 제어 신호를 모사하기 위한 것이다. 즉, 예제의 "Pipeline : 8 { D(89) }"는 이 하드웨어의 파이프라인 깊이가 8 이고, 89 비트로 구성된 배열 D가 파이프라인 되는 제어 신호임을 의미한다.

Machine 섹션의 나머지 행들은 일반적인 하드웨어 컴퍼넌트들을 정의한다. 그림 2, 3에서와 같이 각 행은 컴퍼넌트의 타입과 정의할 컴퍼넌트 이름으로 구성되며, 컴퍼넌트의 타입은 내장된 기본 컴퍼넌트 타입과 User Defined Component 섹션에서 정의된 사용자 정의 컴퍼넌트를 자유로이 이용할 수 있다. 컴퍼넌트 이름 뒤의 괄호에 기술된 숫자는 해당 컴퍼넌트의 크기를 의미한다. 예를 들어 그림 3에서 "Register : R0(32), R1(32)"는 32 비트 레지스터 R0와 R1를, 그리고 "Memory : Pmem(32, 4096)"은 32 비트 단위의 4kb 메모리 Pmem을 정의한다.

다음의 Instruction Description 섹션은 선택적인 항목으로, 사용자가 기정의된(pre-defined) 디코딩 함수를 이용할 때 기술한다. 이는 명령어 디코딩 시 페치(fetch)된 명령어의 비트 패턴을 분석하여 해당 명령어를 구분해 내는 작업이 필요하므로, 이 과정을 하나의 Decode() 함수로 수행할 수 있도록 한 것이다. 예를 들어, 그림 3의 Decode(D, IR)은 레지스터 IR의 내용을 읽어 디코딩한 후 그 결과를 제어 신호 비트 배열 D에 기록함을 나타낸다. 따라서, 이 섹션에서는 0, 1, 또는 x로 나타낸 명령어 비트 형태들과, 페치된 명령어가 해당 명령어 비트 형태와 일치할 때 이를 기록할 제어 신호 비트 번호를 기술하여 Decode() 함수가 명령어 디코딩 시 이용할 수 있도록 한다. 명령어 비트 형태 중 x는 돈 캐어(don't care) 비트를 의미한다.

그 뒤의 두 섹션은 모두 RTL 기술에 관련된 부분이며, User Defined Function 섹션은 명령어 동작을 기록할 때 빈번히 사용되는 루틴들을 함수로 정의하는 부분이다. 이 섹션은 C++의 함수 정의 형태(cpp_function_definition)로 기술되며, RTL Description 섹션의 동작 기술을 간략하게 하기 위한 매크로 정의 섹션의 역할을 수행한다.

RTL Description 섹션은 실제 명령어의 동작을 기술하는 부분으로, 각 기능 유닛이 수행하는 단위 동작들이 레지스터 전송 수준으로 기술되며, 각각의 단위 동작에는 해당 동작을 실행할 조건이 되는 제어 비트들

```
%User Defined Component {
   class Vadder : public adder {
      ...
   }
}

%Machine {
    Timer : T(SyncClock,8), t(T,4)
    Pipeline : 8 {
        Control : D(89)
    }
    Register : R0(32), R1(32), R2(32), R3(32)
    Register : AR0(24), AR1(24), AR2(24), AR3(24)
    Register : IR(32), PC(24)
    Memory : Pmem(32, 4096), Dmem(32, 4096)
    Control : L(1), F(1), ...;
    ...
}
```

```
%Instruction Description {
    0 : 01x0 0000 xxxx xxxx xxxx xxxx xxxx
    1 : 01x0 0001 xxxx xxxx xxxx xxxx xxxx
    ...
}

%User Defined Function {
    void uf1(void) {
       ...
    }
}

%RTL Description {
    FT, T(0) : IR = Pmem[PC];
               PC = PC + 1;
    T(1) : Decode(D, IR);
    D(24), T(3) : Dmem[IR(23,16)] = 0;
    ...
}
```

그림 3. 스크립트 파일 예제
Fig. 3. An Example of the script.

