회로 분할법에 의한 정확한 논리 시뮬레이션

Accurate Logic Simulation Using Partitioning

오 상호, 조 동균, 강 성호
Sangho Oh, Dongkyoon Cho, Sungho Kang

한국시뮬레이션학회 논문지 : 제5권 제2호「별쇄」

회로 분할법에 의한 정확한 논리 시뮬레이션

Accurate Logic Simulation Using Partitioning

오 상호*, 조 동균*, 강 성호*
Sangho Oh, Dongkyoon Cho, Sungho Kang

요 약

회로의 크기가 점점 방대해지고 복잡해짐에 따라 설계검증을 위해 시뮬레이션은 매우 중요한 역할을 하고 있으며 빠른 속도와 정확성이 요구 되어지고있다. 좋은 시뮬레이터는 실제회로에서 출력되는 정확한 값을 예상할 수 있어야 하지만 3논리값 시뮬레이션에서는 X값 전파(unknown propagation)문제를 발생시켜 출력의 정확도를 떨어뜨리게 된다. 본 논문은 X값 전파 문제를 효과적으로 다루기 위해 분할기법을 사용했으며 분할의 깊이를 선택적으로 조절하는 효율적인 알고리즘을 개발하였고, 이를 토대로 미지값을 쉽고 빠르게 처리하는 시뮬레이터를 개발하였다. 그리고 벤치마크회로를 이용하여 새로 개발한 알고리즘과 시뮬레이터의 효율을 입증하였다.

Abstract

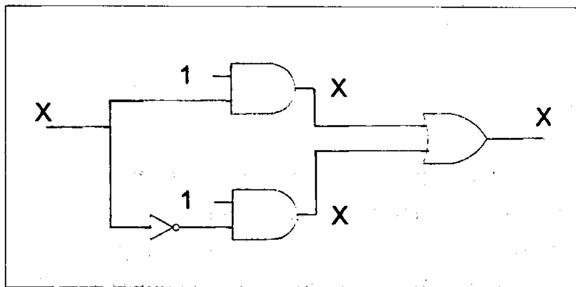
As circuits are larger and more complicated, logic simulation is playing a very important role in design verification. A good simulator should be fast and accurate, but unknown values in 3 value simulator may generate X-propagation problem which makes inaccurate output values. In this paper, a new partitioning method is devised to deal with X-propagation problem efficiently and an efficient algorithm is developed which is able to optimize time and accuracy by controlling partition depths. The results prove the effectiveness of the new simulation algorithm using some benchmark circuits.

* 연세대학교 공과대학 전기공학과

1. 서 론

논리 시뮬레이션(logic simulation)[1,2]은 회로를 설계하고 검증하는데 매우 중요한 역할을 하고 있으며 특히 회로의 크기가 점점 방대해지고 복잡해짐에 따라 설계과정에서의 시뮬레이션의 정확성과 속도는 점점 더 강조되고 있다. 좋은 회로 시뮬레이터는 실제회로에서 출력되는 정확한 값을 예상할 수 있어야 하지만 실제회로에서 발생하는 미지값(unknown value)은 X값 전파 (unknown propagation 또는 X propagation) 문제[3,4]를 발생시켜 정확한 결과를 얻어내는데 큰 장애로 작용하게 된다.

가장 기본적인 0과 1만을 사용한 2논리값 시뮬레이터는 실제 회로에서 발생하는 미지값을 처리할 수 없고 회로를 초기화 할 수 없다는 한계로 인하여 현재는 미지값을 포함하는 3논리값 시뮬레이션이 미지값을 처리하기 위해 널리 사용되고 있다. 지금까지 0과 1 그리고 미지값 X의 3논리값을 이용해 미지값을 처리하기 위한 다양한 방법들[5~10]이 고안되었고 설계되었지만 여기서 사용되는 미지값은 다음 〈그림 1〉과 같은 X값 전파 문제를 발생시켜 출력의 정확도를 떨어뜨리므로 정확한 시뮬레이션을 위해서는 이러한 미지값을 정확하고 빠르게 처리함으로써 X값 전파 문제를 효율적으로 해결하는 방법이 필요하다.



〈그림 1〉 X값 전파 문제

본 논문은 정확한 시뮬레이션을 위해 X값 전파 문제를 효과적으로 처리하기 위한 여러 방법 중 분할기법을 사용하였으며, 분할기법을 통하여 회로에서 발생하는 미지값을 정확하고 효율적으로 처리하였다. 2장에서는 미지값을 처리하는 여러 가지 방법들을 소개하고 3장에서는 분할기법을 이용한 시뮬레이션의 구현 방법과 본 시뮬레이터의 특징 그리고 4장에서는 분할기법을 적용해 실제 회로를 시뮬레이션한 결과를 실었으며 마지막으로 5장에 결론을 실었다.

