SLEDS: 비동기 마이크로프로세서를 위한 상위 수준 사건구동식 시뮬레이터

(SLEDS: A System-Level Event-Driven Simulator for Asynchronous Microprocessors)

최 상 익 † 이 정 근 †† 김 의 석 ††† 이 동 익 ††††
(Sangik Choi) (Jeong-Gun Lee) (Euiseok Kim) (Dong-Ik Lee)

요 약 VHDL이나 Verilog와 같은 기존의 하드웨어 기술 언어(Hardware Description Language)를 이용하여 비동기 마이크로프로세서를 모델링하고 시뮬레이션을 수행할 수 있으나, 핸드셰이크 프로토콜(handshake protocol)에 의해 동작하는 비동기 마이크로프로세서의 기술이 지나치게 복잡해진다. 결과적으로, 성능 평가 시간이 너무 길어져 상위 수준(system level)에서의 효과적인 설계 공간 탐색에 많은 어려움을 겪는다. 따라서, 상위 수준에서 비동기적 특성인 핸드셰이크 프로토콜을 쉽게 모델링하고, 빠른 시간 내에 효과적으로 시뮬레이션할 수 있는 방법론과 도구가 필요하다. 이런 목적 하에, 프로세서 모델링과 시뮬레이션을 통하여 성능 평가를 수행할 수 있는 자동화 도구 *SLEDS*(System-Level Event-Driven Simulator)를 개발하였다. 본 도구의 궁극적 목표는 프로세서를 구성하는 모듈들의 지연을 조절하여(delay balancing) 전체적으로 프로세서가 고성능을 얻을 수 있도록 최적화 조건을 구하는 것이다. 이와 더불어, 정의된 행위를 실제로 수행함으로써, 예상한 결과와 실제 결과를 비교하여, 설계가 제대로 되었는지 상위 수준에서의 검증을 목표로 한다.

키워드 : 비동기 마이크로프로세서, 성능평가, 산건구동식 시뮬레이션, 상위수준 시뮬레이션

Abstract It is possible but not efficient to model and simulate asynchronous microprocessors with the existing HDLs(Hardware Description Languages) such as VHDL or Verilog. The reason is that the description becomes too complex, and also the simulation time becomes too long to explore the design space. Therefore, it is necessary to establish a methodology and develop a tool for modeling the handshake protocol of asynchronous microprocessors very easily and simulating it very fast. Under this objective, an efficient CAD(Computer-Aided Design) tool, *SLEDS*(System-Level Event-Driven Simulator), was developed which can evaluate performance of a processor through modeling with a simple description and simulating with event-driven engine in the system level. The ultimate goal in the tool *SLEDS* is to find the optimal conditions for a system to produce high performance by balancing the delay of each module in the system. Besides, *SLEDS* aims at verifying the design through comparing the expected results with the actual ones by performing the defined behavior.

Key words : Asynchronous Microprocessors, Performance Evaluation, Event-Driven Simulation, System-Level Simulation

· 본 연구는 한국과학재단 한일국제공동연구(20006-302-01-2) 및 교육부 BK21 사업에 의한 지원으로 수행되었음.
† 비 회 원 : MJL Technology 기술연구소 연구원
hessed@doctor.com
†† 비 회 원 : 광주과학기술원 정보통신공학과
eulia@kjist.ac.kr
††† 학생회원 : 광주과학기술원 정보통신공학과
uskim@geguri.kjist.ac.kr
†††† 종신회원 : 광주과학기술원 정보통신공학과 교수
dilee@kjist.ac.kr
논문접수 : 2001년 2월 23일
심사완료 : 2001년 10월 9일

1. 서 론

최근 급속도로 복잡해지고 커져가는 하드웨어를 고려할 때, 향후 10년 안에 4GHz에서 동작하는 10억 트랜지스터(billion transistors)급 프로세서가 나올 것이며, 100 BIPS(Billion Instructions Per Second)의 성능을 가진 프로세서들이 실생활에 쓰여질 것이다. 이러한 프로세서들의 구현에 있어서, 동기 방식 설계의 한계로 인해 비동기 방식의 설계는 큰 몫을 담당하게 될 것이다

[1]. 특히, 저전력·고성능 마이크로프로세서에 관한 관심이 점차 고조되면서 비동기 설계 방식이 하드웨어 설계자들 가운데 널리 확산되고 있다.

그러나 비동기식 설계의 필요성에도 불구하고 설계 과정을 효과적으로 지원해 줄 비동기식 설계 자동화 도구의 부재는 비동기식 설계의 확산에 가장 커다란 장애물로써 작용해 오고 있다. 이러한 문제를 해결하기 위하여 지난 10여년 간 비동기식 설계 자동화 도구에 관한 연구가 활발히 진행되어 왔으나 주로 비동기식 논리 합성기(logic synthesizer)의 개발에 한정되었다. 최근 들어, 설계하고자 하는 프로세서의 규모 증가와 더불어 다양한 종류의 비동기식 설계 자동화 도구에 관한 연구들이 요구된다. 본 논문에서는 대규모 프로세서의 설계 및 구현에 필수불가결한 비동기 마이크로프로세서의 모델링 및 시뮬레이터의 구현에 관한 작업을 수행하였다.

동기 마이크로프로세서의 설계에 사용되는 기존의 수많은 시뮬레이터들이 있음에도 불구하고, 동기 마이크로프로세서와 본질적으로 다른 특성을 갖는 비동기 마이크로프로세서의 설계 과정에서 그들을 직접 사용하는 것은 매우 힘들다. 즉, 전역 클럭의 발생에 따라 동작하는 동기 마이크로프로세서와는 달리 핸드셰이크[2][3] 방식에 기반한 모듈간의 통신에 의해 동작하는 비동기 마이크로프로세서는 다양하고 새로운 모델링 및 시뮬레이션 방법을 필요로 한다. 예를 들어, 하드웨어 설계자들이 가장 많이 사용하는 하드웨어 기술 언어인 VHDL이나 Verilog 등은 비동기 마이크로프로세서의 핸드셰이크 기반의 모듈간 통신 동작을 간결하고 정확하게 기술하지 못하므로 비동기 마이크로프로세서의 시뮬레이션 수행에 있어서 적합하지 못하다[4]. 이에 비동기 마이크로프로세서의 고유한 특성을 쉽고 효과적으로 표현하여 비동기 마이크로프로세서의 적절한 모델링 및 시뮬레이션을 위한 도구 개발에 있어서 다수의 시도가 있었다. 본 연구에서 개발한 도구와 추상성의 차이가 있겠지만, 하드웨어를 모델링하는 언어라는 측면에서 다음과 같은 관련 연구들과 비교 가능하다.

CSP[5]에 기반한 Tangram[6]과 Occam[7]이라는 언어가 있는데, Tangram과 Occam은 CSP에 기반한 범용 언어여서, VLSI에 관한 심도있는 지식 없이도, VLSI 시스템 기술이 가능하다. 또한, CSP 모델의 실용화를 목적으로 한 언어로서, 병행적으로 실행되는 기본 단위인 프로세스간의 메시지 교환을 위해 채널이라는 개념을 도입하여, 비동기 마이크로프로세서 기술에 편의를 제공하였다.

또한, Manchester 대학의 Amulet 그룹에서는 LARD (Language for Asynchronous Research and Development)[8]라는 언어를 개발하여 비동기 마이크로프로세서를 기술하는 데에 도움을 주었다. Tangram이나 Occam에 비해 고급 언어(high-level language)인 C에 가까운 형태를 지니고 있어서 쉽게 이용할 수 있다는 장점을 지니며, 언어 차원에서 채널이라는 개념을 도입함으로써 앞서 언급한 복잡성을 간단한 기술로 대치할 수 있게 되었다.

이외에도 본 연구에서 모델링에 사용하는 아키텍처 기술 방법과 흡사한 ARAS(Asynchronous RISC Architecture Simulator)[9][10]라는 도구가 있다. ARAS는 Washington 대학에서 개발한 그래픽 시뮬레이션 도구로, 편리한 사용자 인터페이스(user interface)를 제공하며, 간결한 입력 형태와 명령어 trace를 통하여 그래픽 출력 형태(visualization)를 지닌다. 하지만, 실제 시뮬레이션 핵심 부분에서는 미약한 점이 많이 나타난다. 특히, 실제 하드웨어의 동작을 그대로 묘사하기 위해서는, 마이크로프로세서의 행위 기술(behavior description)이 필수적임에도 불구하고, 마이크로프로세서의 행위 기술뿐만 아니라, 명령어의 행위 기술도 지원되지 않는 큰 단점이 있다. 단지, 모듈의 지연(delay)만 주어지고, 프로세서의 전체 속도나, 프로세서를 구성하는 각 모듈의 사용 정도를 구하는 것만을 목표로 삼는다. 또한, 아키텍처를 기술함에 있어서도, 선형 구조의 마이크로파이프라인(micropipelines)[2]만을 가정하기 때문에, 다른 형태의 아키텍처를 기술하기 힘들다.