이 선행된다. 예를 들어, 그림 3에서 첫 번째 동작 기술은 FT 신호와 T0가 셋트(1)되었을 때, PC가 가리키는 메모리 번지로부터 IR로 값을 로드하고 PC값을 증가시키는 페치 단계를 기술한 것이다. 동작 기술은 C++ 수식 형태(cpp_expression)를 따르며, 빈번하게 사용되는 연산 또는 동작은 표 1에 제시된 간략한 표현을 이용할 수 있다. 이러한 구성으로 인해 스크립트에 사용되는 C++ 문장은 극히 기본적인 것들로 제한되므로, C++ 언어에 대해 지식이 없는 사용자도 손쉽게 스크립트를 작성할 수 있다.

표 1. 추가된 RTL 표현
Table 1. Additional RTL expressions.

동작 / 표현	의미
Mem[addr]	Mem으로 지정된 메모리의 addr 번지
Reg1 + Reg2	레지스터 Reg1 과 Reg2의 값을 더함
Mem[addr] = Reg	addr 번지에 Reg 값을 대입
Reg(Start,End)	레지스터 Reg의 Start 비트와 End 비트 사이의 값

2. 컴퍼넌트 라이브러리

시뮬레이터 생성기가 이용하는 컴퍼넌트 라이브러리는 하드웨어의 각 컴퍼넌트가 궁극적으로는 데이터 스토리지(data storage)로 모델링 가능하다는 관점에서 작성된 것이다. 이러한 관점에서 각각의 하드웨어 컴퍼넌트는 그 상태(state)를 저장하는 데이터 멤버와 이를 입출력/조작하는 메소드로 묘사되며, 하나의 하드웨어는 이러한 컴퍼넌트들의 집합으로 간주될 수 있다. 따라서, 제공되는 컴퍼넌트 라이브러리는 그림 4과 같은 계층을 가진다.

Storage 클래스는 모든 컴퍼넌트들의 기반 클래스이며, 데이터 저장 공간과 이 값을 읽고 쓸 수 있는 메소드들로 구성되어 있다. 여기에서 파생된 Memory 클래스는 메모리와 같이 일정 크기의 단위로만 데이터를 억세스 하는 컴퍼넌트를 표현하고, BitStorage 클래스는 비트 수준의 억세스가 필요한 컴퍼넌트를 표현한다. 따라서, 레지스터, 제어 신호, 버스, 스위칭 네트웍, 연산기 등의 클래스들이 BitStorage 클래스에서 파생되는 구조를 취하고 있다.

한편, 컴퍼넌트 라이브러리에 속한 클래스들은 기본적인 단위 조작 메소드와, 이들을 조합한 상위 수준의 조작 메소드를 모두 제공한다. 따라서, 새로운 하드웨어 컴퍼넌트가 필요한 경우, 유사한 기존 클래스가 제공하는 단위 조작의 결합을 통해 새로운 상위 조작 메소드를 정의하는 것만으로 대부분의 컴퍼넌트를 작성할 수 있게 된다.

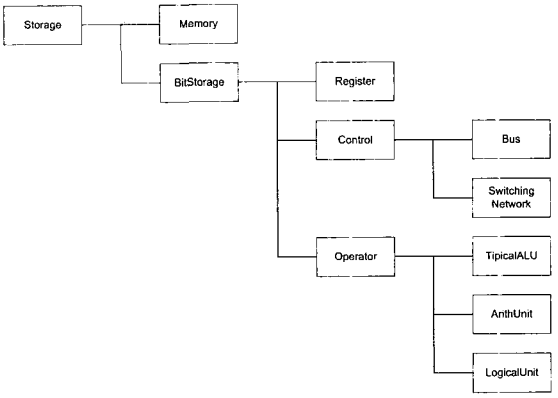


그림 4. 컴퍼넌트 클래스 계층
Fig. 4. Hierarchy of component classes.