2. 미지값의 처리 방법

미지값을 처리하는 3논리값 시뮬레이션의 가장 기본적인 방법으로 〈그림 2〉의 진리표를 이용하여 각 게이트 단위에서 미지값을 처리하는 방법[5,6]이 있다. 이 방법은 게이트 수준에서 시뮬레이션이 이루어지게 되므로 시간적인 효율성이 있지만 〈그림 1〉에서 보여주는 바와 같은 X값 전파 문제를 일으키므로 거의 미지값을 처리하지 못해 그 정확성에 명확한 한계가 있다[7].

AND 게이트

╲	0	1	X
0	0	0	0
1	0	1	X
X	0	X	X

OR 게이트

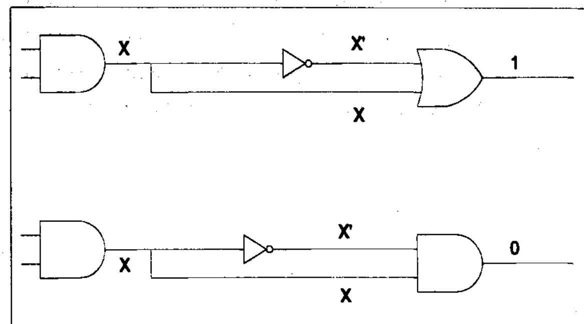
╲	0	1	X
0	0	1	X
1	1	1	1
X	X	1	X

XOR 게이트

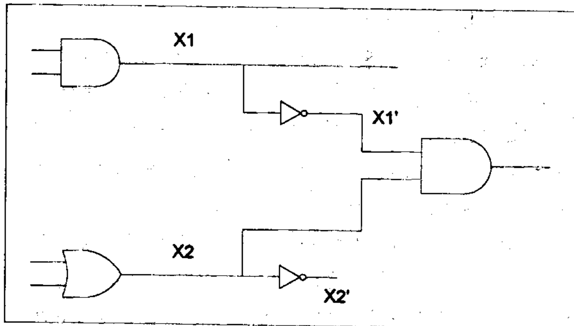
╲	0	1	X
0	0	1	X
1	1	0	X
X	X	X	X

〈그림 2〉 게이트에 대한 기본적인 3논리값 연산표

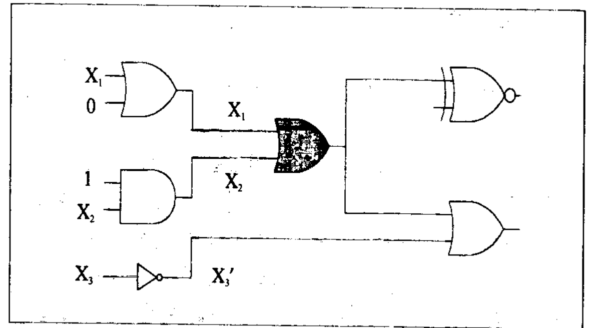
좀 더 개선된 3논리값 시뮬레이션 방법으로 미지값과 그 보수(complement)를 구분하는 방법[8]이 있는데, 이 방법은 〈그림 3〉과 같이 미지값과 그 미지값의 보수를 구분하여 미지값을 처리하는 방법으로 0, 1, X, X′의 4개의 논리값을 이용하여 어느 정도 미지값을 정확하게 처리할 수 있다. 하지만 그 적용범위가 한 개의 게이트에서 어떤 미



〈그림 3〉 X′를 이용한 시뮬레이션



〈그림 4〉 X'를 이용한 시뮬레이션의 오류



〈그림 5〉 출력값을 결정할 때 3개의 미지값이 고려되어야 하는 OR게이트

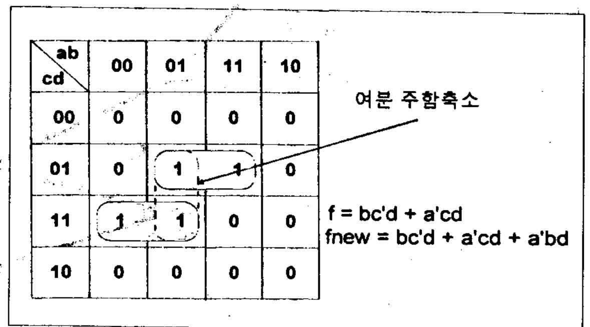
지값과 그 보수가 다시 만나는 경우로 적용 범위가 극히 제한되어있고 〈그림 4〉와 같이 서로 다른 미지값을 구분하지 못하므로 인해 잘못된 결과를 출력하게 되는 치명적인 단점이 있다[8].