본 연구팀이 개발한 비동기 마이크로프로세서 시뮬레이터인 *SLEDS*는 모델링할 비동기 마이크로프로세서의 아키텍처 기술(architecture description)과 주어진 아키텍처 모델(architectural model)상에서 수행될 명령어 trace를 입력으로 받는다. 이 때, 하드웨어 동작을 그대로 묘사하기 위해서 정의된 행위 기술에 따라 마이크로프로세서의 시뮬레이션을 수행한다. ARAS와 본 시뮬레이터의 궁극적인 차이는 행위 기술의 가능 여부에 있다. 시뮬레이터의 기능은 성능 평가 이외에도 기본적인 검증이 되어야 하는데, ARAS에서는 이 부분을 간과하고 있다. *SLEDS*에서는 보다 정확하고 다양한 정보를 얻기 위하여, 하드웨어 실제 동작을 그대로 묘사하여, 레지스터나 메모리의 참조 정보도 뽑아내게 된다. 또한 ARAS에서 반영하지 않는 모듈의 계층성을 *SLEDS*에서는 하위 계층을 의미하는 서브모듈(sub- module)의 개념으로 추가하였다.

본 논문의 구성은 다음과 같다. 총 6장으로 구성되어 있으며, 2장에서는 본 논문의 독자들의 이해를 돕기 위하

여 비동기 마이크로프로세서의 동작과 기존의 동기식 시뮬레이터의 문제점을 간략히 기술한다. 본 논문의 핵심인 3장과 4장에서는 본 연구팀이 개발한 *SLEDS*에서 비동기 마이크로프로세서를 모델링하고 시뮬레이션을 수행하는 방법을 설명한다. 5장에서는 몇몇 벤치마크(benchmark)에 대한 모의 실험 결과를 보여주고, 마지막으로 6장에서는 결론 및 향후 연구 과제로 논문을 맺는다.

2. 예비 지식

2.1 비동기 마이크로프로세서의 동작

그림 1과 2는 동기식 파이프라인과 비동기식 파이프라인의 제어 흐름을 나타낸 것이다. 기본적으로 동기 마이크로프로세서는 전역 클럭에 의해 전체 프로세서가 동작되므로, 각 모듈 간의 통신을 위한 특별한 프로토콜 없이도 전체적인 제어가 쉽게 이루어진다. 하지만, 비동기 마이크로프로세서는 전역 클럭에 기반하지 않고, 그림 2에서 보듯이 핸드셰이크에 기반한 각 모듈의 국부적인 통신에 의해서 자율적으로 동작하므로 고유의 프로토콜을 필요로 한다. 따라서, 기존의 하드웨어 기술 언어로는 비동기식 프로토콜의 기술이 복잡해지게 된다.

비동기식 프로토콜 제어를 위해 그림 2는 그림 1 보다 더 복잡하게 구현됨을 볼 수 있다. 기본적으로 데이타와 함께 Request-Acknowledgement 신호를 주고받음으로 각 모듈에서 주고받는 데이타의 유효성을 보장하게 만든다. 이러한 방식에 있어 bundled-data 방식과 dual-rail encoding 방식이 있다[11]. bundled-data 방식은 지연회로(delay padding)를 필요로 한다는 부담이 있지만, 쉽게 구현이 가능하며, 하드웨어 오버헤드가 적다. 반면에 dual-rail encoding 방식은 실제 데이타 한 비트를 표현하기 위해 두 개의 전선(wire)을 사용하여, 부가적인 지연회로 없이 고속으로 동작한다는 장점이 있지만, 그만큼 하드웨어 오버헤드가 커지게 된다.

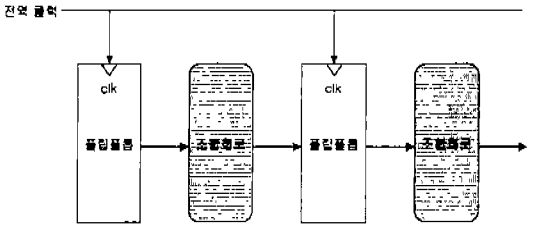

그림 1 일반적인 동기식 파이프라인

핸드셰이크 프로토콜로는 2-phase 방식과 4-phase 방식이 있다. 2-phase 방식은 상승 전이와 하강 전이가 같은 의미를 가지므로 4-phase 방식에 비해 빠르게 동작하지만, 상대적으로 구현이 어렵다는 단점이 있다[2] [3].

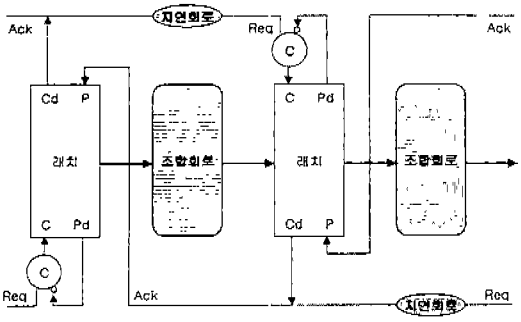

그림 2 비동기식 파이프라인(마이크로파이프라인)

2.2 동기 마이크로프로세서를 위한 시뮬레이션 도구의 부적합성

앞서 언급했듯이, 비동기 마이크로프로세서의 본질적인 특성 때문에, 동기 마이크로프로세서를 위한 시뮬레이션 도구를 그대로 비동기 마이크로프로세서에 적용하기에는 많은 문제가 있다. 그림 3의 네 가지 예시 모두 모듈 A와 B가 비동기적으로 통신하는 것을 기술하고 있지만, 각각의 비동기적 특성을 표현하는 능력 차이에 따라 기술이 달라짐을 볼 수 있다. VHDL과 같은 하드웨어 기술 언어에서는 비동기식 통신을 위한 Request-Acknowledgement 신호의 관계를 일일이 기술해 주어야 하지만, 비동기 마이크로프로세서의 특성에 적합하게 개발된 도구에서는 간략한 기술만으로도 복잡한 개념을 내부적으로 처리할 수 있다.

시뮬레이션 수행 과정 또한 비동기적 특성을 어떻게 기술하느냐에 따라 용이성과 효율성이 결정되는데, VHDL 기술이 복잡해지므로 시뮬레이션 수행 시간도 길어진다. 결과적으로 VHDL 시뮬레이터를 이용하여 비동기 마이크로프로세서의 성능 평가를 수행한다면 이중 오버헤드가 발생하게 된다. 반면에, *SLEDS*에서는 비동기식 핸드셰이크 프로토콜을 개념 그대로 자연스럽게 기술을 할 수 있고, 따라서 시뮬레이터 내부에서도 그러한 기술을 쉽게 인식하고 수행함으로써 시뮬레이션 속도 면에서도 큰 이득을 얻게된다.

시뮬레이션 엔진 측면에서도, 일정한 클럭 주기에 따라 동작하는 동기 마이크로프로세서를 위한 시뮬레이터는 일정한 사이클 시간마다 모든 모듈을 살펴보는 사이클중심 방식[12][13]이 적합하겠지만, 유동적 사이클 시간을 갖는 비동기 마이크로프로세서에는 적합하지 않다.

VHDL	process begin
	wait until A_Req=1;
	val:=A_Data;
	A_Ack<=1;
	wait until A_Req=0;
	A_Ack<=0;
	
	B_Data<=f(val);
	
	B_Req<=1;
	wait until B_Ack=1;
	B_Req<=0;
	wait until B_Ack=0;
	end process;
		LARD	forever(
			A?(
			val:=?A
			);
			B!f(val)
			)
				ARAS	"A B"	SLEDS	"A B"

그림 3 기존 도구 및 *SLEDS*의 기술 방법 예시

앞서 언급했듯이, 비동기 마이크로프로세서는 전역 클럭 없이 각 모듈이 정해진 프로토콜에 의해서 자율적으로 동작하기 때문에, 사건구동식 방법[12][13][14]은 비동기 마이크로프로세서의 동작 및 타이밍 관계를 자연스럽게 나타낸다.

사이클중심 방식은 전역 시간을 기준으로 단위 시간마다 모든 모듈에서 발생하는 이벤트들을 일일이 살펴보아야 하는 단점이 있지만, 이 연구에서 채택한 사건구동식 방법은 이벤트가 발생하는 모듈만을 살펴보면서 이벤트를 관리하는 장점이 있다.