3. 템플릿 코드

1) 하드웨어 정의부

하드웨어 정의부는 하드웨어를 구성하는 컴퍼넌트들을 가상기계에 속한 객체로 선언하는 부분이며, 스크립트의 Machine 섹션을 분석하여 생성된다. 그림 5는 사용자가 작성한 입력 스크립트와 이의 해석을 통해 생성된 하드웨어 정의부의 C++ 소스 코드를 도시한 것으로, 생성 과정이 기계적이고 단순하게 이루어짐을 보여준다. 즉, 하드웨어 정의부의 각 행은 일반적으로 "컴퍼넌트 유형 : 정의될 컴퍼넌트 이름 리스트"의 형식으로 기술되는데, 컴퍼넌트 유형은 이에 대응되는 C++ 클래스 명으로, 각각의 컴퍼넌트는 해당 클래스 인스턴스로 변환된다. 또한, 이렇게 정의된 컴퍼넌트들은 이용이 용이하도록 Machine 클래스의 멤버로서 정의된다.

다만, 그림 5에서 타이머 T는 별도의 명시가 없어도 파이프라인 깊이와 같은 수의 인스턴스를 가지는데, 이는 타이머 T가 각 파이프라인 스테이지를 위한 로컬 타이머로 지시되었기 때문이다. 여기에서 로컬 타이머라 함은 하나의 명령이 페치된 후 경과된 시간을 담고 있는 제어 신호 비트로서, 매 실행 스텝마다 글로벌 타이머(global timer)에 의해 자동적으로 갱신되며 파이프라인을 따라 이동한다.

하드웨어 정의부에는 사용자에 의해 정의된 새로운 하드웨어 컴퍼넌트가 이용될 수 있으며, 새로운 컴퍼넌트의 정의를 담고 있는 User Defined Component 섹션은 C++ 형태로 기술되므로 별도의 처리 없이 하드웨어 정의부 선두에 그대로 삽입된다.

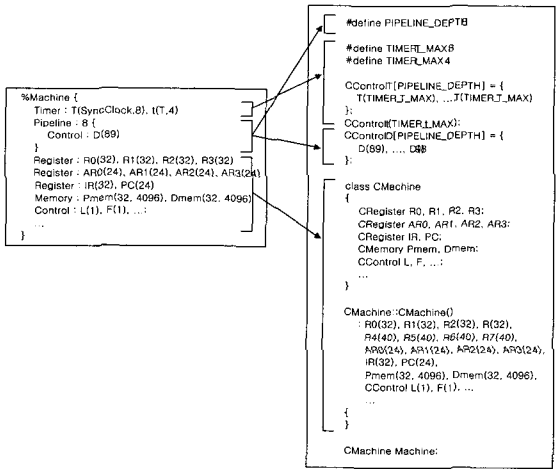


그림 5. 스크립트와 생성된 하드웨어 정의부 소스 코드

Fig. 5. mapping a script to C++ source code for hardware definition.

2) RTL 테이블 정의부

RTL 테이블은 각 동작의 실행조건 리스트와, 이 조건을 만족할 때 수행할 함수의 포인터들로 구성된다. 실행조건 리스트는 0, 1, x로 구성된 비트 스트링의 형태로 표현되며, RTL Description 섹션의 동작 기술 중 콜론 좌측의 조건들을 떼어내서 얻을 수 있다. 이를 위하여 시뮬레이터 생성기는 RTL 테이블 정의부 전체에서 동작 조건에 나열된 모든 제어 비트들의 리스트를 작성하고, 이것을 하나로 연결한 비트 스트링의 형태로 표현한다. 예를 들어 그림 6에 나타난 RTL만으로 표현할 수 있는 기계가 있다면, 이 기계는 단지 FT, T, D만을 RTL 동작 조건으로 이용하는 것이다. 따라서 이러한 경우 RTL 테이블의 실행조건 리스트는 전체 길이가 98(= 1 + 8 + 89)인 비트 스트링으로 나타나게 되고, 0번째 비트가 FT, 1-8번째 비트가 T, 나머지 비트가 D에 대응된다. 따라서, 구조체 SRTLtbl의 cond 필드의 길이인 STATE_BLOCK_SIZE는 이 경우에 98로 정의될 것이다.

