한국통신학회 1986년도 춘계학술발표회 논문집(1986.5.24)

86353

디지탈 하드웨어 시뮬레이터에서의 인공지능 기법 적용에 관한 연구

○
조 현 수
한국전기 통신공사 사업지원본부

김 재 희
연세 대학교

A Feasibility Study on Artificial Intelligence Approaches to the Design of Digital Hardware Simulators

Hyun Soo Cho
K.T.A Research Center

Jaihie Kim
Yonsei University

This paper describes an Artificial Intelligence (AI) approach to simulate the behaviors of designed digital hardware circuits at a gate level.

In this approach, Frames, a well known AI knowledge representation formalism, are used to represent the network connectivities of digital hardware equipments, and some procedures are attached to Frames to simulate the circuit behavior in a data-driven method. In this paper a new way of backward simulation method as well as the conventional forward simulation method are developed and compared.

1. 서론

컴퓨터를 포함한 디지탈 하드웨어의 구조와 기능이 복잡하여지고 집적회로소자의 집적도가 날로 증가함에 따라 디지탈 하드웨어의 설계, 분석 및 고장진단등을 위한 보조기기로써의 컴퓨터 이용이 연구되어 왔다. 〈1〉
연구되어온 기존의 통상적인 방법은 일련의 TABLE을 이용한 것으로 〈2〉 디지탈회로를 세밀하게 나타낼 수는 있으나, 표현된 내용을 이해하기 어렵고 회로설계자의 설계및 고장진단에 관한 지식을 SYSTEM에 쉽게 이식할 수가 없고 주어진 경우의 예상 업무만 처리 가능하며 차후 다른 업무로의 확장성이 결여되어 있다.
이러한 제반 문제점들을 해결하기 위하여, 본 연구는 인공지능의 지식표현 방법중의 하나인 프레임 〈FRAME〉을 이용한 디지탈 하드웨어 시뮬레이터의 설계에 대한 방법을 제시 하였다. 이 방법에서는 디지탈 소자들의 기능및 연결상태등 구조적이고 정적인 특성들은 프레임〈FRAME〉구조로 표현되었고, 신호들을 전파〈PROPAGATION〉 시킨다던지 설계규격 〈DESIGN CONSTRAINT〉의 만족을 검사하는등의 동적인 기능은 DEMON이라는 일종의 자료구동형 프로그램 〈DATA-DRIVEN FUNCTION〉을 FRAME에 포함시켜 표현하였다.
이와함께 본 논문은 지금까지의 INPUT에서부터 OUTPUT쪽으로 신호의 흐름을 따라 시뮬레이션하는 방법을 제시함은 물론, 이러한 순방향성〈FORWARD〉 시뮬레이터가 갖는 문제점들을 해결하기 위하여 필요한 OUTPUT에서 부터 역으로 시뮬레이션하는 역방향성〈BACKWARD〉시뮬레이터를 이용하는 방법도 제시하여 상호간의 장단점을 비교하였다.

2. 프레임 〈FRAME〉

FRAME은 1975년 MINSKY 〈3〉에 의해 제안되었으며, 많은 인공지능 시스템에서 하나의 지식표현 방법으로 쓰이고 있다. 프레임은 하나의 물체〈OBJECT〉,개념 〈CONCEPT〉, 혹은 사건〈EVENT〉등을 나타내며 그 FRAME이 나타내는 대상의 성질〈PROPERTY〉을 나타내는 슬럿〈SLOTS〉들과 경우에 따라서는 파싯〈FACETS〉, 그리고 그에 해당되는 값〈VALUES〉이 정보로 저장된다. 〈그림. 1〉
실제적으로는 많은 FRAME들이 모여 문제 분야의 지식을 나타내게 된다.