3. 모델링 방법

상위 수준에서의 성능 평가를 통해서 설계자가 얻고자 하는 것은 설계 이전에 고려할 수 있는 부분, 즉 프로세서의 전체적인 성능을 감소시키는 병목 구간(bottleneck)이 되는 모듈이나 동작을 찾아내는 것이다. 실제 하드웨어 설계에 앞서 다양한 설계 공간을 탐색해 봄으로써 사전에 문제를 해결할 수 있다. 이러한 참조 모델의 신뢰성을 높이기 위해서는 하드웨어를 고려한 정확한 모델링은 필수 조건이 되겠고, 이와 더불어 다양한 설계 라이브러리를 제공하여 정확한 모듈 지연을 반영해야 한다. 상위 수준 시뮬레이터인 *SLEDS*에서는 이러한 점을 감안하여 설정 정보(configuration infor- mation) 형식을 사용하여 아키텍처의 구조를 기술하게 된다.

목표하는 마이크로프로세서의 아키텍처를 기술하기 위해서 설계자는, 우선 마이크로프로세서에 필요한 모듈들과 그들의 연결 관계를 정의해야 한다. 모듈에 대한 설정 정보와 명령어 그룹에 대한 설정 정보를 통하여 이루어지는데, 모듈 설정 정보에서는 각 모듈의 지연과 서브모듈에 관한 정보를 다룬다. 모듈은 아키텍처 기술과, 시뮬레이션 수행 및 결과의 기본 단위가 되며, 병행적으로 수행되는 프로세스의 의미를 갖는다. 한편, 명령어 그룹 설정 정보에서는 각 명령어가 지나며 수행하게 될 모듈의 수행 경로(execution path)가 정해지게 되는데, 동일한 수행 경로를 갖는 명령어들끼리 모은 것이 명령어 그룹이다.

제안된 설정 정보에서는 아키텍처의 위상 정보(topology information)만을 표현하며, 실제 행위는 설정 정보와 구분되어 각 모듈이나 명령어에 해당하는 함수에서 독립적으로 기술한다. 마이크로프로세서를 구성하는 각 모듈의 행위와 아키텍처 모델상에서 수행될 명령어들의 행위는 순수한 C/C++로 기술한다. 이런 방식으로 모듈과 명령어의 행위를 기술함으로써, 실제로 하드웨어가 수행되는 그대로 시뮬레이션을 수행할 수 있게 된다. *SLEDS*의 전체적인 흐름은 그림 4와 같다.

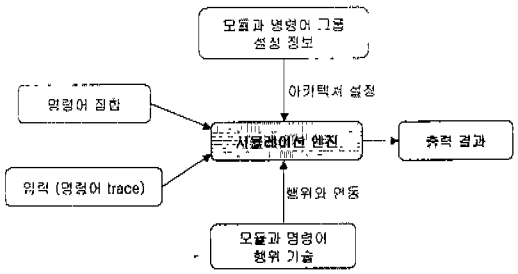

그림 4 *SLEDS*의 전체 흐름도

3.1 아키텍처 기술

아키텍처를 기술하기 위해서는 모듈 설정 정보와, 명령어 그룹 설정 정보가 필요하다. 필요한 모듈들을 정의할 때, 각 모듈의 처리 시간을 의미하는 지연과 서브모

듈 정보를 포함한다. 서브모듈이란, 원래 모듈의 기능을 다른 여러 모듈에 분산하여 수행하기 위한 하위 구조의 모듈을 의미한다. 즉, 원래 모듈을 몇 개의 서브모듈로 나눔으로써 전체적인 성능을 높일 수 있다.

아키텍쳐 기술에 있어 모듈 설정 정보와 더불어, 실제 모듈들의 연결 구조를 나타내는 위상 정보를 위해, 명령어 그룹 설정 정보가 있어야 한다. 모든 명령어들은 모듈 수행 경로에 따라 몇몇 그룹으로 나뉘어진다. 한 그룹에 속한 명령어들은 같은 모듈들을 지나며 수행하게 되는데, 예를 들어, 그림 5에서 우측 상단의 "1 2 3 5"라는 명령어 그룹에 속한 모든 명령어들은 모듈 1, 2, 3, 5에 해당하는 IF, ID, EX, WB를 순서대로 지나며 수행한다. 모듈 IF에서는 명령어 trace 화일에서 한 줄씩 입력으로 받는다. 모듈 ID는 읽어온 스트링을 opcode (operation code)와 각각의 오퍼랜드(operand)들로 구성 분석(parsing)하는 작업을 수행한다. 모듈 EX에서는 opcode에 따른 연산을 수행하게 되고, 마지막 모듈인 WB는 필요에 따라 결과를 다시 피드백(feedback)해야 하는 경우에 수행된다. 각각의 명령어 그룹 설정 정보가 주어지면 마이크로프로세서의 모듈들의 연결 구조를 파악할 수 있다.

그림 6의 모듈 EX는 모듈 설정 정보에 의해 EX_1, EX_2, EX_3라는 서브모듈을 포함하기 때문에, 비결정적으로 선택되어 수행하게 된다. 또한, 그림 6에서 주목할 점은 모듈을 그냥 지나가도록 하는 개념(bypass)을 추가한 기술 방법이다. 그림 6의 우측 상단 첫째 명령어 그룹 "1 2 3 4 _8 9"에서 모듈 8은 수행되지 않고, 그냥 지나가도록 하는 역할만 한다. 그림 5에서 '_' 없이 모듈 번호만을 적어주거나, 그림 6에서 모듈 번호 앞에 '_'를 포함하는 차이에 의한 위상 변화를 살펴볼 수 있다. 이렇게 모듈 설정 정보와, 명령어 그룹 설정 정보에 의해 아키텍쳐 기술이 이루어진다.

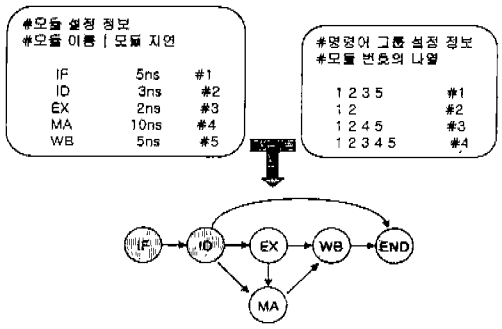

그림 5 모듈 설정 정보와 명령어 그룹 설정 정보를 통한 아키텍쳐 모델링 (I)

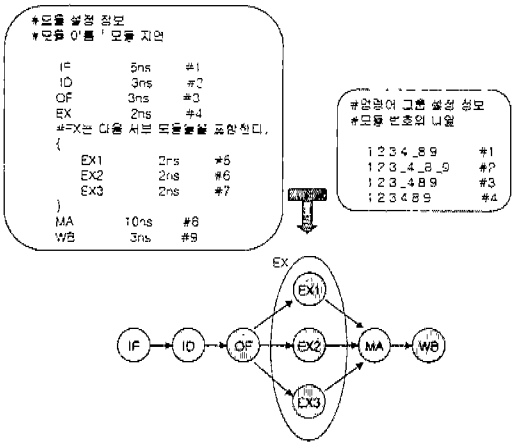

그림 6 모듈 설정 정보와 명령어 그룹 설정 정보를 통한 아키텍쳐 모델링 (II)

그림 5와 6은 서브모듈이라는 일종의 계층적 모델링에 따라 달리 표현된 아키텍쳐들이다. 예를 들어, 하나의 덧셈기만 지닌 경우라면 서브모듈이 필요없겠지만, 똑같은 'add' 명령어일지라도, 세 개의 다른 덧셈기 자원(resource)이 있는 경우에는, 매 연산마다 다른 덧셈기를 사용할 수 있게 된다. 이런 경우, 각 덧셈기의 구현이 다르므로 지연도 달라지고, 전체적인 성능도 이에 따라 달라질 수 있다.

3.2 행위 기술

3.2.1 모듈 행위

프로세서의 성능은 프로세서를 구성하는 모듈과 밀접한 관련이 있기 때문에, 모듈의 행위를 제대로 기술하는 것은 중요하다. 예를 들어, 명령어 인출(fetch)에 관련된 모듈에서는 실제 입력 화일로부터 명령줄을 읽어와야 하며, 명령어 해독(decode)에 관련된 모듈에서는 읽어온 명령줄을 opcode와 오퍼랜드들로 구분해야 한다.

이외에도 정의된 각 모듈의 행위 기술이 필요하게 된다. 이는 보다 정확한 성능 평가를 위함이다. 프로세서의 성능이란 전체적인 수행 시간을 의미하기도 하는데, 이는 프로세서를 구성하는 각 모듈 지연에 전적으로 의존한다. 따라서, 모듈 설정 정보에서 주어지는 고정된 모듈 지연만으로는 하드웨어의 동작을 그대로 반영할 수 없다. 이렇듯이 현재 모듈의 입력으로 들어오는 데이타에 따라 지연이 달라질 수 있는데, 이를 모듈 행위 기술에서 담당하게 된다.

