논리회로 시뮬레이터에 있어서 실행상태의 동적패턴 추출과 고속화

이 필 우† · 板野肯三††

요 약

본 논문에서는 논리회로 시뮬레이션시 상태의 동적 보존과 재이용 방식을 개발하여 시뮬레이션시의 계산 효율성을 향상시킬 수 있는 방법을 제시하였다. VHDL을 비롯한 많은 하드웨어 기술언어(hardware description language)로 기술된 하드웨어를 시뮬레이션하기 위한 계산비용은 대단히 크며 많은 양의 계산을 필요로 한다. 이런 하드웨어 시뮬레이션시의 처리과정을 정밀하게 분석한 결과, 하드웨어의 기술 전체가 적당한 크기의 모듈로 분할되어 있으면, 각 모듈의 상태전이(state transition)의 패턴은 그다지 많지 않은 것으로 나타났다. 따라서 패턴이 처음 발생했을 때 시뮬레이터에 동적으로 보존하여, 시뮬레이션 중에 같은 패턴이 반복될 경우 보존한 패턴을 이용함으로써 시뮬레이션에 필요한 계산을 대폭 감소시킬 수 있다는 것을 확인하였다. 본 논문에서는 몇 가지 시뮬레이션 사례연구를 통하여 본 방식의 유효성을 입증하였다.

Dynamic Pattern Abstraction of a Logic Circuit Simulator and Its Speed Up

Phil Woo Lee† · Kozo Itano††

ABSTRACT

This paper presents the methodology to improve the computation efficiency of the simulation by developing the concept of the dynamic preservation and reutilization of the state transitions. The computation cost is emormous for the simulation of hardware described in hardware description languages including VHDL. Analyzing the process of simulation precisely, we have found that the number of the patterns for the state transition is limited if the sizes of hardware modules are determined properly. The patterns are preserved dynamically when they appeared first, and are utilized in later simulation in order to reduce the simulation costs. In this study, the efficiency of the present method was verified using case studies for the simulation.

1. 서 론

컴퓨터 등의 하드웨어를 설계하기 위한 방법으로서, 적절한 레벨에서의 하드웨어 기술언어에 의한 기술과 시뮬레이션에 의한 동작의 검증은 중요한 부분을 차지하고 있다[1,2,3,4]. 그러나 시뮬레이션 처리를 위한 계산비용은 일반적으로 매우 높으며, 또한 하드웨어의 기술과 검증은 LSI 설계에서도 꼭 필요한 도구가 되어 있는 현재, 대규모의 중요한 계산이라고 말할 수 있다[5,6,7,8,9].

이러한 계산비용을 경감하기 위한 방법으로서는 목적에 따라 보다 높은 레벨에서의 기술과 시뮬레이션을 하는 방법, 또는 높은 레벨과 낮은 레벨을 혼합 기술하

†준 회 원 : 일본 筑波대학 대학원 공학연구과 박사과정
††비 회 원 : 일본 筑波대학 전자정보공학계 교수

논문접수 : 1998년 4월 30일, 심사완료 : 1998년 6월 22일

여 시뮬레이션 하는 이른바 혼합모드 시뮬레이션 방법이 이용되기도 한다. 그러나 낮은 레벨에서의 기술이 하드웨어의 표현에 보다 가까운 것은 확실하기 때문에 낮은 레벨로 기술된 하드웨어의 시뮬레이션을 고속화하는 것은 중요한 의미가 있다고 할 수 있다 [10,11,12,13].

따라서 본 연구에서는 논리회로 시뮬레이션 시스템에 있어서 하드웨어의 실행시 상태천이를 분석해 보았다. 일반적으로 논리회로의 실행시의 상태는, 하드웨어 전체를 두고 보면 같은 상태가 나타나거나 천이를 반복하는 경우는 적다. 그러나 하드웨어 전체를 보다 작은 모듈로 분할하여, 각 모듈 레벨에서의 상태에 주목해 보면 같은 상태가 나타나는 비율은 매우 높고, 같은 패턴의 상태천이도 많이 반복되는 것을 쉽게 예측할 수 있다.

극단적인 예로서, 하나의 플립플럽(flipflop)의 단위에서 나타나는 상태는 한정되어져 있으며 완전히 같은 상태천이가 반복된다. 따라서 하드웨어 전체를 적당한 크기의 모듈로 분할 설정할 수 있다면 전체의 상태를 보다 작은 상태의 집합으로 표현할 수 있고, 그것이 가능하다면 시스템 전체의 동적인 상태를 대폭 단순화시킬 수 있다. 이에 본 연구에서는 상태의 분할과 반복패턴의 축적을 전체적으로 행하지 않고 시뮬레이션 시에 발생한 패턴을 학습하여 축적시켜 나가는 방식을 개발하였다.

본 논문에서는 저자들이 제작한 VHDL 언어[5,6,7]의 서브세트(subset) 시뮬레이터를 사험 사례로 사용하여, 이 방법의 적용가능성을 확인한 결과에 대하여 설명하고자 한다. 시뮬레이션은 타임휠(time wheel)을 이용하며, 계산 방식은 사건구동식(event driven)으로 실행하는 방식을 채용하였다.

2. 상태천이 그래프

시뮬레이션의 대상인 하드웨어의 상태 변화는 그 시점의 모듈의 입력과 상태에 의해서 결정되기 때문에 상태변화의 패턴은 기본적으로 오토머톤(automaton)으로서 표현 가능하다. 하지만 VHDL에 있어서 하드웨어의 동작모델은 지연을 이용한 신호대입문에 의하여 신호의 미래 값을 표현하기 때문에 각각의 상태에 머무는 경과시간은 상태에 따라 다르다. 또 하드웨어의 상태를 전부 기술하는 것이 아니라 시뮬레이션시 발생한 상태를 순차적으로 추가해 가는, 논리적으로는 불완전한 오토머톤이기 때문에, 저자들은 오토머톤의 개념을 확장하여 상태천이 그래프(state transition graph)라는 데이터 구조를 고안하였다.