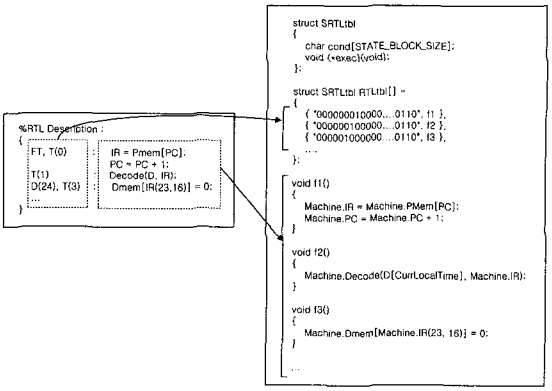


그림 6. 스크립트와 생성된 RTL 테이블 정의부 소스 코드

Fig. 6. mapping a script to C++ source code for RTL table definition.

한편, 스크립트의 RTL 동작들은 단순한 몇 가지 규칙에 의해 C++ 소스 코드로 변형되어 각각 하나의 함수로 생성되므로, 그 함수 포인터만을 기록함으로써 RTL 테이블을 완성할 수 있다. 스크립트에서 사용하는 대부분의 표현은 C++ 언어의 형식과 유사하며, 표 1에 제시된 표현은 컴퍼넌트 작성 시에 연산자 중첩(operator overloading)을 이용하여 구현되므로 문자열 처리 수준의 단순한 변환만으로 C++ 소스 코드를 생성할 수 있다. 즉, RTL 표현 중에서 하드웨어 컴퍼넌트의 이름을 발견하면, Machine 오브젝트의 멤버로 표시하기 위해 그 앞에 "Machine."을 삽입하고, 그림 6의 D와 같이 파이프라인 처리가 수반되는 제어 비트에는 해당 파이프라인 스테이지를 정확하게 첨부해주는 정도 간단한 처리로 C++ 소스코드를 생성할 수 있다. RTL 매크로를 정의하는 User Defined Function 섹션은 User Defined Component 섹션과 같이 그대로 RTL 테이블 정의부 소스 코드에 삽입된다.

3) 시뮬레이션 수행부

이상과 같이 하드웨어 정의부와 RTL 테이블 정의부가 분리된 구조를 채택함으로써, 시뮬레이션 수행부는 하드웨어에 관계없이 동일한 코드를 가질 수 있게 된다. 실제로 시뮬레이션 수행부의 소스 코드는 다수의 계층적인 타이머가 사용될 경우, 이에 따라 타이머를 증가시키는 루프의 중첩이 깊어지는 점 외에는 스크립트의 영향을 받지 않고 고정되어 있다.

시뮬레이션 수행부의 코드는 글로벌 타이머의 한 클럭을 의미하는 스텝(step) 단위로 시뮬레이션을 수행하

도록 작성되었으며, 한 스텝의 시뮬레이션 과정은 그림 7과 같이 이루어진다. 도시된 바와 같이 생성된 시뮬레이터는 타이머를 한 클럭 증가시키고 제어신호의 파이프라인 전송을 수행한 뒤, 타이머와 각 제어신호 비트들의 상태를 파악하여 하드웨어 상태 리스트를 구성한다. 이렇게 구성된 하드웨어 상태 리스트는 각각의 RTL 동작 실행조건과 비교되고, 이것이 일치되면 이에 대응하는 함수가 수행된다. 이 과정은 한 스텝 내에서 파이프라인의 깊이만큼 반복되며, 각각의 반복에서는 파이프라인 되는 제어 신호의 각기 다른 인스턴스가 이용된다. 즉, 생성된 시뮬레이터는 파이프라인 되는 각각의 제어 신호 인스턴스와, 타이머, 제어 신호들의 상태를 결합하여 파이프라인 깊이와 같은 수의 하드웨어 상태 리스트를 얻어 이들 조건을 만족하는 RTL 동작을 실행한다.

이와 같은 방법으로 RTL 동작들이 실행되면, 그 동작에 따라 가상기계의 메모리, 레지스터, 하드웨어 스택과 같은 컴퍼넌트들의 상태가 변경되며 이러한 상태 값 변화는 다음 스텝에서 반영된다.