현재 처리중인 명령어나 프로세서의 상태, 또는 처리하고자 하는 데이타에 따른 세밀한 성능 평가를 수행하

기 위하여 모듈 지연을 동적으로 할당한다. 그림 13의 모듈 행위 기술 예를 보면 각 함수의 마지막 부분에서 상수를 반환(return)해주는데, 이 값이 실제 동적 모듈 지연에 해당한다. 덧셈 연산 모듈에 대해 데이타 의존 연산(data dependent computation)에 따른 동적 모듈 지연을 반영하고자 할 때, 통계적인 접근 방식을 이용하여 모듈 지연을 계산해 내거나, 실제 입력되는 데이타를 분석하여 모듈 지연을 계산해 낼 수 있다. 구체적인 예로서, Ripple Carry Adder의 경우 처리 시간은 입력으로 들어오는 두 개의 데이타를 더하는 과정에서 발생하는 Longest Carry Propagation Path(LCPP)에 의존하므로 이를 분석하는 소스 코드를 모듈 행위 기술 부분에 삽입함으로써 데이타 의존적인 모듈 지연을 알아 낼 수 있다.

각 모듈의 행위 기술은 시뮬레이션 엔진과의 독립성을 최대한 유지함으로써, 설계자는 시뮬레이션 엔진의 내부 수행 과정을 고려하지 않고도, 그림 7과 같은 형태로 쉽게 기술할 수 있다. 또한, 모듈 행위 기술에 있어서 필수적이고 유용한 기능들은 붙박이 함수(built-in function)나 전역 변수로 제공한다.

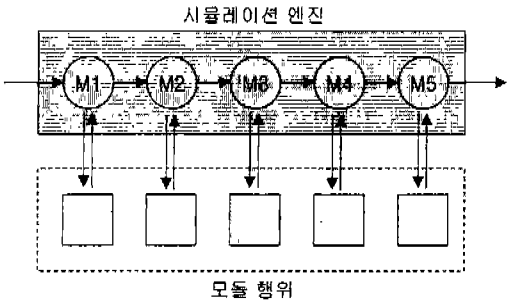

그림 7 시뮬레이션 엔진과 모듈 행위 기술의 독립성

3.2.2 명령어 행위

본 연구의 목표에서 얻고자 하는 결과인 전체 수행 시간과 모듈의 사용 정도와 더불어, 하드웨어의 동작을 그대로 묘사하기 위해서, 명령어 행위의 기술도 필요하다. 즉, 실제로 명령어를 수행함으로써, 메모리나 레지스터가 어떻게 사용되고 있는지 살펴볼 수 있다.

우선, 명령어 행위 기술을 위한 전처리 작업으로서, 명령어 집합(instruction set)을 정의한다. 그림 8은 명령어 집합의 일부를 보여주고 있다. 첫째 값은 opcode를 나타내고, 둘째 값은 명령어가 속한 명령어 그룹 번호이며, 마지막 셋째 값은 목적 필드(destination field)

```
#명령어 집합
#명령어 코드 | 그룹 번호 | 목적 필드 위치
    add          1          3
    ld           1          2
    mov          4          2
    jmp          4          -1
    ...
```

그림 8 명령어 집합의 예

의 위치이다. 예를 들어, 그림 8의 'add'는 명령어 그룹 번호가 1이고, 목적 필드가 셋째 필드라는 것을 의미한다. 따라서, 'add'는 첫째 명령어 그룹의 모듈 수행 경로를 따라 지나며 각 모듈을 수행하게 된다. 또한, 명령어 행위를 기술하는 함수에서 첫째 필드와 둘째 필드를 더하여, 그 결과를 셋째 필드에 넣을 수 있게 된다. 'jmp'의 경우를 보면, 명령어 그룹 번호는 4가 되고, 목적 필드는 필요없기 때문에, -1을 취하게 된다. 실제로 목적 필드는 데이타 의존성 검사시에 이용하며, 이에 대한 상세한 내용은 4.2절에서 다루기로 한다.

명령어 행위 기술은 여러 함수로 나뉘어질 수 있다. 같은 명령어라 할지라도, 각 모듈에서 서로 다른 기능을 수행할 수 있기 때문이다. 예를 들어, 'add %o1, 1, %o2'라는 명령어는 두 가지 행위를 필요로 한다. 첫째 행위는 레지스터 %o1의 값과 상수 1을 더하는 연산 행위가 되겠고, 둘째 행위는 덧셈 연산의 결과를 레지스터 %o2에 저장하는 것이다. 따라서, 하나의 'add' 명령어에 대해서 행위는 두 개로 나뉘어지게 된다.

명령어 행위를 기술하는 데에 있어서, 필수적인 것은 다양한 주소 방식(addressing mode)의 제공이다. 따라서, 시뮬레이션 엔진에서는 address[]와 value[]의 자료 구조를 동시에 제공함으로써, 다양한 주소 방식을 사용하여, 명령어 행위를 기술하게 만든다. address[]와 value[]는 정수 타입의 배열로서, address[]에는 레지스터의 번호(index number)나 메모리의 주소가 저장되어 있으며, value[]에는 실제 레지스터나 메모리의 데이타가 저장되어 있다. 예를 들어, add %o1, 3, %o2에 대한 address[im_ptr][0]에는 레지스터 %o1의 번호인 1이 들어있고, address[im_ptr][1]에는 상수 3 그 자체가 들어있고, address[im_ptr][2]에는 레지스터 %o2의 번호인 2가 들어있게 된다. 또한, value[im_ptr][0]에는 레지스터 %o1에 저장된 값이 들어있고, value[im_ptr][1]에는 상수 3이 들어있고, value[im_ptr][2]에는 레지스터 %o2에 저장된 값이 들어있게 된다. 따라서, 이러한 자료 구조를 통해 다양한 주소 모드를 이용할 수

있어, 설계자로 하여금 의도하는 바를 정확하게 표현할 수 있게 한다.

이와 같이 각 모듈에 따른 명령어 행위를 기술함으로써 실제 하드웨어 동작대로 수행이 가능해지며, 설계자는 임의의 시점에서 메모리나 레지스터 값의 변화를 관찰할 수 있게 된다. 실제 예제는 후반부 실험에서 다루기로 한다.

4. 시뮬레이션

시뮬레이션에서는 수행 속도가 상당히 중요한 요소가 되기 때문에, 시뮬레이션 엔진이 시스템의 특성을 효과적으로 반영하여 최대한 빠르게 수행해야 한다. 따라서 비동기 마이크로프로세서의 개념과 부합하는 사건구동식 방법을 채택하였다. 기존의 일반적인 사건구동식 엔진의 개념에 실제로 필요한 기능들을 추가하였다.

기존의 ARAS라는 도구와 구별되는 특징은, 아키텍쳐의 행위 기술이 추가된 점이고, 이를 위한 C/C++를 시뮬레이션 엔진에 끼워 넣는(embedded behavior description) 방식을 이용한다는 것이다. 따라서, 원시 프로그램을 한 번 컴파일한 후에는 상당히 빠른 속도로 시뮬레이션을 수행하여 결과를 얻어내는 장점을 갖는다.

4.1 사건구동식 시뮬레이션 엔진

4.1.1 이벤트 테이블

이벤트 테이블은 분산된 모듈에서 발생하는 모든 이벤트들을 중앙에서 효율적으로 관리하기 위한 순환큐(circular queue) 형태의 자료 구조이다. 표 1에서 보듯이 이벤트 테이블에는 명령어 그룹 번호(grp)와, 현재 수행되고 있는 모듈 번호(snd), 다음으로 수행해야 할 모듈 번호(rcv), 그리고 이벤트 삽입 시점(entry_time), 모듈의 지연(dly) 및 블록킹 정보(blked) 등을 담고 있다. 'ID'와 'int_blked'는 이벤트 테이블에서 관리되는 것이 아니라, 실제로 의존성 테이블(dependency table)의 값을 참조만 한다(표 2). 이벤트 테이블 내의 이벤트 선후 관계는 각각의 이벤트 삽입 시점과 모듈 지연에 의해서 관리된다. 따라서, 이들이 이벤트의 선후 관계를 결정하는 데에 있어서 중요한 역할을 한다. 삽입 시점과 지연의 합은 이벤트가 빠져나가야 하는 출력 시점을 의미하는데, 늦은 출력 시점을 갖는 이벤트보다, 상대적으로 이른 출력 시점을 갖는 이벤트가 수행 우선권을 가진다. 수행 후에 빠져나가는 이벤트는 또 다른 이벤트를 생성하여 테이블에 삽입하며, 현재 rcv 값은 새 이벤트에서 snd 값이 되고, 현재 수행중인 명령어의 그룹 번호에 따라 그에 해당하는 새로운 rcv 값이 정해진다.