2.1 상태의 표현

먼저, 상태천이 그래프의 상태는 현재의 신호값 뿐만 아니라 타임휠에 등록되어 있는 미래의 사건(event)도 상태로서 포함시킬 필요가 있다. 구체적으로 상태로서 기록하는 것은, 그 시점에서 모듈이 가지는 신호의 값과 각 신호의 미래의 사건으로서 확정되어진 값을 한 쌍으로 한 것이다. 이 예를 (그림 1)에 제시한다.

신호명	값	사건리스트
Set	1	
Clr	1	
Clk	0	(10ns,1)
Data	0	(8ns,1)
Q	0	
QBar	1	
A	0	
B	1	
C	1	
D	1	
QI	0	
QBI	1	

(그림1) 모듈 상태의 데이터구조
(Fig. 1) Data structure of a module state

신호 Clk와 Data에 관해서는 각각 10ns후와 8ns후의 값으로서 1이 등록되어져 있고, 나머지 신호에 관해서는 미래의 값이 미등록된 상태이다. 이와 같이 상태 중에 미래의 사건이 포함되어 있으면, 이 상태는 미래의 상태를 구속하게 된다. 즉, (그림 1)에 표현한 상태의 경우에는 입력의 변화가 없을 경우 8ns후에 새로운 상태로 전이하는 것이, 이 상태 중에 기술되어져 있는 것이다. 물론 8ns가 경과하기 전에 입력이 변화하는 경우에는 그 입력에 따라 일단 다른 상태로 전이하게 된다.

2.2 상태의 전이

상태천이 그래프에서 상태와 상태간의 전이는, 보통의 오토머톤과 같이 화살표로 표현한다. (그림2)에서는 상태 001(신호 A, B, C의 값으로 상태를 표현했을 경

다. 이 경우는 화살표에 입력신호가 붙어 있지 않기 때문에 10ns 사이에 모듈의 입력에 변화가 없었다는 것을 나타내고 있다.

이와 달리, 입력이 존재할 경우의 상태전이의 예를 (그림3)에 제시한다. 이 경우에는 상태 001이 5ns 경과 후에 입력 A가 0에서 1로 변환되어 상태 101로 전이하는 경우와, 8ns 경과후 입력 A가 1로 변환되어 상태 101로 전이하는 경우, 그리고 입력의 변화 없이 10ns후에 상태 010으로 전이하는 경우를 나타내고 있다.

신호명	값	사건리스트
A	0	
B	0	(10ns,1)
C	1	(10ns,0)

10ns →

신호명	값	사건리스트
A	0	
B	1	
C	0	

(그림2) 입력변화가 없는 경우의 상태전이
(Fig. 2) State transition without input change

신호명	값	사건리스트
A	0	
B	0	(10ns,1)
C	1	(10ns,0)

A=1, 5ns →

신호명	값	사건리스트
A	1	
B	0	(5ns,1)
C	1	(5ns,0)

A=1, 8ns →

신호명	값	사건리스트
A	1	
B	0	(2ns,1)
C	1	(2ns,0)

10ns →

신호명	값	사건리스트
A	0	
B	1	
C	0	

(그림3) 입력변화가 있는 경우의 상태전이
(Fig. 3) State transition with input change

2.3 안정상태와 미평가상태

(그림3)의 상태 010와 같이 상태가 신호의 현재값만으로 구성되어 신호의 미래 값이 하나도 등록되어져 있지 않은 상태를 안정상태(stable state)라 부르기로 한다. 안정상태는 입력이 변화하지 않으면 자동적으로 다른 상태로 전이하는 경우는 없다. 피드백을 갖는 논리회로인 순서회로에서는 입력의 변화가 있은 후 일정한 시간 내에 안정상태에 도달한다. 상태전이 그래프상에서는 안정상태로부터 입력변화를 동반하지 않는 전이는 존재하지 않는다.

상태전이 그래프는 기본적으로 지연평가형의 계산모델에 따라 작성하기 때문에 논리적으로 존재하는 모든 전이가 작성되는 것은 아니다. 따라서 어떤 상태로부터 나타나야 하는 다음의 전이상태가 존재하지 않는 경우도 있다. 즉, 전이할 상태가 존재하지 않는 것이 반드시 그 상태가 존재하지 않는 것을 의미하지는 않는다. 이와 같이 미래에는 존재해야 할 것임에도 불구하고 아직 tL 시점에서 평가되어지지 않았기 때문에 존재하지 않는 상태를 미평가상태라고 부르기로 한다.

3. 시뮬레이션 알고리즘

시뮬레이션을 개시하여 제2절에서 설명한 바와 같이 상태전이 그래프가 작성되어 가면, 같은 상태패턴이 반복해서 나타났을 경우에는 하드웨어 기술의 신호대입문의 값을 하나하나 계산하지 않고 이미 보존되어 있는 상태를 그대로 사용할 수 있게 된다. 이 절에서는 시뮬레이션의 알고리즘에 관하여 자세히 설명한다.

3.1 순수 스크래치 모드

순수 스크래치(pure scratch) 모드에서는 모듈 중의 각 신호(signal)에 대해서 드라이버(driver)라 불리는 데이터 구조를 생성하여, 신호의 현재 시각의 값과 미래의 값을 보존하고, 시간의 경과에 따라 드라이버를 갱신하여 신호의 현재 값을 변경해 가는 방법으로 시뮬레이션을 진행한다. 이 과정을 (그림4)에 나타낸다.

0	+10ns	+5ns	+10ns	.....
	1	0	1	

(그림4) 드라이버의 데이터구조
(Fig. 4) Data structure of a driver

드라이버에 미래의 신호 값을 사건으로서 등록하는 것은 신호대입문이 평가되어진 시점에서 하며, 신호대

입문의 평가는 신호대입문 우변의 신호 값에 변화가 발생하였을 때 한다.