FRAME : a23		
(slot)	(facet)	(value)
ako	value	and-2
input	value	r31 35
output	value	a36 n3 r21 r12

〈그림. 1 프레임의 구조〉

모든 FRAME들은 자기위의 레벨, 즉 자기가 속해있는 집단을 나타내는 FRAME과 AKO〈A kind Of〉 슬럿을 통하여 연결되어 있는것이 보통이다. 여기서 위의 레벨의 FRAME을 제네릭〈GENERIC〉 FRAME이라 하고, 아래 레벨의 FRAME을 인스턴스〈INSTANCE〉 FRAME이라 한다. 모든 INSTANCE FRAME 들의 공통되는 성질들은 GENERIC FRAME에 모아 놓았다가 필요할 때 AKO SLOT을 통하여 꺼내볼 수 있으므로 모든 INSTANCE FRAME들에 중복되게 정보를 저장시킬 필요가 없어진다.
FRAME의 또 다른 특성중의 하나인 데몬〈DEMON〉은, 주어진 상황에 대처하는 행동을 하도록 SLOT에 저장되어진 〈ATTACHED〉 일종의 프로그램 루틴〈PROCEDURE〉이다. 임의의 SLOT에 새로이 정보가 입력되었을때 그 정보를 처리하는 DEMON을 IF-ADDED demon 〈그림. 2〉이라 하고, SLOT의 정보가 변화되었을때 그에 대처하는 DEMON을 IF-CHANGED demon이라 하는 등 데몬에는 여러가지 종류가 있을 수 있다.

FRAME : gate		
input	if-added	*check-input
output	if-added	*check-fanout
		*add-input
out-number	default	10
delay	default	10
state	if-changed	*selective-trace
	default	(-1 x)

〈그림. 2 제네릭 프레임〉

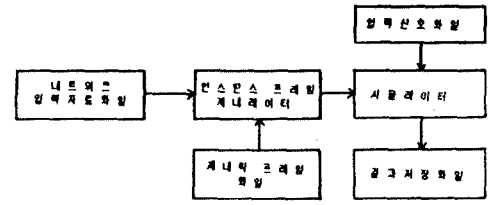
본 시스템은 위와 같이 여러가지 지식과 DATA들을 저장한 FRAME들과 이를 적절히 구동시키는 INTERPRETER로써 구성되어 있다.

3. FRAME 을 이용한 SIMULATOR의 실현

3-1. 시스템 개요 〈SYSTEM OVERVIEW〉

본 연구에서는 2 종류의 시뮬레이터를 제시 하였다. 첫째는 기존의 순방향성 〈FORWARD〉 시뮬레이터를 FRAME을 이용하여 실현시킨 것이다. 이는 모든 종류의 게이트와 프립프롭 〈FLIP-FLOP〉의 기능을 시뮬레이션 할 수 있으며, 모든 소자들에게 각기 다른 딜레이 타임 〈ASSIGNABLE DELAY TIME〉을 적용시킬 수 있다. 이는 또한 로직 〈LOGIC〉 시뮬레이터로써 '0' 과 '1' 그리고 'X' 〈UNKNOWN〉의 값을 가질 수 있으며 경우에 따라서는 그 이상도 가능하다. 또한 FLIP-FLOP 과의 FEED-BACK을 처리할 수 있도록 설계되었다.

또 하나의 시뮬레이터는 본 연구에서 처음으로 제시되는 BACKWARD 시뮬레이션을 위한 것으로, 모든 종류의 게이트들을 FORWARD의 경우와 같이 시뮬레이션 할 수 있으나 경우에 따라서는 처리속도가 단축된다. 이 경우에 각 소자의 딜레이는 없는 것으로 가정하였으나, 이의 확장도 가능할 것이다.
두개의 시뮬레이터는 서로 성격을 달리하지만, 시스템의 전반적 구조는 대동소이하다. SYSTEM의 전반적 구조를 나타내는 블럭 다이아그램 〈BLOCK DIAGRAM〉은 〈그림. 3〉과 같다.



〈그림. 3 블럭 다이아그램〉

3-2. 순방향 시뮬레이터

회로 설계자가 입력한 네트워크 입력자료 화일에서 각 소자들의 ELEMENT 타입을 읽어 〈 Fput 〉 함수로 그 소자의 AKO SLOT에 이를 기록한다.