4.1.2 이벤트 관리

상위 수준에서의 이벤트라 함은, 현재 모듈(sending module) 안에 있는 명령어가 다음 모듈(receiving module) 수행을 위해 보내는 신호(request)라 할 수 있다. 예를 들어, 표 1의 11번째 포인터(ptr)에 위치한 이벤트는 모듈 0(snd)에 있는 명령어가 다음으로 모듈 1(rcv)을 수행하고자 신호를 보내고 있음을 나타낸다. 프로세서를 구성하는 모듈 개수가 많아지면, 그만큼 발생하는 이벤트 수도 늘어나므로, 사건구동식 방법에서는 효율적인 이벤트 관리가 필수적이다.

전체적인 이벤트 수행 과정 알고리즘은 그림 9와 같다. 우선, 현재 모듈의 수행에 앞서 데이타 의존성에 의해 블록킹되어 있는 모듈을 풀어준다. 이는 4.2절에서 자세히 다루기로 한다. 블록킹된 모듈을 풀어준 후, 현재 명령어의 그룹 번호와 ID를 구한다. 그 후, 현재 모듈 안에 있는 명령어는 미리 기술된 모듈의 행위에 따라 수행된다. 수행 후, 새로 생성되는 이벤트를 이벤트 테이블에

표 1 이벤트 테이블

ptr	grp	ID	snd	rcv	entry_time	dly	blked	int_blked
11	N/A	N/A	0	1	10	0	NON	NON
12	1	0	6	7	14	3	NON	3
13	N/A	2	2	3	15	3	BLKED	NON
14	4	1	3	4	13	3	NON	BLKED

- ptr : 이벤트 테이블의 포인터
- grp : 명령어가 속한 그룹 번호 (N/A:미정)
- ID : 명령어의 ID (N/A:미정)
- snd : sending 모듈의 번호
- rcv : receiving 모듈의 번호
- entry_time : 이벤트의 삽입 시점
- dly : 모듈의 지연
- blked : receiving 모듈에 의한 블록킹 플래그 (NON:블록킹 안 됨. / BLKED:블록킹됨.)
- int_blked : 데이타 의존성에 의한 의도적인 블록킹 정보값
  (NON:블록킹 안 됨. / BLKED:블록킹됨. / 양수 k:모듈 k가 블록킹됨.)

```
void deal_with_request(int sending_module, int receiving_module, module m[], group g[], event_table evtab[], int
*head, int *tail)
{
    현재 명령어에 의해서 release 되어야 할 블록킹된 모듈이 있다면, release 시킨다.

    receiving_module의 명령어 그룹 번호와 명령어 ID를 얻는다.

    현재 모듈이 bypass 모듈이 아니라면, 모듈 행위를 수행한다.
    수행 후 가변 지연을 얻는다.

    next_module ← receiving_module 다음으로 수행해야 할 모듈 번호
    next_module의 서브모듈이 있을 경우, 서브모듈 번호를 구한다.

    next_delay ← 미리 정의된 BYPASS_DELAY, if (현재 모듈이 bypass 모듈인가?)
                 가변 지연, if (모듈 행위 수행에서 얻은 가변 지연이 유효한가?)
                 next_module의 정해진 지연, otherwise

    새로운 속성값을 모두 얻은 후에 생성된 이벤트를 이벤트 테이블에 삽입한다.
    receiving_module의 processing flag을 1로 만든다.
}
```

그림 9 이벤트 수행 과정 알고리즘

삽입하기 위해서 명령어 그룹 번호에 의해 다음으로 수행해야 할 모듈 번호(next_module_of_receiving_module)를 구하고, 그 모듈의 지연(next_delay)도 얻는다. 새로 삽입될 이벤트의 모든 속성값을 얻은 후에는 이벤트 테이블에 삽입할 수 있다.

이 때, 이벤트 테이블은 새 이벤트의 삽입 후에도 선후 관계가 유지되어야 한다. 사건구동식 방법에서의 최대 관건은 얼마나 효율적이고 신속하게 '가장 먼저 수행해야 할 이벤트'를 찾는가이다. 즉, 이벤트 삽입 시, 정렬된 순서를 유지하여, 상수 시간에 '가장 먼저 수행해야 할 이벤트'를 얻을 수 있어야 한다. 현재는 단순히 이벤트 출력 시점과 블록킹 정보값을 기준으로 오름차순 정렬을 하는 알고리즘을 채택함으로써, 이벤트 테이블의 최상위에는 항상 '가장 먼저 수행해야 할 이벤트'가 위치한다.

정렬시, 첫째 기준값은 다른 모듈에서 수행 중인 명령어와의 데이타 의존성에 의한 블록킹을 의미하는 'int_blked'가 된다. 이 블록킹 정보값은 한 모듈에서 수행 중인 명령어의 입력 오퍼랜드(source operand)가 다른 모듈에서 수행 중인 명령어의 결과값을 기다려야 하는 상황에서 변화한다. 따라서 다른 모듈에서 수행 중인 명령어가 결과값을 낼 때까지 현재 모듈의 명령어는 블록킹되어야 한다. 이 때 블록킹된 모듈의 블록킹 정보값은 -1을 갖게 된다. 반면에, 블록킹을 풀어줄 모듈의 블록킹 정보값은 블록킹된 모듈 번호를 갖게 된다. 데이타 의존성에 의한 블록킹은 4.2절에서 자세히 다루기로 한다. 둘째 정렬의 기준이 되는 값은 'blked'이다. 이는, 현재 모듈의 명령어가 수행되기 위해 선택될 수 있음에도 불구하고, 다음으로 수행해야 할 모듈에서는 아직 명령어의 수행이 끝나지 않았을 때 발생하는 순수한 블록킹 상황을 의미한다. 평상시 'blked'는 0을 유지하고, 블록킹이 발생하면 'blked'는 1을 갖는다. 마지막으로, 아무런 블록킹이 일어나지 않았을 때는, 이벤트의 출력 시점이 기준이 된다. 이벤트의 삽입 시점과 모듈 지연의 합인 출력 시점 순으로 정렬이 되어야 한다.

이렇게 세 가지 값을 기준으로 정렬된다면, 어떠한 순서 오류(sequence violation)도 발생하지 않는다. 따라서, 가장 빨리 수행되어 출력되어야 할 이벤트는 항상 이벤트 테이블의 최상위에 위치하게 된다.

4.2 데이타 의존성 관리

앞서 언급했듯이, 한 명령어가 다른 명령어의 결과를 기다림으로 인해, 수행되지 못하고 블록킹되는 상황이 발생하는데, 이른바 데이타 의존성(data dependency)[15]에 의한 블록킹이다.

표 2는 의존성 테이블의 예이다. 첫째 항목 'Instruction ID'는 현재 모듈에서 수행중인 명령어 ID를 나타낸다. EMP(-1)를 가진 모듈 2, 4, 6은 명령어를 수행하지 않고, 현재 비어 있음을 의미한다. 둘째 항목 'Blocking Instruction'은 블록킹시킨 명령어 ID를 나타낸다. 마지막 셋째 항목 'Blocked Module'은 세 가지의 의미를 갖는데, BLKED(-1)인 경우는 블록킹된 상태를 나타내고, NON(999)인 경우는 블록킹되지 않은 상태를 나타내며,

그 외의 양수를 가지는 경우는 그 값에 해당하는 모듈을 블록킹시키고 있음을 나타낸다. 따라서, 다른 모듈을 블록킹시키고 있는 상황의 정보는 'Blocking Instruction'과 'Blocked Module'이 항상 짝을 이루어 의미를 갖게 된다. 즉, 각각의 값은 블록킹시킨 명령어 ID와 블록킹된 모듈의 번호를 갖는다. 이 경우에 정보값이 저장되어 있는 모듈은 블록킹과 직접적인 관련이 있는 모듈은 아니고, 단지 블록킹을 푸는 시점의 역할을 한다.

표 2 의존성 테이블

module no. / attributes	1	2	3	4	5	6
Instruction ID	2	EMP	1	EMP	0	EMP
Blocking Instruction	N/A	N/A	N/A	N/A	N/A	0
Blocked Module	NON	NON	BLKED	NON	NON	3

- Instruction ID : 현재 모듈 안에 있는 명령어의 고유한 ID (EMP:비어 있음.)
- Blocking Instruction : 어떤 모듈의 명령어를 블록킹시킨 명령어의 ID (N/A:미정)
- Blocked Module : 블록킹된 모듈의 번호 (양수 k:모듈 k가 블록킹됨. / BLKED:자기자신이 블록킹됨. / NON:블록킹되지 않음.)