일반적으로 대부분의 모듈은 순서회로로 구성되어있기 때문에 시뮬레이션은 처음에 어떤 신호가 형식적으로 변화하면 그 영향을 받은 다음 신호가 변화하고, 이것이 몇 사이클 후 처음의 신호가 다시 변화함으로써, 드라이버에 미래 값의 등록이 사이클 적으로 이루어진다. 여기서 사건 구동 계산방식을 채용하여 신호의 현재 값에 동일한 값이 대입되어지는 것을 방지하면 실질적으로 신호가 변화했을 때만 현재 값의 변경이 이루어지도록 할 수 있다.

또 안정상태에 도달하면 드라이버 중에 미래 값은 존재하지 않으며, 드라이버에 미래 값을 등록하는 사이클은 자동적으로 절단된다. 한편 모듈에 대해 입력이 존재할 경우, 즉 VHDL의 포트(port) 신호가 변화하였을 때는 포트에 대한 드라이버로 사건이 등록됨으로써 시뮬레이트 되어지지만 이것은 부정기적으로 발생하기 때문에 상태전이를 일으키는 원인이 된다.

3.2 그래프 생성 모드

순수 스크래치 모드에서는 상태를 생성하지 않지만 순수 스크래치 모드와 같이 시뮬레이션을 하고, 이에 대응하여 항상 새로운 상태를 생성하여 상태전이 그래프에 그 상태를 추가하는 모드가 그래프 생성모드이다. 이 때 만약 그래프 중에 같은 상태가 존재할 경우에는 모드가 다음에서 설명하는 리사이클(recycle)모드로 전환된다. 그래프 생성모드로부터 리사이클모드로 전환될 때는 드라이버 중에 있는 미래의 사건을 모두 상태에 기록하고 드라이버를 소멸시킴으로써 사건의 윤회를 절단한다.

상태전이 그래프는 시뮬레이션 개시와 동시에 모듈별로 작성한다. 그 알고리즘을 단계별로 자세히 설명하면 다음과 같다.

step 1 : 시뮬레이션 사이클 중에 모듈의 상태가 변화했을 때에는 임시의 상태를 생성하여, 먼저 모듈의 현재상태에 링크 되어져있는 상태에 임시상태와 같은 상태가 존재하는지 검색한다. step 2로 간다.

step 2 : 같은 상태가 존재한다면 임시상태를 지우고 step 3으로간다. 존재하지 않는다면 모듈의 상태전이 그래프 전체를 검색하여 그 중에 임시상태와 같은 상태가 존재하는지 검색한다. 존재한다면 임시상태를 지우고 step 4로 가고, 존재하지 않으면 step 5로 간다.

step 3 : 그 상태를 모듈의 현재상태로 변경하고, step 1로 간다.

step 4 : 모듈의 현재상태로 부터 그 상태에 링크를 하고, 현재상태의 경과시간을 링크 중에 보존한 후 step 3으로 간다.

step 5 : 임시상태를 모듈의 상태전이 그래프에 추가하고 step 3으로 간다.

3.3 리사이클 모드

이 모드에서는 드라이버를 사용하지 않고 상태전이 그래프 중에 보존되어있는 상태에 그대로 전이하는 것만으로 시뮬레이션이 가능하다. 따라서 신호대입문을 평가하지 않아도 실행가능하기 때문에 시뮬레이션을 가장 고속화할 수 있는 모드이다. 이 모드에서는 상태전이 그래프 중의 현재상태에 다음 상태가 링크 되어져 있다면 기본적으로 그 상태를 타임휠에 등록하여 시뮬레이션 하면 된다. 구체적인 알고리즘은 아래와 같다.

step 1 : 타임휠에 등록되어져 있는 상태에 출력이 존재하면 그 출력을 모듈의 출력으로써 출력신호에 연결되어진 모듈의 드라이버에 등록한다.

step 2 : 타임휠에 등록되어져 있는 상태를 모듈의 현재 상태로서 변경한다.

step 3 : 모듈의 상태전이 그래프의 새로운 현재상태 다음에 입력신호의 변화가 없는 상태가 존재한다면 그 상태를 타임휠에 등록하여 종료한다. 단, 상태를 타임휠에 등록할 때에는 상태간의 링크에 보존되어져 있는 현재상태 경과시간을 참조한다.

step 4 : 모듈의 새로운 상태의 다음에 입력변화가 없는 상태가 존재하지 않을 경우, 만약 현재의 상태가 안정상태라면, 리사이클모드 그대로 대기 플래그(flag)를 세운다. 안정상태가 아니라면 현재상태 중에 보존되어 있는 미래의 사건을 드라이버로서 재생하여 그래프 생성모드로 전환한다.

3.4 입력대기 모드

시뮬레이션 모드가 리사이클모드일 때, 다른 모듈로부터 신호가 입력되면 모드를 입력대기 모드로 전환하여 입력신호를 드라이버에 보존한다. 입력대기 모드로 전환할 때는 타임휠에 등록되어 있는 상태를 소멸시킨다.

입력대기 모드에서 타임휠의 입력 사건이 도착하면, 그 입력이 상태전이 그래프의 현재상태에서 다음 상태로 전이하기 위한 조건으로서 존재하는지를 검색하여 만약 존재한다면 그 상태를 모듈의 새로운 상태로 선택하면 되고, 존재하지 않는다면 그래프 생성모드로 전환한다.

4. 계층적 모듈과 상호작용

VHDL에서는 하드웨어를 계층형으로 정의 할 수 있도록 추상화의 개념이 있다. 이 개념 그대로 상태를 보존하면, 상위모듈의 상태가 너무 커질 수 있으므로 하드웨어 정의시, 모듈의 계층구조의 처리에 관해서는 특별히 주의를 기울였다.

4.1 계층구조의 전개

VHDL의 모듈을 정의할 때는, *component*문을 이용하여 다른 모듈 정의인 *entity*에 선언함으로써 그 인스턴스(instance)를 모듈 내에 생성시킬 수 있다. 이 인스턴스는 논리적으로는 포함한 쪽의 모듈 내부에 놓여지지만, 이 논리적 구조 그대로 상태보존의 모듈단위로 사용하면, 포함한 쪽의 모듈이 포함되어진 쪽의 모듈의 상태를 포함하게 된다. 따라서 이러한 것을 방지하기 위해, 본 연구의 시뮬레이터에서는 포함되어진 쪽의 모듈의 인스턴스를 포함한 쪽의 모듈밖에 놓아 외부모듈과 같이 취급하고 있다.