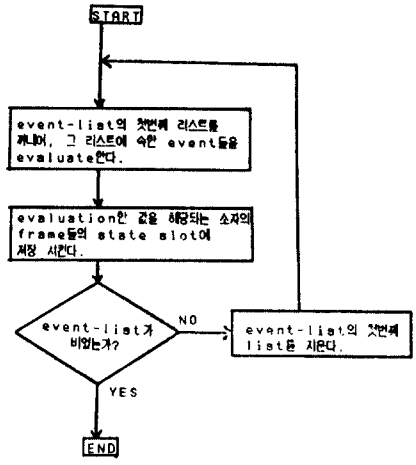
〈 Fput 〉 함수는 이때 각 소자의 프레임 구조를 구성한다. 이와함께 시스템인풋과 아웃풋 〈SYSTEM INPUT, OUTPUT〉의 프레임들도 만들어진다. 프레임의 골격이 만들어지고 나면 네트워크를 구성하게된다. 입력자료화일에는 각 소자의 출력에 연결되는 소자들만 표시되어 있다. 본 시스템은 이를 이용하여 각 프레임의 OUTPUT SLOT을 채워넣는다. 이때 OUTPUT SLOT에 하나의 소자가 채워지면, OUTPUT SLOT에 저장되어 있는 IF-ADEED 데몬, 즉 ADD-INPUT 이라는 데몬은 OUTPUT SLOT에 입력된 소자의 프레임을 찾아서 그 프레임의 INPUT SLOT에 원래의 소자를 써넣게 된다. 이와함께 각 SLOT에 DATA들이 쓰여질 때마다 그에 해당되는 데몬들이 동작하여 각종 설계규격들을 CHECK한다. 예로써 CHECK-INPUT

과 CHECK-FANOUT이라는 DEMON은 각각 ELEMENT에 연결되는 INPUT 개수와 FANOUT 개수를 CHECK 한다.
이와 같이 본 시스템은 자료를 프레임의 입력 슬롯에 넣어주기만 하면 데몬들이 자동적으로 작동되어 일을 처리하는 자료구동형 〈DATA-DRIVEN〉으로 구성되었다. 네트워크의 구성이 끝나고 나면 각 소자마다 〈그림. 4〉와 같은 형식의 프레임이 만들어지고, 시스템은 입력신호 확인을 읽어서 이벤트-리스트 〈EVENT-LIST〉에 변화되는 입력신호들을 등록하여, 시뮬레이션을 시작한다.

FRAME : Element-name		
ako	value	Ele-type
output	value	Fanout-nodes
input	value	Fanin-nodes
delay	value	Delay-time
state	value	Time & State
out-number	value	Number
in-number	value	Number

〈그림. 4〉 소자의 모델링

시뮬레이터의 전체 흐름도는 〈그림. 5〉과 같다.



〈그림. 5〉 시뮬레이터의 흐름도

시뮬레이터는 이벤트 리스트의 첫번째 리스트를 꺼내어 그 리스트에 속한 이벤트들을 EVALUATION 하고, 그 값을 해당되는 소자의 상태 〈STATE〉 SLOT에 집어 넣는다. 여기서 EVALUATION이란, 입력에 연결된 소자들의 상태를 알아내고, 이를 이용하여 자신의 출력값을 결정하는 것이다. 이때 시뮬레이터는 〈 Fput * 〉을 이용하게 되는데, 이 함수는 새로이 저장되어지는 출력값이 현재의 출력과 다르면 이를 상태 SLOT에 기록하는 동시에 IF-CHANGED 데몬을 동작시킨다. 만약 저장되어지는 값이 현재의 출력과 같다면, 이 소자는 신호의 상태가 변화되지 않았으므로, 이 소자에 연결된 FANOUT 소자들은 더이상 EVALUATION할 필요가 없다.