표 2에 나타난, 모듈 1, 2, 4, 5의 'Blocking Instruction'과 'Blocked Module' 값이 각각 N/A(-1)과 NON(999)이므로, 블록킹하지도 않으며, 블록킹되지도 않음을 알 수 있다. 또한, 모듈 3의 'Blocked Module'이 BLKED(-1)라는 것은 블록킹되고 있음을 의미한다. 그리고 모듈 6의 항목들을 주시하여야 하는데, 모듈 6에 새로 들어와서 수행될 명령어의 ID가 모듈 6에 저장된 'Blocking Instruction'의 값인 0과 일치하는 순간에, 'Blocked Module'의 값에 해당하는 모듈 3을 블록킹 상태에서 풀어주게 된다. 즉, 명령어 ID 0을 가진 명령어가 모듈 6에 도착한 순간에 모듈 3을 블록킹 상태에서 풀어준다. 데이타 의존성을 다루는 알고리즘은 그림 10과 같다.

5. 실 험

본 논문에서의 실험 결과는 크게 두 가지로, *SLEDS*를 통하여 일반적인 MIPS 아키텍처를 모델링한 것과, 임의의 아키텍쳐를 모델링한 후, 입력을 넣었을 때 나오는 출력 결과에 대한 것이다.

본 연구에서 목표로 하는 상위 수준 시뮬레이션의 결과로는 전체 프로세서의 수행 시간과 정의된 모듈의 사용 정도와 메모리나 레지스터 값의 변화를 나타내는 것으로 충분하다. 다음의 실험 예들을 통하여 *SLEDS*의 출력 결과를 살펴본다.

5.1 MIPS 모델링

```
// '모듈 m2'의 명령어가 '모듈 m1'의 명령어로 인해 데이타 의존성이 발생했을 경우, '모듈 m1'에 있는 명령어가 '모듈 m3'에 도착할 때까지. 의도적으로 블록킹시키는 함수
void intentional_blocking(int m1, int m2, int m3)
{
    dpdc_tab[m3].set_id(id_tab[m1].get());
    dpdc_tab[m3].set_module(m2);
    dpdc_tab[m2].set_module(BLOCKED_FLAG);
}

void release_blocked_module(int mn, int instr_id, double entry_time, event_table evtab[], int head, int tail, module m[])
{
    의존성 테이블로부터 블록킹된 모듈의 번호를 얻는다. → bm

    if (현재 모듈에 들어온 명령어의 ID와 의존성 테이블에 저장된 ID가 일치하는가?)
    {
        // 블록킹되었던 '모듈 bm'을 풀어준다.
        dpdc_tab[bm].set_module(DUMMY_MODULE_NUMBER);
        dpdc_tab[mn].set_id(DUMMY_INSTRUCTION_ID);
        dpdc_tab[mn].set_module(DUMMY_MODULE_NUMBER);

        블록킹되었던 이벤트의 삽입 시점을 갱신한다.
    }
}
```

그림 10 데이타 의존성 처리를 위한 알고리즘

표 3 MIPS 명령어 종류에 따른 functional units

명령어 종류	명령어 종류에 따른 functional units				
	IF	RF	EX	MA	WB
R-format	명령어 인출	레지스터 읽기	ALU		레지스터 쓰기
Load	명령어 인출	레지스터 읽기	ALU	메모리 읽기	레지스터 쓰기
Store	명령어 인출	레지스터 읽기	ALU	메모리 쓰기	
Branch	명령어 인출	레지스터 읽기	ALU		
Jump	명령어 인출				

표 3과 같이, MIPS 아키텍쳐에서는 총 5개의 파이프라인 단계(pipeline stage)로 구성되며, 각 단계에서 사용되는 functional units을 *SLEDS*에서 하나의 모듈로 정의한다. 즉, IF, RF, EX, MA, WB가 된다. 각 모듈의 지연은 비동기적인 프로토콜 오버헤드를 감안하여 가정하였다. 그림 11, 12는 *SLEDS*에서의 모듈 설정 정보와, 명령어 그룹 설정 정보를 보여주고 있다.

```
#모듈 이름 | 모듈 지연
IF          9
RF          8
EX          6
MA          13
WB          6
```

그림 11 모듈 설정 정보

```
#모듈 번호의 나열
1 2 3 5
1 2 3 4 5
1 2 3 4
1 2 3
1
```

그림 12 명령어 그룹 설정 정보

그림 13은 각 모듈의 행위를 기술하고 있다. 둘째 함수인 RF 모듈의 행위를 보면, decode()나 operand_fetch()를 이용하여, 복잡한 스트링 처리 등이 쉽게 이루어짐을 알 수 있다. decode()는 단지 스트링을 파싱(parsing)하는 함수이며, 파싱한 결과를 opcode나 오퍼랜드로 넘겨준다. 이는 프로세서의 아키텍쳐가 바뀌어도 항상 수행해 주어야 하는 모듈이므로, 매번 스트링을 파싱하는 함수를 구현하기 보다는 기존에 작성된 붙박이 함수 decode()를 이용하여 쉽게 구현할 수 있다. operand_fetch()도 이런 측면에서 다른 구조에서도 동일하게 사용될 수 있다. 셋째 함수 EX 모듈에 대한 행위에서도, 앞서 언급한 데이타 의존성 검사 등을 간단한 함수 하나로 대치하였다. 명령어 행위의 수행은 instruction_behavior()에서 연동시킨다. 특히, MA나 WB 모듈에서는 명령어의 행위 수행을 위한 함수와 연동시킨다. 명령어 행위 기술이 그림 14, 15, 16과 같이 나뉘어지는 이유는, 각 모듈에 따른 명령어 행위가 달라지기 때문이다. 이에 대한 설명은 3.2.2절에서 'add' 명령어를 예로 들어 이미 자세히 서술하였다.

```
int instruction_fetch(int im_ptr)
{
    if (flag_for_existence &&
        !fgets(instruction_memory[im_ptr].instruction,
        MAX_LENGTH, input_file_pointer))
            flag_for_existence = 0;

    return 0;
}
```

```
int register_fetch(int im_ptr)
{
    decode(im_ptr);
    operand_fetch(im_ptr);

    return 0;
}
```

```
int execute(int im_ptr)
{
    // data_dependency(id_tab[4].get(), id_tab[3].get()) means
    examining the dependency between instructions in Modules
    4 and 3.
    // intentional_blocking(m1, m2, m3) means that Module m1
    makes Module m2 be blocked, and released at Module m3.

    if(data_dependency(im_ptr, id_tab[3].get()))
        intentional_blocking(4, 3, 6);

    result[im_ptr] = instruction_behavior(instruction_memory[
    im_ptr].opcode, value[im_ptr]);

    return 0;
}
```

```
int memory_access(int im_ptr)
{
    instruction_behavior2(instruction_memory[im_ptr].opcode,
        address[im_ptr], value[im_ptr], main_memory, &result[
        im_ptr]);

    return 0;
}
```

```
int write_back(int im_ptr)
{
    instruction_behavior3(instruction_memory[im_ptr].opcode,
        address[im_ptr], result[im_ptr], main_memory,
        registers);

    return 0;
}
```

그림 13 각 모듈의 행위 기술

```
int instruction_behavior(char *opcode, int value[])
{
    int result;

    ...
    if (strcmp(opcode, "ld") == 0 || strcmp(opcode, "st") == 0 || strcmp(opcode, "mov") == 0)
                                                    result = value[0];
    else if (strcmp(opcode, "add") == 0)            result = value[0] + value[1];
    else if (strcmp(opcode, "sub") == 0)            result = value[0] - value[1];
    else if (strcmp(opcode, "sll") == 0)            result = value[0] << value[1];
    else if (strcmp(opcode, "srl") == 0)            result = value[0] >> value[1];
    else if (strcmp(opcode, "and") == 0)            result = value[0] & value[1];
    else if (strcmp(opcode, "or") == 0)             result = value[0] | value[1];
    ...

    return result;
}
```

그림 14 execution 모듈에 대한 명령어 행위의 기술

```
void instruction_behavior2(char *opcode, int address[], int value[], int main_memory[], int *result)
{
    if (strcmp(opcode, "load") == 0)                *result = main_memory[address[0]];
    else if (strcmp(opcode, "store") == 0)          main_memory[address[1]] = value[0];
}
```

그림 15 memory-access 모듈에 대한 명령어 행위의 기술

```
void instruction_behavior3(char *opcode, int address[], int result, int main_memory[], RF registers[])
{
    ...
    if (strcmp(opcode, "ld") == 0 || strcmp(opcode, "mov") == 0) registers[address[1]].set_value(result);
    else if (strcmp(opcode, "st") == 0) main_memory[address[1]] = result;
    else if (strcmp(opcode, "add") == 0 || strcmp(opcode, "sub") == 0)
        registers[address[2]].set_value(result);
    else if (strcmp(opcode, "sll") == 0 || strcmp(opcode, "srl") == 0 || strcmp(opcode, "and") == 0 ||
        strcmp(opcode, "or") == 0) main_memory[address[2]] = result;
    ...
}
```