〈그림 4〉에서의 경우와 같은 오류를 없애기 위한 방법으로 서로 다른 미지값을 구분하여 미지값 상호간의 간섭현상을 피하는 방법이 있는데, 모든 미지값에 각각 고유한 값(X_1, X_2, X_3, X_4 . . . . .)을 할당한 다음 $X_n+X_n'=1$, $X_n \times X_n'=0$ 과 같은 방식[8]으로 미지값을 처리함으로써 미지값간의 간섭에 의해 발생할 수 있는 〈그림 4〉의 경우와 같은 오류를 없앨 수 있다. 하지만 이 방법은 일단 미지값이 NOT 게이트가 아닌 다른 게이트를 지나고 나면 전혀 다른 미지값으로 인식된다는 점에서 그 정확성에 한계가 있으며, 회로가 방대해지고 단계(level)가 높아짐에 따라 처리할 수 있는 미지값의 수도 적어지고, 〈그림 5〉와 〈표 1〉에서 보여주듯이 미지값의 개수도 기하급수적으로 많아져 한 개의 게이트연산에서 고려되어져야하는 미지값의 개수도 커지므로 시뮬레이션하는데 소요되는 시간도 미지값의 개수의 제곱의 형태로 늘어나게 된다[9].

〈표 1〉 3개의 미지값을 갖는 다중값 시뮬레이션에서의 OR 게이트 진리표

OR	0	1	X_1	X_1'	X_2	X_2'	X_3	X_3'
0	0	1	X_1	X_1'	X_2	X_2'	X_3	X_3'
1	1	1	1	1	1	1	1	1
X_1	X_1	1	X_1	1	X_{n+1}	X_{n+2}	X_{n+3}	X_{n+4}
X_1'	X_1'	1	1	X_1'	X_{n+5}	X_{n+6}	X_{n+7}	X_{n+8}
X_2	X_2'	1	X_{n+9}	X_{n+10}	X_2	1	X_{n+11}	X_{n+12}
X_2'	X_2'	1	X_{n+13}	X_{n+14}	1	X_2'	X_{n+15}	X_{n+16}
X_3	X_3	1	X_{n+17}	X_{n+18}	X_{n+19}	X_{n+20}	X_3	1
X_3'	X_3'	1	X_{n+21}	X_{n+22}	X_{n+23}	X_{n+24}	1	X_3'

또 다른 방법으로 〈그림 6〉의 예에서 보여주는 것과 같이 회로에 대한 카르노 맵(Karnaugh map)을 작성한 다음 여분 주함축소(redundant prime implicant)를 찾아내어 추가함으로써 정확하게 미지값을 처리하는 방법[9]이 있지만, 이 방법으로 회로를 시뮬레이션하기 위해서는 회로에 대한 카르노 맵을 작성하고 작성한 카르노 맵에서 주함축소(prime implicant)와 여분 주함축소를 모두 찾아 회로를 시뮬레이션해야 하기 때문에, 이 방법으로 미지값을 정확하게 처리하는 논리 시뮬레이션은 결국 NP-complete 문제[10]를 갖고 있어 실제 회로에 적용하는데 어려움이 많다는 것이 밝혀져있다.