〈그림. 2〉의 게이팅 프레임 게이트의 상태〈STATE〉 슬롯에는 IF-CHANGED demon SELECTIVE-TRACE가 저장되어 있다. 이 DEMON은 현재의 프레임의 OUTPUT SLOT에 들어있는 소자들을 〈현재시간 + 딜레이〉 만큼의 시간에 해당되는 이벤트 리스트의 서브 리스트 〈SUB LIST〉에 등록시킨다. 이때 딜레이 값은 그 프레임의 딜레이 슬롯에서 찾아낸다.
위와 같은 과정을 모든 소자들에 대하여 되풀이하고, 이러한 과정이 끝나면 이벤트-리스트에서 제일 처음 리스트를 지운다. 이벤트-리스트가 비었는지를 조사하여 비었으면 시뮬레이션을 끝내고, 아니면 다시 첫번째 리스트에 위의 과정을 적용시킨다. 이 과정을 이벤트-리스트가 빌 때 까지 되풀이한다.

기존의 시뮬레이터가 타임-휠〈TIME-WHEEL〉 〈4〉을 이용하여 타임 스케줄링 〈TIME SCHEDULING〉을 한데 비하여, 본 시스템은 LISP의 리스트 구조를 이용하여, 이벤트-리스트를 구성하였다. 이는 처리가 끝난 소자가 들어있는 리스트를 지워버리고 LISP이 자동적으로 가베지 콜렉션 〈GARBAGE COLLECTION〉을 하므로 TIME-WHEEL의 개념이 필요 없어지게 된다.

3-3. 역방향성 시뮬레이터

역방향성 시뮬레이터는 앞서 제시한 FORWARD 시뮬레이터와는 달리 사용자가 결과를 보고자 원하는 소자에서 부터 시스템 인풋 쪽으로 역방향으로 EVALUATION을 한다.
각 소자들의 프레임 구조는 토직-레벨이라는 슬롯이 추가된 것을 제외하고는, 순방향 시뮬레이터에서와 같고 INTERPRETER에서만 차이가 난다.
토직-레벨이란 소자와 시스템 인풋과의 논리적 거리를 수치로 나타낸 것이다. 즉 제일 앞단의 소자는 토직-레벨의 값으로 '1'을 갖고 나머지 각 소자의 토직-레벨값은 입력소자들의 토직-레벨값 중에서 최대의 것에 '1'을 더한 값을 갖게 된다.

이러한 토직-레벨 슬롯도 네트워크를 구성할 때 IF-INCREMENT demon 인 카운트 토직레벨 〈COUNT LOGIC LEVEL〉이 동작하여 각 소자들의 토직-레벨을 계산하여 기록한다.
여기서 제시되는 시스템에서는 두가지 개념을 사용

하여 역방향성 시뮬레이터를 설계하였다.
한가지는 소자의 현상태를 알아내기 위하여 리커시브 〈RECURSIVE〉 함수를 설계한 것이고, 또 다른 한가지는 소자가 현재 어떤 시스템 인풋에 디펜던트 〈DEPENDENT〉하고 있는 지를 알아내서 이 인풋들이 변할 때만 시뮬레이션을 하도록 하는 것이다.

리커시브 함수는 역방향으로 현재 소자의 출력값을 찾아갈 때 다음과 같은 몇가지 휴리스틱 〈HEURISTIC〉 한 방법을 사용하였다.

〈1〉 결정적〈DOMINANT〉 인풋의 검사 :

결정적 인풋이란 〈그림. 6〉에서 보는 바와 같이 그 소자의 상태값이 다른 인풋의 상태값에 상관없이 현재의 소자의 상태값 출력을 결정하는 인풋을 뜻한다.

Gate	Dominant input	Resulting output
AND	LOW	LOW
NAND	LOW	HIGH
OR	HIGH	HIGH
NOR	HIGH	LOW

〈그림. 6〉 결정적 INPUT

이런 결정적 인풋이 여러 인풋들 중에 있을 경우에는 나머지 인풋들을 검사할 필요도 없이 그 소자의 상태가 결정 되므로 검사해야 될 경우의 수가 대폭 줄어들게 〈PRUNNING〉 된다.