그림 16 write-back 모듈에 대한 명령어 행위의 기술

5.2 시뮬레이션 예

5.2.1 비동기 마이크로프로세서

그림 17은 *SLEDS*를 통하여 기술한 비동기 마이크로프로세서에 C로 작성된 임의의 원시 프로그램을 역어셈블(disassemble)해서 얻어낸 20000개의 명령어를 넣었을 때 나온 결과이다. EX는 서브모듈 3개를 갖기 때문에, EX 대신 EX_1, EX_2, EX_3 중에서 비결정적으로 선택하여 수행한다. 따라서 각각의 활용 시간(busy time)이 거의 비슷하게 나뉘어짐을 알 수 있다. 또한, EX_1, EX_2, EX_3의 휴지 시간(idle time)이 크게 나타나는 이유는, IF, ID, OF에 의한 기아현상(starvation)이 일어나기 때문이다. 다시 말해서, IF, ID, OF의 지연이 각각 5, 3, 3인 것에 반해, EX의 지연은 2로서 상대적으로 작기 때문에 휴지 시간이 커지게 된다. WB는 마지막 모듈이므로, 이를 제외한 다른 모듈에서만 블록킹되는 상황이 발생한다. 특히, EX_1, EX_2, EX_3의 블록킹 시간(blocked time)이 크게 나타나는 이유는 그 다음 모듈인 MA의 지연이 10으로서 상대적으로 크기 때문이다. 또한, IF, ID, OF의 블록킹 시간이 큰 이유는 OF와 EX 사이의 데이타 의존성에 의한 블록킹이 일어나기 때문에 ID, IF로도 전달되는 현상이다. 즉, OF-EX, EX-MA 사이에 병목 현상이 발생함을 알 수 있다.

```
* total execution time = 198295.
* total number of instructions = 30000.
* system throughput = 0.151 instructions / unit time.

* final module utilization
 module        busy              idle              blocked
    IF    150000(75.6%)        9( 0.0%)        42286(24.4%)
    ID     90000(45.4%)    43151(21.8%)        65144(32.9%)
    OF     90000(45.4%)    31882(15.1%)        76413(38.5%)
    EX     ----------      ----------          ----------
   EX1     17313( 8.7%)   136587(68.9%)        44395(22.4%)
   EX2     17479( 8.8%)   136115(68.6%)        44701(22.5%)
   EX3     17457( 8.8%)   136182(68.7%)        44456(22.5%)
    MA    102855(51.9%)    82767(41.7%)        12673( 6.4%)
    WB     75846(38.2%)   122449(61.8%)         0( 0.0%)
```

그림 17 비동기 마이크로프로세서의 예

5.2.2 지연 균형 조절의 효과

그림 19, 21, 23은 동일한 아키텍쳐에 대해 각 모듈의 지연을 다르게 부과한 모듈 설정 정보이다. 그림 20, 22, 24는 각 모듈 설정 정보와 그림 18의 명령어 trace를 입력으로 받아 출력으로 나오는 각각의 결과이다. 이 결과에서, 본 도구의 목표 중에 하나인 지연 균형 조절(delay balancing)의 효과를 보여주고 있다. 세 모델은 모두 1부터 10까지 합을 구하는 명령어 trace를 입력으로 받아, 결과로서 레지스터 o1의 값인 55를 동일하게 내지만, 성능면에서는 모델 AS1이 세 모델 중에서 가장 좋게 나타난다.

```
mov 0, %o1
add %o1, 1, %o1
add %o1, 2, %o1
add %o1, 3, %o1
add %o1, 4, %o1
add %o1, 5, %o1
add %o1, 6, %o1
add %o1, 7, %o1
add %o1, 8, %o1
add %o1, 9, %o1
add %o1, 10, %o1
```

그림 18 1부터 10까지 합을 구하는 명령어 trace

#모듈 이름	모듈 지연
IF	5
ID	3
OF	2
EX	4

그림 19 모듈 설정 정보 (모델 AS1)

```
* total execution time = 74.
* total number of instructions = 11.
* system throughput = 0.149 instructions / unit time.

* final module utilization
 module        busy              idle              blocked
    IF     55(74.3%)       15(20.3%)         4( 5.4%)
    ID     33(44.6%)       26(35.1%)        15(20.3%)
    OF     22(29.7%)       18(24.3%)        34(45.9%)
    EX     44(59.5%)       30(40.5%)         0( 0.0%)

Registers
                ... o1=55 ...
```

그림 20 합 구하기 결과 (모델 AS1)

#모듈 이름	모듈 지연
IF	4
ID	4
OF	4
EX	4

그림 21 모듈 설정 정보 (모델 S1)

```
* total execution time = 96.
* total number of instructions = 11.
* system throughput = 0.115 instructions / unit time.

* final module utilization
 module        busy              idle              blocked
    IF     44(45.8%)       16(16.7%)        36(37.5%)
    ID     44(45.8%)       16(16.7%)        36(37.5%)
    OF     44(45.8%)       12(12.5%)        40(41.7%)
    EX     44(45.8%)       52(54.2%)         0( 0.0%)

Registers
                ... o1=55 ...
```

그림 22 합 구하기 결과 (모델 S1)

#모듈 이름	모듈 지연
IF	6
ID	4
OF	3
EX	5

그림 23 모듈 설정 정보 (모델 AS2)

일반적으로 비동기 마이크로프로세서가 동기 마이크로프로세서에 비해 고성능을 낸다고 하는데, 비동기 마이크로프로세서의 프로토콜 오버헤드가 너무 커지면 오히려 성능 손실을 초래할 수도 있다. 이런 측면에서 모델 S1과 모델 AS1의 성능을, 또한 모델 S1과 모델 AS2의 성능을 비교할 필요가 있다.

```
* total execution time = 98.
* total number of instructions = 11.
* system throughput = 0.112 instructions / unit time.

* final module utilization
 module        busy          idle          blocked
   IF       66(67.3%)     19(19.4%)     13(13.3%)
   ID       44(44.9%)     26(26.5%)     28(28.6%)
   OF       33(33.7%)     19(19.4%)     46(46.9%)
   EX       55(56.1%)     43(43.9%)      0( 0.0%)

Registers
                ...   o1=55   ...
```

그림 24 합 구하기 결과 (모델 AS2)

그림 18은 데이타 의존성으로 인해 블록킹이 발생하는 명령어 trace이다. 블록킹이 전혀 일어나지 않는 명령어 trace를 입력으로 받는 파이프라인의 구조라면, 당연히 각 모듈의 지연이 같은 동기 마이크로프로세서가 우수한 성능을 내겠지만, 이 예에서는 블록킹이 빈번히 발생하는 명령어 trace이기 때문에, 지연이 일정한 동기 마이크로프로세서 보다는 비동기 마이크로프로세서가 적합함을 유추할 수 있다. 실제로 결과에서도, 지연이 4로 모두 같은 동기식 모델 S1보다 비동기식 모델 AS1의 성능이 좋게 나타난다. 모델 AS1에 대한 모듈 설정 정보인 그림 19를 보면 알 수 있듯이, 각 모듈의 지연에 최소의 프로토콜 오버헤드만을 가정하였다. 하지만, 모델 AS2의 모듈 설정 정보인 그림 23을 보면 모델 AS1에 비해 프로토콜 오버헤드를 1만큼 더 크게 설정한 경우에는, 그 성능이 동기 마이크로프로세서인 모델 S1보다 열등하게 나옴을 알 수 있다.

이와 같이 본 도구 *SLEDS*를 통하여, 비동기 마이크로프로세서뿐만 아니라, 동기 마이크로프로세서의 모델링과 시뮬레이션이 가능하며, 동기식 설계와 비동기식 설계의 교환 조건(trade-off) 관계를 파악할 수 있게 된다.

6. 결론 및 향후 연구 과제

본 연구를 통하여 비동기 마이크로프로세서의 성능을 평가하기 위한 모델링 및 시뮬레이션 방법론을 확립하고, 그에 기반한 자동화 시뮬레이션 도구 *SLEDS*를 개발하였다. 본 도구 *SLEDS*는 기본적으로 마이크로프로세서와 같은 구조의 시스템을 모델링하고 시뮬레이션하는 것을 목표로 삼고 있으며, 마이크로파이프라인 형태의 FIFO와 같은 간단한 시스템도 가능하다.