〈그림 6〉 여분 주함축소를 이용한 미지값의 처리

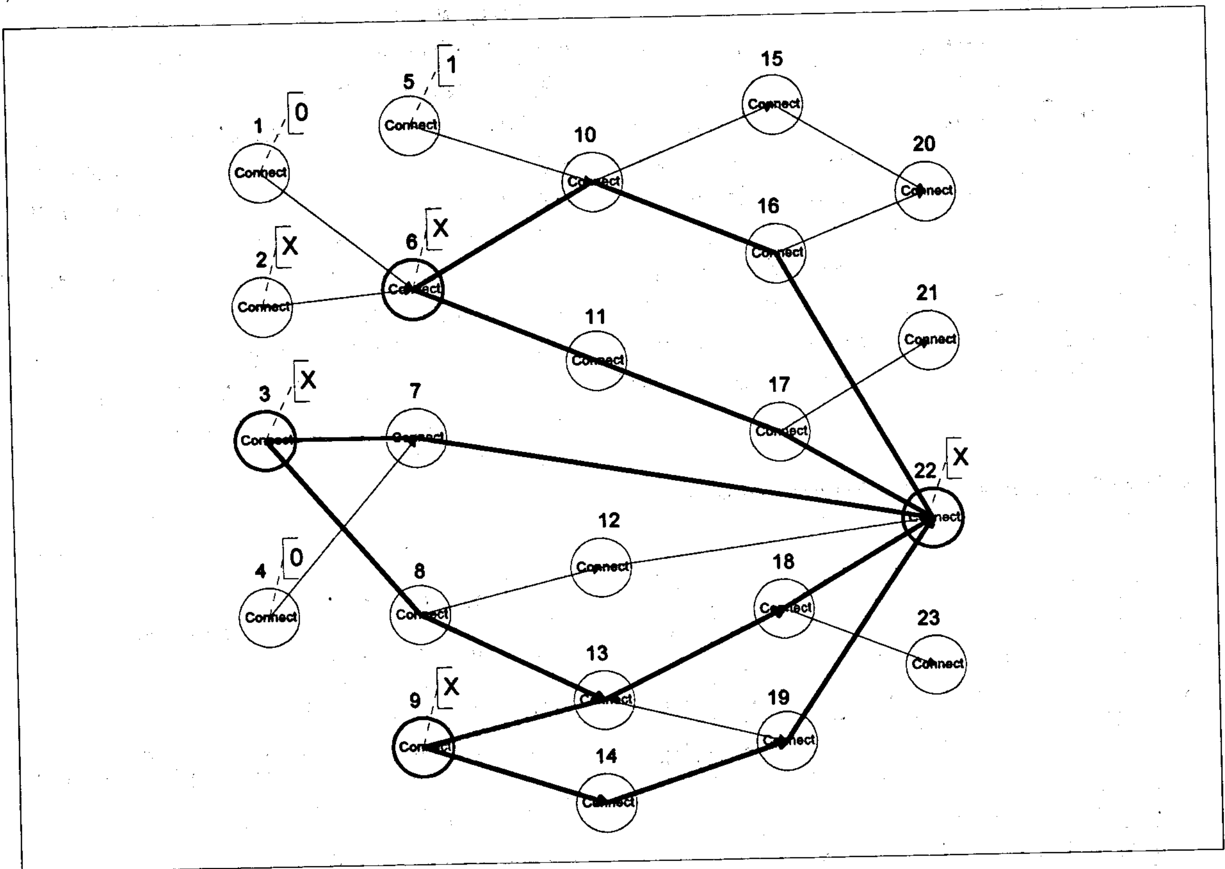
3. 분할기법

3.1 분할기법의 개요

2장에서 살펴본 바와 같이 미지값을 처리하는데는 여러 가지 방법이 있지만 많은 수의 게이트를 포함하는 회로를 분석해야하는 시뮬레이션에서 미지값을 정확하게 처리하기 위해서는 회로를 부분 부분으로 잘라서 분석하는 분할(partition) 방법이 유용하게 작용하는데, 이렇게 회로를 부분 부분으로 자르고 각 부분에 대한 정확한 시뮬레이션을 통해 전체 회로에서의 정확한 출력값을 구하는 시뮬레이션 방법이 분할기법이다.

다음의 〈그림 7〉은 어떤 회로에서의 분할을 보여주고 있다. 분할은 어떤 게이트의 출력의 개수가 둘 이상일 때 이루어지며, 갈라져 나간 출력이 다시 어느 한 게이트에서 모이게 될 때 분할은 끝나게 된다. 이와 같이 분할을 이룰 수 있는 부분은 많지만, 실제 시뮬레이션에서 의미가 있는 분할은 정확하게 처리할 수 있는 미지값이 발생하는 경우인 분할의 시작 게이트와 끝 게이트의 값이 미지값인 경우로 한정된다.