〈2〉 HEURISTIC 함수의 설계 :

리커시브 함수는 일종의 검색 〈SEARCH〉 작업으로, 여러개의 입력들 중에서 어떤 입력을 먼저 검사하느냐를 결정하는 휴리스틱 함수 〈HEURISTIC FUNCTION〉를 실현하여 검색을 실행하였다.
이 휴리스틱 함수는 우선 어느 입력의 로직-레벨이 제일 낮은 가를 검사하고, 만약 로직-레벨이 같으면, 그 입력에 연결된 소자의 입력의 갯수가 적은 것을 선택한다.

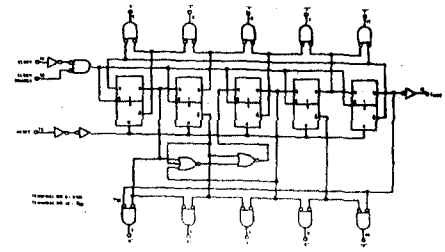
〈3〉 DEPENDENT SYSTEM INPUT의 검사 :

시스템이 시뮬레이션을 시작하면 상태를 알고자 하는 소자에 리커시브 함수를 적용시킨다.
리커시브 함수는 그 소자의 상태값을 알려주는 동시에, 현재 그 소자의 상태값 결정에 결정적 〈DOMINANT〉 역할을 한 시스템 인풋들을 그 소자의 디펜던트 SLOT에 기록한다. 그러면 시스템은 그 시스템 인풋들을 조사하여 그중 하나의 상태가 변할때 비로소 그 다음 시뮬레이션을 시작하게 된다. 그러므로 디펜던트 인풋들이 변하지 않는 한 새로이 시뮬레이션할 필요가 없게된다.

이러한 메카니즘은 시스템 인풋이 변할 때만 시뮬레이션을 하게되어 순방향 시뮬레이터의 선택-추적 〈SELECTIVE-TRACE〉과 같은 효과를 가져온다

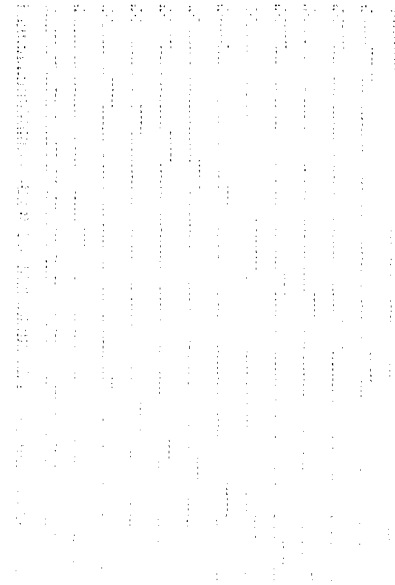
4. 실험 및 결과 고찰

연구에 대한 실험으로써 TEK 4404 AI MACHINE에 FRANTZ LISP을 이용하여 74HC4017 DECADE COUNTER/DEVIDER 〈그림. 7〉를 순방향으로 시뮬레이션하여



〈그림. 7〉 74HC4017 회로

〈그림. 8〉과 같은 정확한 결과를 얻을 수 있었으며



〈그림. 8〉 시뮬레이션 결과

프레임의 특성을 이용하여 회로의 내부상태를 찾아보거나 〈그림. 9〉 설계 잘못의 발견시 시스템이 현재의 상태를 어떻게 결정하였는지 그 과정을 볼 수 있었다.