4. 시뮬레이터 생성 엔진

시뮬레이터 생성 엔진은 제안하는 시뮬레이터 생성기의 커널로서, 사용자가 작성한 입력 스크립트를 해석하고 템플릿 코드의 가변적인 부분을 완성하여 시뮬레이터의 소스 코드를 생성한다. 또한, 생성된 코드에 대한 컴파일도 자동적으로 수행하므로, 사용자는 스크립트를 작성하고 이를 입력으로 하여 시뮬레이터 생성기를 구동하는 것만으로 완성된 시뮬레이터의 실행 파일을 얻어 낼 수 있게 된다. 상술한 바와 같이 스크립트 해석을 통해 정보를 추출하면 시뮬레이터 소스 코드의 완성은 문자열 조작 수준의 단순한 처리로 가능하며, 스크립트 파일, 템플릿 코드의 각 부분, 컴퍼넌트 라이브러리들과 시뮬레이터 생성 엔진의 관계를 도시하면 그림 8과 같다.

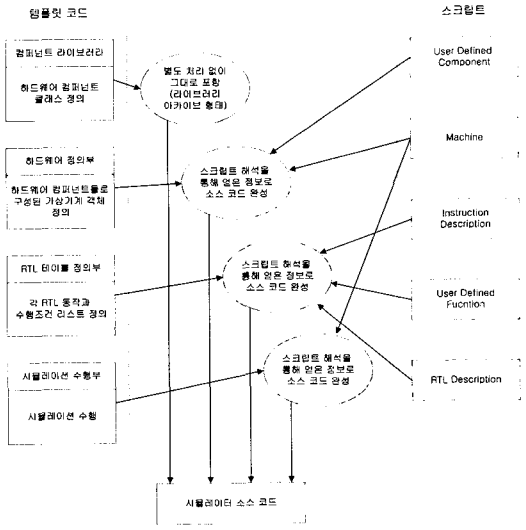


그림 8. 시뮬레이터 생성 방법
Fig. 8. Generation of a simulator.

Ⅳ. 생성기를 이용한 시뮬레이터 구현

본 논문에서 제안하는 시뮬레이터 생성기는 Sun SPARCstation-20과 Solaris 2.6 상에서 gcc 2.7.2.1을 이용하여 개발되었다. 시뮬레이터 생성기는 시뮬레이터 소스 코드 생성 직후 자동적으로 gcc를 호출하여 시뮬

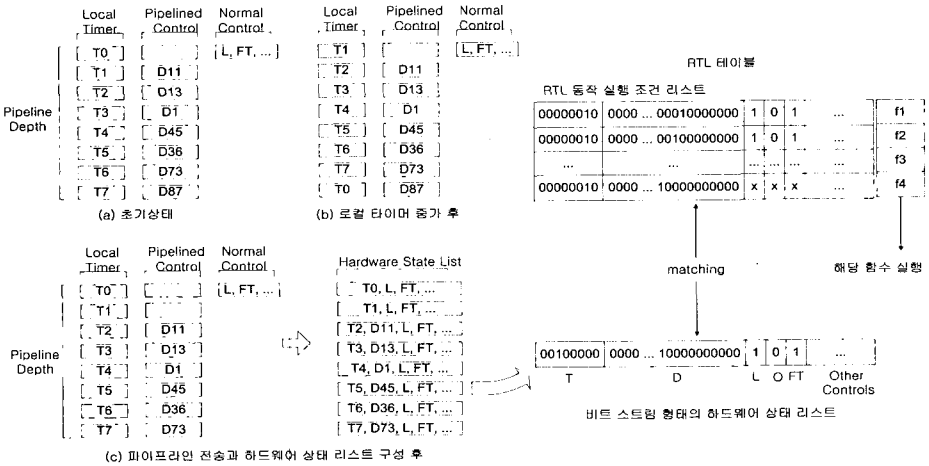


그림 7. 생성된 시뮬레이터의 동작
Fig. 7. Behavior of a generated simulator.