〈그림 7〉은 실제 회로에서 분할이 발생하는 경우를 예로 든 것으로 3개의 분할은 각각 3, 6번 그리고 9번 게이트에서 시작하며 22번 게이트에서 끝난다. 여기서 8번 게이트와 13번 게이트에서 시작하는 분할은 각각 3번과 9번 게이트에서 시작하는 분할들에 포함된다. 그리고 이 3개의 분할들을 포함하는 1개의 대분할이 존재하게 되는데, 여기서 각각의 소분할에 대해 시뮬레이션한 후 22번 게이트의 출력값을 구하는 소분할을 이용한 방법이 있을



〈그림 7〉 3개의 소분할과 이들을 포함하는 하나의 대분할

수 있고, 3개의 소분할을 하나의 대분할에 포함시켜 가능한 입력의 모든 경우를 대입해보고 22번 게이트의 출력값이 어느 한가지 값으로 고정되는지 살펴보는 대분할을 이용한 방법이 있을 수 있다. 이때 대분할을 이용하여 시뮬레이션하게되면 실제 시뮬레이션 결과에서 알 수 있듯이 n개의 소분할이 통합됐을 경우 2^n번의 시뮬레이션이 필요하므로, 실행시간이 지수적으로 증가하지만 정확도는 그렇게 많이 향상되지 않는 단점이 있다. 따라서 본 시뮬레이션에서는 소분할을 이용한 시뮬레이션 방법을 선택하였다.

먼저 일반모드에서 시뮬레이션을 실시한 후 그 결과가 미지값으로 나온 경우에 한해 분할 시뮬레이션 대상이 되며, 이와 같이 분할 부분에 대해서 정확한 값을 구하기 위해 수행하게되는 시뮬레이션이 부분분석 시뮬레이션이다. 부분분석 시뮬레이션은 가능한 모든 경우의 미지값을 대입해보고 분할의 출력이 한가지 값으로 고정되는지 살펴보는 것이다. 〈그림 7〉의 경우는 3개의 분할에 대해 각각 0과 1을 번갈아 대입해 보아야 하며, 따라서 총 6번(2×3)의 시뮬레이션이 필요하게 된다. 이렇게 해서 얻어진 22번 게이트의 입력값들을 종합해서 22번 게이트의 출력값을 결정하면 정확한 값을 얻을 수 있다.