4.2 상태의 공유보존

VHDL의 *entity*의 정의는 기본적으로 클래스(class)이므로 인스턴스에는 공통이다. 따라서 상태의 보존은 정확히 말하면 *entity*단위로 한다. 때문에 모듈로부터 보면 상태의 패턴은 같은 *entity*로부터 생성되어진 모듈간에서 공유할 수 있다.

(그림3)에 나타난 상태전이 그래프의 데이터 구조에는 리사이클모드에서 사용하는 모듈의 출력이 표시되어져 있지 않다. 출력은 모듈의 상태가 변경되어질 때, 모듈 외부에 출력이 있으면 그것을 상태에 보존한다.

보존의 출력은 *entity*단위로 보존할 수 없으므로 인스턴스인 모듈 단위에서 보존해야 한다. 이 때문에 상태선이 그래프에 모듈의 인스턴스를 식별할 수 있는 태그(tag)를 붙여 기록하고 있다.

4.3 입력의 계속적 변동

입출력을 갖는 블랙박스로서 모듈을 볼 때, 어떤 입력이 변화해서부터 상태가 일정한 값에 수렴할 때까지 입력이 변화하지 않는 경우에는 모듈 내에 동작의 독립성이 존재한다. 때문에 모듈내부의 동작계산은 외부로부터 분리시킬 수 있다. 하지만 상태가 수렴하는 과정에서 모듈의 입력이 변화하면, 이것이 상태의 비연속적 변화를 일으키게 된다. 하지만 모듈의 여러 개 입력이 동시에 변화하는 경우는 거의 없으며 변화하는 시각에 미묘한 시차가 발생하는 경우가 많다.

이와 같이 입력이 계속적으로 변화하면 기록되어지는 상태의 변화패턴이 안정상태로 수렴되지 않고 비평가상태가 많이 발생하게 된다. 리사이클모드 시뮬레이션 중에 미정의상태가 다음의 상태로 선택되어지면, 그래프 생성모드로 전환된다. 그 후 입력이 도착하면 상태를 계산하여 이 상태가 상태전이 그래프에 존재하는지를 검색하여 존재할 경우 리사이클모드로 복귀한다. 그래프 생성모드와 리사이클모드 사이의 전환은 시뮬레이션의 계산 비용이 매우 높다. 따라서 본 시뮬레이터에서는 미정의상태를 그래프에 추가할 경우에는, 그래프 생성모드로 전환되는 경우를 줄이기 위해서 더미(dummy)상태 한 개를 먼저 계산하여 추가하고 있다.

5. 시뮬레이션 중의 상태수

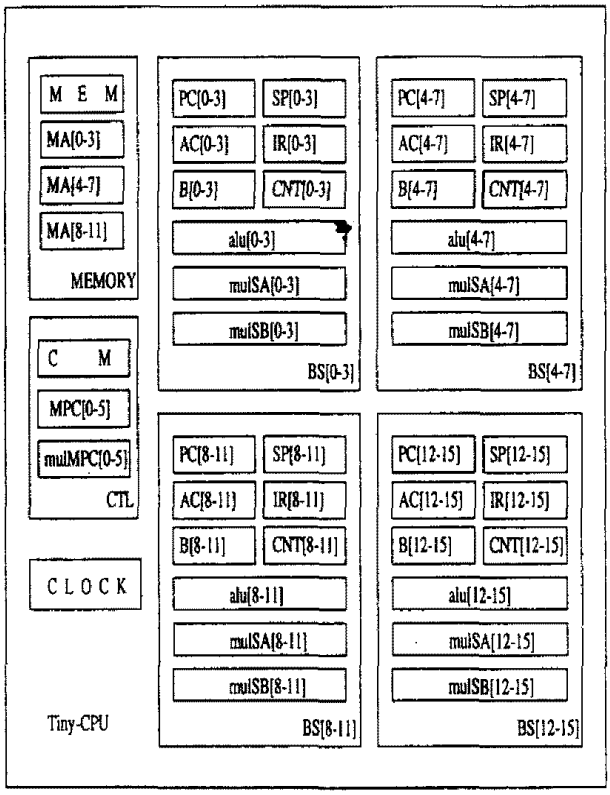
시뮬레이션 중의 상태 수는 모듈의 동작에 의존하지만, 모듈 분해의 방법에도 큰 영향을 받는다. 이 절에서는 몇 가지 프로그램을 실행하여 얻어진 결과에 의해 상태수가 나타나는 경향을 분석한다. 시뮬레이션 대상으로서는 간단한 명령세트를 갖는 CPU(Tiny-CPU)를 VHDL로 기술하여, 이 Tiny-CPU상에서 matrix계산, sort, ackerman함수 프로그램을 실행하였다.

5.1 CPU의 모듈구조

VHDL로 기술한 Tiny-CPU의 모듈구조를 간단히 설명하면 다음과 같다. 기본 데이터의 길이는 16비트로써, 장착한 레지스터는 PC(program counter), AC

(accumulator), B(보조 register), CNT(counter), IR(instruction register),SP(stack pointer), MA (memory address register), MPC(micro program 용 program counter)의 8가지이며, 메모리는 MEM (main memory), CM(micro program memory)의 2가지를 장착하였다. 이 이외의 모듈로서는 연산기(ALU)와 연산기의 2가지 입력을 선택하는 멀티플렉서 2개 (mulSA, mulSB), 클럭 발진기 등이 있다.

명령세트로서는 기본명령으로 LOAD, STORE, ADD, SUB, JUMP, JN(Jump-On-Negative), CALL의 7가지이며, 확장 명령으로서는 IN, OUT, HALT, MULT, RETURN의 5가지이다. 어드레싱 모드는 직접 어드레스와 간접 어드레스를 사용하며 인덱스 수식 기능은 없다.