〈그림. 10〉

```
-> (show 'ff1)

(ff1  (ako          (value      (FF)))
      (input        (k          (i13))
                    (cp         (n12))
                    (j          ((ff5 q*))))
      (output-q     (value      (ff2 j) (i4) (n13)))
      (output-q*    (value      (n1) (n2))))
```

〈그림. 9〉 SHOW의 사용 예

```
-> (how cn4 5000)

d4 >>  d3 >>  cn >>  << cn=h !!  r8 >>  a20 >>
s3 >>  << s3=1 << a20=1 !!  a19
>>  c3 >>  << c3=1 << a19=1 << r8=h !!  r6 >>
a15 >>  s3 >>  << s3=1 << a15=1 !!
  a14 >>  c2 >>  << c2=1 << a14=1 << r6=h !!
r4 >>  a10 >>  s3 >>  << s3=1 <<
a10=1 !!  a9 >>  c1 >>  << c1=h !!  s2 >>  <<
s2=1 << a9=1 << r4=h !!  r2 >>  a5
>>  s3 >>  << s3=1 << a5=1 !!  a4 >>  c0 >>
<< c0=1 << a4=1 << r2=h << d3=1 << d4=h
```

〈그림. 10〉 HOW의 사용 예

이와 함께 74HC181 4BIT A.L.U를 순방향과 역방향으로 각각 시뮬레이션 하였다.
순방향시에는 게이트의 EVALUATION 횟수가 543회였으나 역방향시에는 216회로 감소하여 PERFORMANCE가 대폭 향상됨을 확인 하였다. 〈5〉

5. 결론

본 연구에서는 기존의 시뮬레이터가 갖는 문제점을 해결하기 위하여 소자의 구조와 네트워크를 프레임 구조로 모델링 하였으며 시뮬레이터의 대부분의 기능을 데몬으로 만들어 DATA-DRIVEN으로 설계하였다.
이렇듯 프레임을 이용한 경우 아래와 같은 장점들을 확인할 수 있었다.

〈1〉 사용자가 언제라도 회로의 내부상태를 찾아볼 수 있다.
〈2〉 전문가의 지식을 데몬으로 만들어서 쉽게 이식시킬 수 있다.
〈3〉 회로의 여러가지 규격을 회로의 구성중이나 시뮬레이션 도중에 검사하여 잘못이 발견되면 사용자에게 알려준다.
〈4〉 프릭 프롭을 프레임으로 표현하여, 프릭 프롭간의 FEED-BACK 문제를 해결 하였다.
〈5〉 자료구동형 방식이기 때문에 구조가 간단하다.
〈6〉 사용자가 손쉽게 데몬과 함수들을 찾아볼 수 있으므로 런-타임 수정 〈RUN-TIME DEBUGGING〉이 가능하다.
〈7〉 역방향 시뮬레이션시에 설계 잘못을 초기에 발견할 수 있다.
〈8〉 설계 잘못의 발견시 시스템이 현재의 상태를 어떻게 결정하였는지 그 과정을 보여줄 수 있다.

시스템이 실용적으로 되기 위해서는 SIMULATION 하는 속도의 향상과 보다 조직적이고 효율적인 접근 방식이 필요할 것으로 사료되며, 그래픽 터미날 〈GRAPHIC TERMINAL〉이나 자연언어 〈NATURAL LANGUAGE〉를 이용한 보다 자연스러운 연결이 요구되고, 자동설계 시스템 〈AUTOMATIC DESIGN SYSTEMS〉과 같은 다른 종류의 CAD 시스템과의 유기적인 연결을 위한 연구가 계속되어야 할 것이다.

본 연구를 위하여 기재 사용에 협조하여 준 명 CORPORATION에 감사드린다.

6. 참고문헌

1. Douglas Lewin, Computer Aided Design of Digital Systems, Crane, Russak & Company, Inc., 1977.
2. S.A. Szygenda and E.W. Thompson, "Digital Logic Simulation in a Time-Based, Table-Driven Environment : Part 1. Design Verification", IEEE Computer, Vol. 8, No.3, Mar. 1975, pp. 24-36.
3. M. Minsky, "A Framework for Representing Knowledge", in P.H. Winston (Ed.), The Psychology of Computer Vision, N.Y: Mcgraw-Hill, 1975.
4. Ernst Ulrich, "Table Look Up Techniques for Fast and Flexible Digital Logic Simulation", 18th. Design Automation Conf. 1980.
5. 조현수, '디지탈 하드웨어 시뮬레이터에서의 인공지능 기법 적용에 관한 연구', 석사학위 논문, 연세대학교, 1986.