레이터 실행 파일을 생성하므로, 시스템에 gcc가 설치되어야 정상적으로 동작한다. 그러나, 시뮬레이터 생성기와 생성되는 시뮬레이터의 소스 코드는 운영체제와 컴파일러에 관계없이 동작하도록 작성되었으므로, 극히 적은 수정만으로 다른 플랫폼에 탑재 가능하다.

상술한 바와 같이 시뮬레이터 생성기는 대상 칩의 동작 모델을 스크립트 형식에 맞추어 기술하는 것만으로 새로운 시뮬레이터를 구현 할 수 있도록 한다. 동작 모델의 정의는 크게 하드웨어 컴퍼넌트 정의와 이들의 RTL 동작 기술로 구성되며, 스크립트에는 5 개의 섹션으로 나누어 기술된다. 그림 3은 전형적인 파이프라인 아키텍처를 위한 스크립트의 예로 파이프라인의 깊이와 로컬 타이머의 크기가 8로 정의되어 있으며, 디코딩 신호 D가 파이프라인을 따라 전송되는 제어 신호로 지정되어 있다. 실제 시뮬레이션 수행 중에 각 스테이지의 로컬 타이머는 변하지 않고 고유한 값으로 고정되는데, 이는 로컬 타이머가 글로벌 클럭과 함께 증가된 후 자동으로 파이프라인을 따라 이동하기 때문이다. 즉, 첫 번째 스테이지부터 순서대로 T(0), T(1), T(2)와 같은 고유의 로컬 타이머 값을 가지게 되므로, 해당 스테이지의 타이머 값을 RTL 동작 수행 조건에 넣음으로써, 각각의 동작이 수행되는 스테이지를 정확히 지시할 수 있게 된다. 또한, 다수의 하드웨어 컴퍼넌트가 수행하는 동작간에 정확한 타이밍이 요구되는 경우에는 보다 낮은 단위의 타이머를 이용한다. 그림 3의 타이머 t는 하나의 클럭을 4분하여 동작하는 가상 타이머이다. 따라서, 한 스테이지의 실행 결과가 다른 스테이지의 동작에 영향을 미치거나 인터럽트의 감지와 같이 하나의 클럭 내에서 우선 수행되어야 하는 동작이 있는 경우에는, 해당 동작의 수행 조건에 클럭 내에서의 수행 타이밍 t(0)-t(3)를 명기하여 각 동작들이 정확한 타이밍에 수행되도록 한다.

한편, 파이프라인 구조를 채택하지 않은 하드웨어는 파이프라인의 깊이를 0으로 하고 단지 로컬 타이머의 크기만을 지정하여 시뮬레이터를 구현할 수 있다. 예를 들어, 하나의 명령어 수행에 4 클럭이 소요되는 하드웨어를 시뮬레이션 한다면, 로컬 타이머 T(4)를 선언하고 T(0)-T(3)에 수행될 동작을 적절히 기술함으로써 시뮬레이터를 구성한다. 모든 타이머 컴퍼넌트의 값은 적절한 시점에서 자동적으로 증가되며, 최대 값에 이르면 최소 값으로 순환하므로 타이머 값의 조절은 RTL 동작에 별도로 기술할 필요가 없다.

최근 프로그래머블(Programmable) DSP 칩에 많이 도입되고 있는 VLIW(Very Long Instruction Word) 또는 Superscaler 아키텍처의 경우에도 하나의 디코딩 스테이지가 다수의 디코더와 디코딩 신호로 구성된다는 점 외에는 일반적인 파이프라인 아키텍처와 동일하게 구현 가능하다. 따라서, 하나의 디코딩 스테이지 내에서 제공되는 디코딩 함수를 연속적으로 호출하거나, 독자적인 디코더 컴퍼넌트를 정의하고 여러 개의 인스턴스를 생성하여 이를 구현한다. 다만, VLIW 아키텍처에서 기정의된 디코딩 함수를 이용할 경우, 하나의 명령을 구성하는 각각의 단위 명령마다 가상의 명령어 레지스터(instruction register)를 정의하고 하나의 명령을 이들 각각에 분할하여 대입하는 과정이 추가되는 점이 다르다. 이러한 하드웨어 구성이 이루어지면, 각각의 가상 명령어 레지스터와 대응하는 디코딩 신호를 파라미터로 하여 디코딩 함수를 연속적으로 호출함으로써 다수의 디코더를 시뮬레이션 할 수 있다.