(그림5) Tiny-CPU의 모듈구조
(Fig. 5) Module Structure of a Tiny-CPU

Tiny-CPU의 기술에 사용한 모듈의 포트수, 신호수, 신호대입문수를 〈표 1〉에 제시한다.

레지스터는 각 비트를 논리회로의 조합으로 구성한 D-플립플롭(DFF)으로 기술하였다. 입력신호로서는 WRITE용 데이터와 WRITE신호, RESET신호가 있으며, 각 비트 값이 항시 출력되어진다. Tiny-CPU의 내부구성으로서는 비트 슬라이스 모듈(bit slice module: BS)을 도입하여 모듈의 분할을 계층적으로 설계하였다. 슬라이스 되어진 비트는 4비트, 3비트, 2비트, 1비트의 4가지 종류를 기술하여 비교하였다. 연산기의 연산 종류는 덧셈과 뺄셈 등 필요 최소한으로 한정하며, 곱셈은 쉬프트(shift)와 덧셈의 조합으로 실행하는 방식을 채택하였다. 또 제어는 64마이크로 명령의 마이크로 프로그램으로 제어하고 있다. (그림5)에 4비트인 경우의 Tiny-CPU모듈구조를 제시한다.

〈표 1〉 각 모듈의 신호수
〈Table 1〉 Signals of each module

module	in-port	out-port	signals	assigns
CM	6	31	31	31
CLOCK	0	2	3	4
mulMPC[6]	31	6	4	10
reg[6]	8	6	31	37
reg[5]	7	5	25	30
reg[4]	6	4	20	24
reg[3]	5	3	15	18
reg[2]	4	2	10	12
reg[1]	3	1	5	6
mulSA[4]	27	4	6	10
mulSB[4]	15	4	3	7
alu[4]	14	5	20	25
alu[3]	12	4	16	20
alu[2]	10	3	12	15
alu[1]	8	2	8	10
BS[4]	29	9	59	24
BS[3]	28	8	45	20
BS[2]	26	6	20	16
BS[1]	25	5	20	16
CTL	15	21	31	9
MEMORY	21	16	14	2
Tiny-CPU	0	0	292	98

5.2 출현 상태수

시뮬레이션은 메모리 모듈의 메모리 내용을 모두 보존하는 것은 어려우므로 MEM만은 순수 스크래치모드로 실행하고, 나머지 모듈은 동적으로 모드를 전환해 가며 실행하였다. 〈표 2〉는 4행4열의 행렬계산(matrix (4x4)), 9개의 요소를 갖는 quick sort(sort(9)), ackerman함수(acker(2,3))의 프로그램을 실행하였을

때, Tiny-CPU에 있어서 출현한 상태 수를 나타낸다. 이 표 중에서 nonshare(n)는 비트폭 n의 모듈로 분할한 Tiny-CPU상에서 모든 모듈의 상태를 독립된 인스턴스별로 보존했을 경우의 상태 수이고, share(n)는 같은 *entity*로 정의되어진 모듈의 상태를 클래스 단위로 일괄하여 보존했을 경우의 상태 수이다.

〈표 2〉에서 알 수 있듯이 모듈별로 독립적으로 실행하였을 때의 상태 수와 *entity*별로 일괄하여 실행하였을 때의 상태수의 차이가 별로 없는 것은 모듈별로 출현하는 상태 중에 공통의 상태가 그다지 많지 않다는 것을 의미한다. 이것은 당초 예상하지 못했던 것으로, 특히 4비트의 레지스터는 Tiny-CPU에서 27개를 사용하고 있으나, 출현하는 상태 중에 공유하는 상태는 전체의 5%이하이다. 비트 수가 감소하면 비율은 높아지나, 1비트인 경우에도 상태공유율은 15~22% 정도이다.

다음, 4비트의 레지스터 27개에 대해 레지스터마다의 상태출현수를 〈표 3〉에 나타내었다. 레지스터명[m-n]의 m-n은 레지스터의 비트 위치를 의미한다. 예를 들어 4-7은 레지스터의 비트 4부터 7까지의 비트를 뜻한다. MSB측의 레지스터의 상태수가 적은 것은, Tiny-CPU상에서 실행되어지는 프로그램의 레지스터 상위부분이 0에 가까운 값이 많기 때문이다.

〈표 2〉 출현상태의 총 수
〈Table 2〉 Total created states

	matrix(4x4)	sort(9)	acker(2,3)
nonshare(4)	151459	141654	129213
share(4)	144835	137118	124814
nonshare(3)	145825	137331	127066
share(3)	139109	131792	121996
nonshare(2)	143092	132991	124304
share(2)	135562	126959	118779
nonshare(1)	140883	130977	122573
share(1)	134161	125413	117301

5.3 상태수의 수렴

시뮬레이션 중에 상태가 새롭게 발생하는 모습이 어떻게 변화해 가는가를 행렬계산 프로그램을 실행했을 경우에 대해 관찰하였다. 그 모습을 (그림6)에 제시하였다. x축은 시뮬레이션 사이클 수로 0에서 140만 사이클까지, y축은 발생한 상태의 누적 수를 나타내고 있다. 약 15만 사이클까지는 상태수가 급격히 증가한 다음, 이후는 서서히 증가하는 것을 알 수 있다. 이 성향은 슬라이스된 비트 수가 1부터 4까지인 경우 모두 동일하다. 상태의 증가율이 완만해지는 15만 사이클 이후의 증가율은 슬라이스가 1비트인 경우에는 더욱 완만해진다.