설계자의 관심사는 전체 프로세서를 이루는 모듈의 사용 정도에 관한 결과이다. 모듈 지연이나 행위가 전체 프로세서의 성능을 좌우하기 때문이다. 특히, 각 모듈의 활용 시간과 휴지 시간 및 블록킹 시간은 지연 균형 조절에 중요한 요소가 된다. 다시 말해서, 모듈의 사용 정도를 보고, 정의된 모듈들이 얼마나 효율적으로 사용되는지, 혹은 전단 모듈(predecessor)에 의한 기아현상이나, 후단 모듈(successor)에 의한 병목 현상은 발생 안하는지 밝혀낼 수 있다.

*SLEDS*의 특징은 손쉬운 아키텍쳐 기술을 통한 모델링과 사건구동식 시뮬레이션 엔진에 모듈 행위 기술과 명령어 행위 기술을 C/C++로 끼워 넣는 방식을 이용한다는 것이다. 각 모듈과 명령어의 행위는 C/C++로 기술되어, 하드웨어 기술 언어에 국한되기보다는 기존의 프로그래밍 언어를 사용하여 기술함으로써 좀 더 많은 융통성을 갖는다. 기존의 도구와 추상성 측면에서 차이가 있겠지만, C/C++로 개발한 시뮬레이션 엔진에 C/C++로 기술한 행위 기술을 끼워 넣는 형태가 되므로 일단 컴파일된 실행 화일에 대해서는 시뮬레이션 수행 속도 측면에서 기존의 도구보다 우세하다. 반면에, 모듈이나 명령어의 행위를 변경할 때마다, 전체적으로 다시 컴파일을 해야 한다는 단점이 생긴다.

또 다른 장점으로는, 동일한 아키텍쳐에 대하여 동기식으로 설계했을 때와, 비동기식으로 설계했을 때의 성능을 비교분석할 수 있다는 것이다. 이는 넓은 의미에서, 본 연구의 목표 중에 하나인, 지연 균형 조절의 효과로 볼 수 있다. 동기 마이크로프로세서는 최악 성능(worst case performance)을 내는 데에 반해, 비동기 마이크로프로세서가 갖는 큰 장점 중에 하나는, 평균 성능(average case performance)을 낸다는 것이다. 그 이유는 전역 클럭 없이, 모듈간의 핸드셰이크 프로토콜을 통해 전체 프로세서가 자율적으로 제어되기 때문이다. 하지만, 프로토콜에 의한 오버헤드가 생기는 단점이 있다. 이런 측면에서, 동기식과 비동기식 교환 조건을 어느 정도까지 허용한 것인지 결정해야 할 경우가 생기는데, 이런 면에서 *SLEDS*는 길잡이가 되어 준다. 이는 개념적으로 비동기 마이크로프로세서가 동기 마이크로프로세서를 포함하므로 비동기 마이크로프로세서를 위해 개발한 환경에서도 동기 마이크로프로세서의 모델링과 시뮬레이션이 가능하기 때문이다.

앞으로의 남은 과제는 프로세서를 모델링하는 부분에 있어서 미약한 점이 많이 나타나는데, 이 부분을 보완하는 것이다. 현재로서는 모듈 설정 정보, 명령어 그룹 설정 정보 등의 것만으로 아키텍쳐를 기술하지만, 조금 복잡한 아키텍쳐를 위해서라면 보다 많은 확장성을 제공해야 한다. 특히, 슈퍼스칼라(superscalar)[15]나 VLIW

(Very Long Instruction Word)[16]와 같은 아키텍처를 모델링하고 시뮬레이션을 수행하기 위해서, 현재 하나의 이벤트만 수행하는 모듈의 개념을 두 개 이상의 이벤트가 수행 가능하도록 확장하는 것이 불가피하다. 또한, 비동기 마이크로프로세서의 큰 특징인 데이타 의존 연산(data dependent computation)을 제대로 구현하기 위한 추가 작업도 필요하겠다.

참 고 문 헌

[1] Rakefet Kol and Ran Ginosar, "*Future Processors will be Asynchronous (sub-title: KIN: A High Performance Asynchronous Processor Architecture),*" Technical Report CC PUB#202 (EE PUB#1099), Department of Electrical Engineering, Technion - Israel Institute of Technology, July 1997.

[2] Ivan E. Sutherland, "*Micropipelines,*" Communications of the ACM, 32(6):720-738, June 1989.

[3] Stephen B. Furber and Paul Day, "*Four-Phase Micropipeline Latch Control Circuits,*" IEEE Transactions on VLSI Systems, 4(2):247-253, June 1996.

[4] Pirouz Bazargan-Sabet and Huu-Nghia Vuong, "*Trade-offs in Designing a VHDL Simulation Tool for Digital VLSI,*" Proceedings of the 26th Southeastern Symposium on System Theory, pp. 380-384, 1994.

[5] C. A. R. Hoare, "*Communicating Sequential Processes,*" London: Series in Computer Science, Prentice-Hall International, 1985.

[6] Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs, and Frits Schalij, "*The VLSI-programming language Tangram and its translation into handshake circuits,*" Proceedings of the European Conference on Design Automation (EDAC), pp. 384-389, 1991.

[7] Georgios K. Theodoropoulos, G. K. Tsakogiannis, and J. V. Woods, "*Occam: An Asynchronous Hardware Description Language?*" Proceedings of the 23rd IEEE EUROMICRO Conference '97 New Frontiers of Information Technology, 1997.

[8] Philip Endecott and Stephen B. Furber, "*Modelling and Simulation of Asynchronous Systems using the LARD Hardware Description Language,*" Proceedings of the 12th European Simulation Multiconference, pp. 39-43, Manchester: Society for Computer Simulation International, June 1998.

[9] Chia-Hsing Chien and Mark A. Franklin, "*Simulation of Asynchronous Instruction Pipelines,*" Proceedings of 1996 Summer Computer Simulation Conference, pp. 155-162, July 1996.

[10] Chia-Hsing Chien, Mark A. Franklin, Tienyo Pan, and Prithvi Prabhu, "ARAS: Asynchronous RISC Architecture Simulator," Proceedings of 2nd Working Conference on Asynchronous Design Methodologies, pp. 210-219, London, England: IEEE Computer Society Press, May 1995.

[11] Al Davis and Steven M. Nowick, "*An introduction to asynchronous circuit design,*" Technical Report UUCC-97-013, Department of Computer Science, University of Utah, September 1997.

[12] Sabih H. Gerez, "*Algorithms for VLSI Design Automation,*" pp. 167-193, JOHN WILEY & SONS, 1999.

[13] Prithviraj Banerjee, "*Parallel Algorithms for VLSI Computer-Aided Design,*" pp. 441-457, Englewood Cliffs, New Jersey: PTR Prentice Hall, 1994.

[14] Wen-King Su, "*Reactive-Process Programming and Distributed Discrete-Event Simulation,*" pp. 61-83, Ph. D. Thesis, California Institute of Technology, Pasadena, California, 1990.

[15] Mike Johnson, "*Superscalar Microprocessor Design,*" Englewood Cliffs, New Jersey: PTR Prentice Hall, 1991.

[16] Joseph A. Fisher, "*Very Long Instruction Word Architectures and the ELI-512,*" Proceedings of the 10th Annual International Symposium on Computer Architecture, pp. 140-150, Stockholm, Sweden, June 1983.

최 상 익

1999년 서강대학교 전자계산학과 공학사. 2001년 광주과학기술원 정보통신공학과 공학석사. 2001년 ~ 현재 MJL Technology 기술연구소 연구원. 관심분야는 비동기 회로 설계 자동화, Embedded Processor, FPGA를 이용한 SoC 설계 등

이 정 근

1996년 한림대학교 전자계산학과 학사. 1998년 광주과학기술원 정보통신공학과 석사. 1998년 ~ 현재 광주과학기술원 정보통신공학과 박사과정. 관심분야는 비동기 회로, 병렬 및 분산 계산, formal methods 등

김 의 석

1995년 연세대학교 전산과학과 학사. 1997년 광주과학기술원 정보통신공학과 석사. 1997년 ~ 현재 광주과학기술원 정보통신공학과 박사과정. 관심분야는 병행 시스템(Concurrent Systems) 해석 및 설계, Petri Nets 이론, 비동기 회로 설계 및 CAD 등

이 동 익

1985년 영남대학교 전기공학과 학사. 1989년 오오사카 대학 전자공학과 석사. 1993년 오오사카 대학 전자공학과 박사. 1993년 ~ 1994년 일리노이 대학 컴퓨터 공학과 방문연구원. 1990년 ~ 1995년 오오사카 대학 전자공학과 문부교관. 1995년 ~ 현재 광주과학기술원 정보통신공학과 부교수. 관심분야는 병행시스템(Concurrent Systems) 해석 및 설계, Petri Nets 이론, 이동 에이전트 시스템, 보안시스템, 비동기 회로 설계 및 CAD 등