V. 결론 및 향후 연구

동작 수준의 칩 시뮬레이션은 새로운 마이크로프로세서의 설계 및 최적화 단계와, 하드웨어 완성 후 어플리케이션 개발에 필수적인 요소이다. 그러나, 기존의 시뮬레이션 툴들은 시뮬레이터 구성과 시뮬레이션 작업에 상당한 시간이 소요되고, 어플리케이션 개발자를 위한 명령어 집합 시뮬레이터로는 이용되기 어렵다는 문제를 안고 있다. 따라서, 본 논문에서는 칩 제작과정에서 발생하는 잦은 설계 변경에 신속하고 유연성 있게 대응할 수 있으며, 빠른 동작 모델 시뮬레이션을 가능하게 하는 레지스터 전송 수준의 명령어 집합 시뮬레이터 생성기를 제안하고 그 설계 및 구현에 관해 기술하였다.

제안된 시뮬레이터 생성기는 사용자에 의해 작성된 스크립트를 해석하여 시뮬레이터의 소스 코드를 생성하는 구조를 취하고 있으나, 궁극적으로는 GUI 기반의 시뮬레이터 생성기 환경을 구축함으로써 스크립트를 이용한 하드웨어 표현에서 탈피하고 직관적, 시각적인 방법을 통해 시뮬레이터 생성이 가능하도록 하는 접근 방법을 모색해 나갈 것이다.

참 고 문 헌

[1] 조위덕, "디지털 이동통신 신호처리 프로세서 설계," *Telecommunications Review*, 제6권 제1호, pp1-7, 1996년 1-2월

[2] 선우명훈, "디지털 이동통신용 DSP 칩 기술", *Telecommunications Review*, 제6권 제1호, pp1-5, 1996년 1-2월

[3] Jonathan Allen, "Computer Architecture for Digital Signal Processing," *Proceedings of IEEE*, Vol. 73, No. 5, pp.852-873, May 1985

[4] Motorola, *DSP 56100 Digital Signal Processor Family Manual*, Motorola Inc.,1993.

[5] Texas Instruments, *TMS320C6000 CPU and Instruction Set Reference Guide*, Texas Instruments, 1999.

[6] Mary L. Bailey, Jack V. Briner, JR., Roger D. Chamberlain, "Parallel Logic Simulation of VLSI Systems" *ACM Computing Surveys*, Vol. 26, No. 3, September 1994.

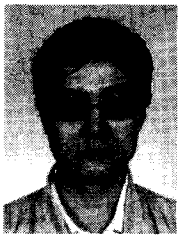
[7] Aposporidis. E. & Lohnert. F. "Parallel multi-level VLSI Simulator an Object-oriented approach.", *European Simulation Multiconference*, June 1989.

[8] Markus Levy, "Hands-On Evaluation: μP simulators", *EDN Asia Korean Edition*, pp. 20-35, May 1998.

저 자 소 개

李 晟 旭(正會員)

1994년 아주대학교 컴퓨터공학과 학사. 1996년 아주대학교 교통공학과 석사. 1997년~현재 아주대학교 컴퓨터공학과 박사과정. 관심 분야는 병렬처리



洪 晩 杓(正會員)

1981년 서울대학교 자연과학대학 계산통계학과 학사. 1983년 서울대학교 자연과학대학 계산통계학과 석사. 1991년 서울대학교 자연과학대학 계산통계학과 박사. 1983년 ~ 1985년 울산공과대학 전자계산학과 전임강사. 1985년 ~ 현재 아주대학교 정보 및 컴퓨터공학부 교수. 1993년 ~ 1994년 미네소타대학 전자공학과 교환교수. 관심분야는 병렬처리