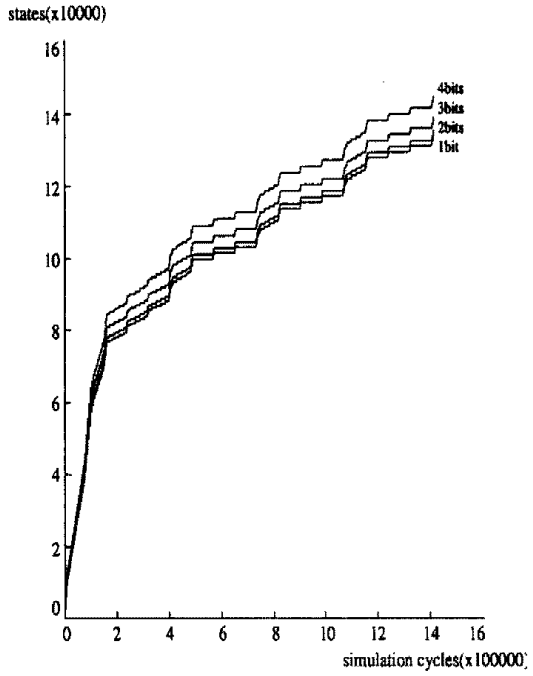
상태의 생성을 레지스터 모듈에 한정해서 보면 극히 양호한 수렴 결과를 얻을 수 있다. 레지스터의 비트 폭이 4비트,3비트,2비트,1비트의 경우의 상태 수 증가 모습을 (그림7), (그림8), (그림9)에 제시한다.

표 3〉 각 레지스터의 발생 상태수
〈Table 3〉 Created states for register modules

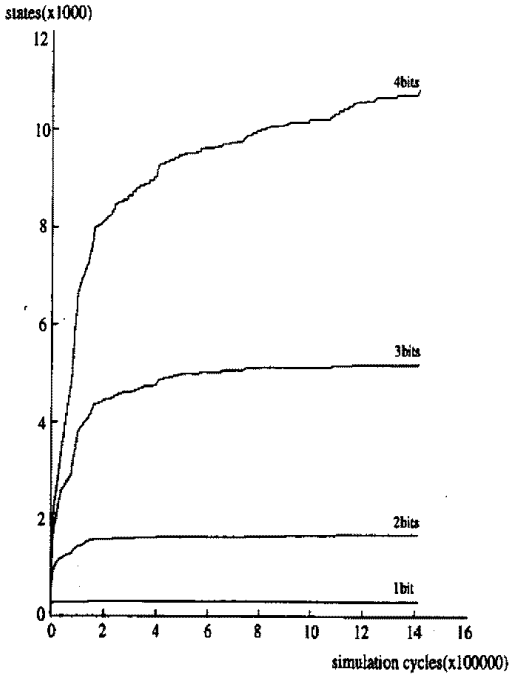
	matrix(4x4)	sort(9)	acker(2,3)
CNT[0-3]	46	11	51
CNT[4-7]	79	16	50
CNT[8-11]	113	18	51
CNT[12-15]	888	403	33
PC[0-3]	19	28	27
PC[4-7]	41	40	56
PC[8-11]	444	426	403
PC[12-15]	1537	1776	1516
AC[0-3]	324	185	94
AC[4-7]	364	190	60
AC[8-11]	493	306	187
AC[12-15]	3766	3765	2191
B[0-3]	72	10	8
B[4-7]	68	10	8
B[8-11]	98	13	8
B[12-15]	268	10	8
SP[0-3]	11	11	34
SP[4-7]	33	17	134
SP[8-11]	42	42	324
SP[12-15]	105	163	2006
IR[0-3]	346	348	283
IR[4-7]	208	109	241
IR[8-11]	284	321	196
IR[12-15]	2298	3084	622
MA[0-3]	163	81	237
MA[4-7]	476	452	411
MA[8-11]	3553	3498	2177

x축은 시뮬레이션 스텝으로 행렬계산의 경우 시뮬레이션 개시 후부터 140만 사이클까지(그림7), sort의 경우 60만 사이클까지(그림8), ackerman함수에서는

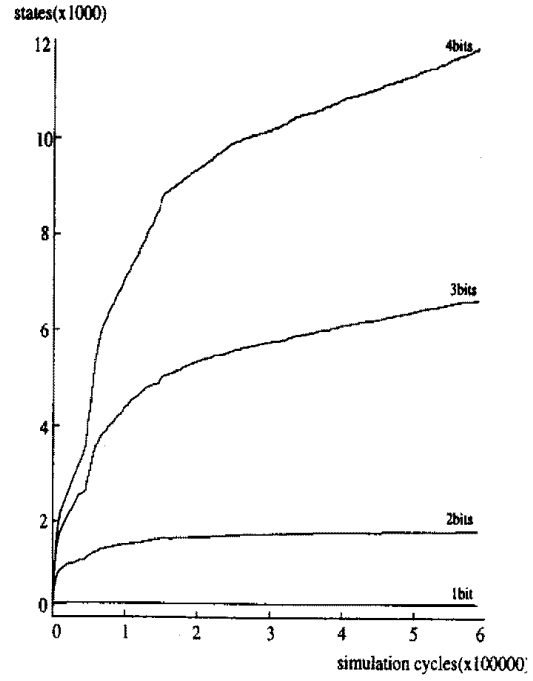
35만 사이클까지(그림9)를 나타내고 있다. y축은 상태 수를 나타낸다. 행렬 계산에서는 3비트 이하에서, sort와 ackerman함수에서는 2비트 이하에서 수렴되어지는 것을 알 수 있다.



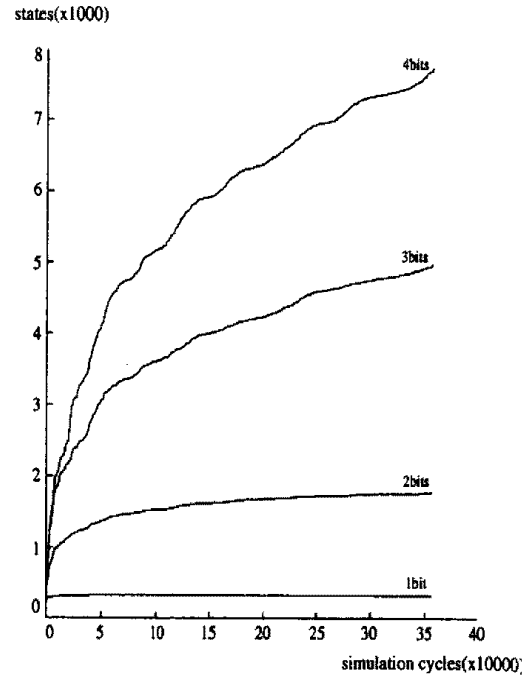
(그림 6) 행렬 계산을 실행한 경우의 상태 생성 (share)
(Fig. 6) State creation for matrix multiplication in the case of share



(그림 7) 행렬계산을 실행한 경우 레지스터모듈의 상태 생성
(Fig. 7) Register state creation for matrix multiplication



(그림 8) sort를 실행한 경우 레지스터모듈의 상태 생성
(Fig. 8) Register state creation for sort



(그림 9) ackerman 함수를 실행한 경우 레지스터모듈의 상태 생성
(Fig. 9) Register state creation for ackerman

6. 시뮬레이션의 평가

일반적으로, 상태를 보존하게 되면 기존 상태와의 비교 또는 상태의 보존과 회복 등의 처리가 필요하게 되며, 이러한 처리는 시뮬레이션 시간을 증가시키는 원인이 된다. 하지만 각 신호마다 사건처리를 하지 않아도 되는 경우, 즉 리사이클모드와 입력대기모드에서는 시뮬레이션 시간을 감소시키는 효과가 있다. 그러므로 이러한 경우들의 균형으로 시뮬레이션 시간이 단축되지는지 여부가 결정된다. 아래에 제시하는 데이터는 Ultra-SPARC(143MHz) + 64MB메모리 상에서 측정한 결과이다.

6.1 상태의 등록

상태의 등록에 필요한 시간을 측정하기 위하여, 상태가 없는 경우에서 출발하여 시뮬레이션이 종료할 때까지 필요한 시간과, 시뮬레이션에 필요한 모든 상태를 보존한 후 시뮬레이션한 경우와의 시간차를 생성된 상태수로 나누어 처리 비용을 측정하였다. 이와 같은 방법으로 산출한 상태등록에 필요한 시간을 〈표 4〉에 제시한다. 이 표에서 알 수 있듯이 새로운 상태가 발생하는데 필요한 시간은 약 100마이크로 초 정도이다.

〈표 4〉 상태 등록에 필요한 시간
〈Table 4〉 State creation times
(단위: 마이크로초/상태등록)

	matrix(4x4)	sort(9)	acker(2,3)
nonshare(4)	92.4	85.4	85.1
share(4)	91.8	85.3	84.9
nonshare(3)	91.2	85.9	84.9
share(3)	90.5	85.7	84.4
nonshare(2)	99.2	89.4	87.6
share(2)	98.8	89.0	85.0
nonshare(1)	105.2	99.2	99.5
share(1)	105.0	98.8	98.8

6.2 상태전이

먼저, 각 모듈의 시뮬레이션 모드 비율을 측정하였다. 이 비율은 모듈마다 다르며 실행하는 프로그램마다 다르지만, 그래프 생성모드의 비율은 약 2.6%에서 4.4% 정도이며 나머지 대부분이 리사이클모드와 입력대기 모드로 진행되었다.

다음, 〈표 5〉에 상태전이 1회에 필요한 처리시간을 제시한다. 이 시간은 일단 시뮬레이션에 필요한 상태를 모두 보존한 후, 순수한 상태전이에 필요로 하는 실행시간을 측정하여 총 상태전이 수로 나눈 결과이다. 비교를 위해 〈표 6〉에 상태를 등록하지 않는 순수 스크래치모드에서 상태전이 1회에 대응하는 처리시간을 제시하였다. 엄밀히 말하면 순수 스크래치모드에서는 상태를 생성하지 않기 때문에 상태전이의 개념은 없지만, 똑같은 시뮬레이션 대상 프로그램을 실행하였을 경우, 상태전이 수는 같으므로 순수 스크래치 모드만으로 시뮬레이션한 경우의 실행시간을 총 상태전이수로 나누어 측정하였다. 이 결과에서 알 수 있는 것은 드라이버를 이용하여 시뮬레이션 하는 것에 비하여 리사이클모드로 시뮬레이션 하는 경우가 약 2배 정도까지 시뮬레이션이 고속화되어진다는 사실이다.

〈표 5〉 상태전이에 필요한 시간
〈Table 5〉 State transition times
(단위:마이크로초/상태전이)

	matrix(4x4)	sort(9)	acker(2,3)
nonshare(4)	15.5	16.3	16.2
share(4)	15.1	15.9	15.5
nonshare(3)	13.2	13.5	14.0
share(3)	12.8	13.2	13.4
nonshare(2)	13.1	13.4	13.9
share(2)	12.7	13.0	13.3
nonshare(1)	11.7	11.9	12.1
share(1)	11.1	11.5	11.2

〈표 6〉 스크래치 시뮬레이션 시간
〈Table 6〉 Scratch simulation times
(단위:마이크로초/상태전이)

bit-slice	matrix(4x4)	sort(9)	acker(2,3)
4	31.9	31.3	30.7
3	24.5	25.2	24.8
2	24.1	24.6	23.9
1	19.4	19.9	19.6

6.3 리사이클 시뮬레이션의 효과

마지막으로 〈표 7〉과 〈표 8〉에 리사이클모드 시뮬레이션 시간 및 상태를 동적으로 보존하는 종합모드에서의 시뮬레이션 시간을 나타내었다. 리사이클모드의 시뮬레이션은 시뮬레이션에 필요한 상태를 보존한 후 실행한 것으로 새로운 상태 발생은 포함되지 않으므로,

모두가 상태전이로 시뮬레이션한 경우이다.

〈표 8〉에서 보는 바와 같이, 상태를 보존하지 않는 보통의 시뮬레이션(scratch(n))에 비해, 상태를 보존하여 재 이용하는 방식(nonshare(n)와 share(n))에 있어서 총 시뮬레이션 시간이 약 반으로 경감되었다. 따라서 시뮬레이션의 고속화에 대한 측면에서도 성과가 있었다. 〈표 7〉의 리사이클모드 시뮬레이션 시간에 비해 10%~20% 정도 시뮬레이션 시간이 많은 것은 동적으로 발생하는 상태를 보존하는 오버헤드이다.

〈표 7〉 리사이클 모드 시뮬레이션 시간
〈Table 7〉 Recycle mode simulation times

(단위: 초)

	matrix(4x4)	sort(9)	acker(2,3)
nonshare(4)	115.3	53.5	32.6
share(4)	112.1	52.2	31.2
nonshare(3)	139.4	62.0	38.9
share(3)	135.2	60.6	37.2
nonshare(2)	139.6	62.9	39.1
share(2)	135.4	60.9	37.5
nonshare(1)	197.7	89.6	52.5
share(1)	187.3	83.4	48.7

〈표 8〉 종합모드 시뮬레이션 시간
〈Table 8〉 Mix mode simulation times

(단위: 초)

	matrix(4x4)	sort(9)	acker(2,3)
scratch(4)	236.79	103.03	61.54
nonshare(4)	131.36	65.76	41.96
share(4)	128.81	63.82	40.63
scratch(3)	256.64	114.76	67.99
nonshare(3)	154.14	77.32	48.08
share(3)	147.55	73.28	46.70
scratch(2)	263.86	118.49	68.02
nonshare(2)	150.42	74.61	49.99
share(2)	146.01	71.48	46.05
scratch(1)	327.91	144.70	85.43
nonshare(1)	212.49	99.68	63.87
share(1)	200.83	96.89	59.62

7. 결론과 향후의 연구과제

VHDL언어를 바탕으로 하여, 하드웨어 시뮬레이터의 실행상태를 보존하면서 재 이용하는 계산방식을 설계하고, 실제의 처리시스템을 제작해 유효성을 확인하였다. 본 연구에서는 드라이버를 이용한 보통의 하드웨어 시뮬레이션 방법에 비하여 시뮬레이션의 고속화 면에서 최고 2배정도 성과가 있었으며, 이번 결과에 의해 하드웨어 시뮬레이션 분야의 고속화에 대한 가능성을 제시했다. 본 연구에서는 모듈의 크기를 발생하는 상태 수에 따라서 적절한 크기로 설정해야하는 부분이 중요하였다. 이 설정은 현재의 처리시스템에서는 자동화되어져 있지 않기 때문에, 이것을 자동화하는 알고리즘을 고안하여 구현할 필요가 있다. 본 논문에서 제안한 방식 - 한 번 평가한 결과를 동적으로 보존하여 재 이용함으로써 계산 양을 감소시키기 위한 계산방식 - 을 보편화하는 것도 흥미있는 과제라 생각한다. 또 분산병렬형 계산환경에서 이 계산 모델을 실현하기 위한 방식의 개발과 참조되어지지 않은 상태를 자동적으로 삭제하는 알고리즘, 모듈의 크기를 자동적으로 변경하는 알고리즘등 에대한 연구도 필요하다.

참 고 문 헌

[1] P.W. Tuinenga, SPICE "A Guide to circuit Simulation and Analysis using PSPICE," Prentice-Hall, 1988.

[2] Y. Chu, "Computer Organization and Micro programing," Prentice-Hall, 1972.

[3] H. Nakamura, "사상논리에 기초한 논리회로 검증시스템,"(in Japanese) 일본정보처리학회 논문지, 제30권 제6호, pp.771-778, 1989.

[4] H. Nakamura, etal "Architecture and Imple-mentation Description Language for Advan-ced Processor Design," Proc. of IEEE Asia-Pacific conference on circuit and system, pp. 213-218, 1992.

[5] K. Yasnura, "논리합성시대의 하드웨어 기술언어," 일본 정보처리학회지, 제33권 제11호, pp.1236-1243, 1992.

[6] S. Hosino 외, "UDL/I," 일본 정보처리학회지, 제33권 제11호, pp.1244-1249, 1992.

[7] K. Kohara 외, "SFL," 일본 정보처리학회지, 제33권 제11호, pp.1256-1262, 1992.

[8] H. Noike, "Verilog HDL," 일본 정보처리학회지, 제33권 제11호, pp.1263-1268, 1992.

외 K. Kamihara 외 "하드웨어 기술언어의 비교," 일본 정보처리학회지, 제33권 제11호, pp.1269-1283, 1992.

[10] Kozo Itano, 이필우, "논리회로 시뮬레이션의 고속화를 위한 상태전이 그래프의 Reduction,"(in Japanese) 일본 정보처리학회 논문지, 1997. (투고중)

[11] Kozo Itano, 이필우, "상태전이 그래프를 이용한 논리회로 시뮬레이터의 상태참조 패턴의 측정,"(in Japanese) 일본 정보처리학회 논문지, 1998. (투고중)

[12] Kozo Itano and PillWooLee, "Design of Logic Circuit Simulator Based on State Transition Graphs," 1998 International Technical Conf-erence on Circuits, system, computers and comunications, 1998 (to be published)

[13] 이필우, "상태전이그래프를 이용한 논리회로 시뮬레이터의 설계와실현," Tsukuba대학 전자정보공학계, Technical Note HLLA-492, 1997.

[14] VHDL Language Reference Manual, IEEE Std 1076-1987.

[15] J. R. Armstrong, "Chip-level Modeling with VHDL," Prentice-Hall, 1989.

[16] IEEE Standard 1076 VHDL Tutorial, CAD Language system, Inc., 1989.

이 필 우

1988년 동국대학교 전자공학과 (학사)

1991년 일본 Tsukuba대학 대학원 전자정보공학 석사과정(공학석사)

1997년 Tsukuba대학 대학원 전자정보공학 박사과정 수료

관심분야 : 프로그래밍 시스템, 컴퓨터구조, 하드웨어 시뮬레이터

Kozo Itano

1977년 일본 동경대학대학원 이학계연구과 물리학전문과정(이학박사)

1993년 Tsukuba대학 전자정보공학계 교수

관심분야 : 컴퓨터구조, 분산처리시스템, 프로그래밍시스템 등

※ 일본정보처리학회, 소프트웨어학회, 전자정보통신학회, IEEE, ACM회원