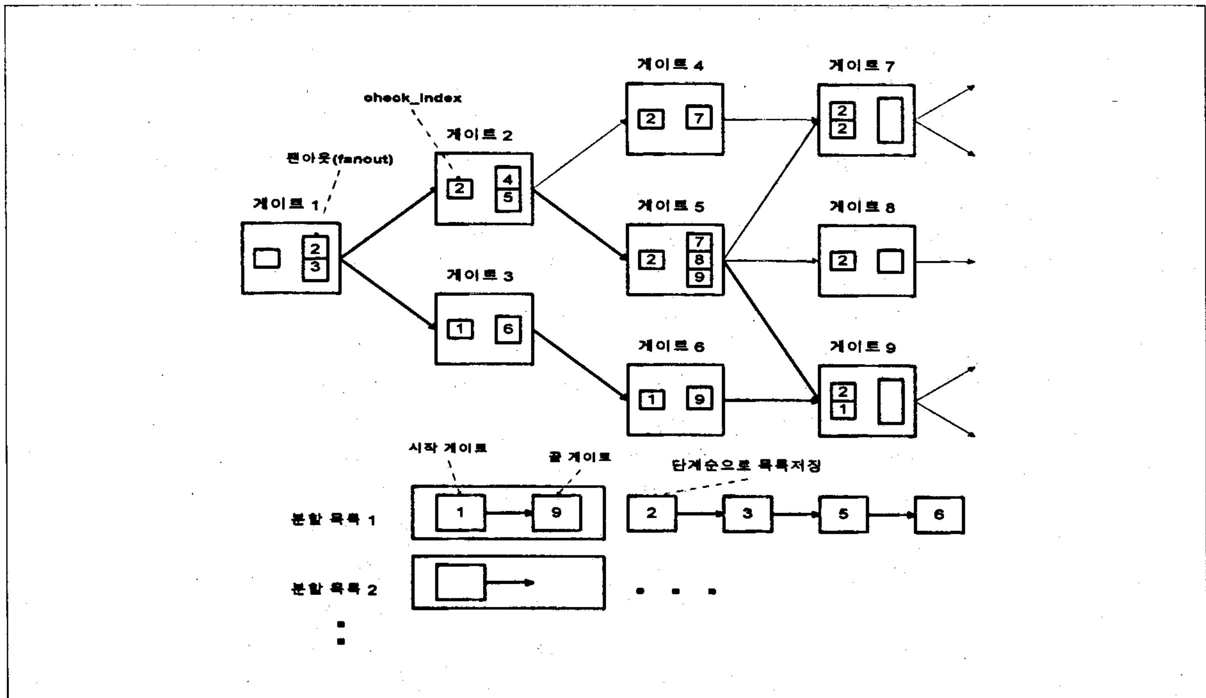
3.2 분할 시뮬레이션의 구현

본 분할모드 시뮬레이션 프로그램의 전체 알고리즘을 〈그림 8〉에 나타내었다. 우선 분할을 하기 위해서 먼저 전체 회로에 대해 단계화(levelization)를 실시한 다음 find_partition( ) 함수가 단계화 되어있는 게이트 목록에서 num_fanout이 2가 넘는 게이트에 대하여 분할이 존재하는지 검사한다. 분할 깊이 3인 분할을 찾는 간단한 예와 분할목록을 〈그림 9〉에 나타내었으며 find_partition( ) 알고리즘은 〈그림 10〉에 나타내었다.

분할 찾기 알고리즘을 살펴보면 먼저 팬아웃의 개수(num_fanout)가 2가 넘는 게이트에 대하여 재귀함수인 test_partition( )을 호출하여 분할이 존재하는지 검사하는데 test_partition( )은 일단 탐색 깊이를 일정하게 고정시켜 트리탐색이 너무 깊어 비효율적으로 되는 것을 방지한 다음, 자신을 재귀 호출하여 깊이우선탐색(depth first search)을 실시한다. 그 다음 check_index를 팬아웃의 개수로 초기화시킨 후 첫 번째 팬아웃(fanout)에 대한 탐색에서는 check_index를 저장하면서 일정한 깊이까지 탐색하고, 두 번째 팬아웃에 대한 탐색부터는 check_index를 1씩 감소시켜 탐색을 수행하면서 만일 탐색 도중 게이트에 저장된 값이 check_index보다 크면 분할이 발행한 것으로 보고 분할 목록(partition_list)에 분할이 이루어진 게이트의 목록(list)을 단계순으로 추가시킨다. 이때 check_index가 저장된 값과 같은 경우는 〈그림 9〉의 게이트 2에서 시작하고 게이트7에서 끝나는 분할과 같이 분할 트리(tree)의 중간에 분할이 존재하는 경우로서, 일단 단계화 되어있는 게이트 목록에 따라 모든 게이트에 대하여 분할 검사를 실시하기 때문에 다음에 다시 확인할 수 있는 기회가 생기므로 일단 무시한다.

```
main( )                                    // partition-mode simulation
{
        parse( );                          // parse the benchmark circuit
        levelize( );                       // levelize the circuit
        find_partition( );                 // find partitions in the circuit
        while (get_input_value( ))
        {       simulate( );               // normal-mode simulation
                simulate_partition( );     // partition-method simulation
                simulate( );               // accurate results
        }
}
```

〈그림 8〉 분할모드 시뮬레이션의 흐름



〈그림 9〉 분할 깊이 3인 분할 찾기와 분할 목록의 예

```
ALGORITHM : find_partition( )
(1) temp <- head of simulation linked list;              //initialize
(2) if (gat[temp->index].num_fanout > 2)
      (2.1) test_partition( );
(3) temp = temp->next;                                    //set temp as next list
(4) if (temp != NULL) goto (2)
(5) return;

ALGORITHM :test_partition( )
(1) if (gat[index].value > check_index)
      (1.1) add_partition_List(index);                    //store found partition
      (1.2) return;
(2) if (gat[index].value == check_index)                  //ignore this case
      (2.1) return;
(3) if (gat[index].value < check_index)
      (3.1) gat[index].value == check_index;              //set value as check_index
(4) if (depth_limit > 0)
      (4.1) depth_limit--;                                //decrease depth_limit
      (4.2) test_partition( );                            //call recursive function
(5) return;
```

〈그림 10〉 find_partition( ) 알고리즘

```
ALGORITHM : simulate_partition( )                          //partition-method simulation
(1) temp <- head of partition linked list;                         //initialize
(2) if (both gat[temp->end].value and gat[temp->start].value are DONOT_CARE's)
        (2.1) suspected++;                                         //increase suspected
        (2.2) p_simulate(temp);                            //partition-part simulation
(3) temp = temp->next;                                             //set temp as next list
(4) if (temp != NULL) goto (2)
(5) return;

ALGORITHM : p_simulate(PLIST *temp)                        //partition-part simulation
(1) p = temp->p_simList;
(2) gat[temp->start].value = 0;                                    //initialize gate.value
(3) value_0 = operate(p);                                  //simulate only suspected partition
(4) gat[temp->start].value = 1;
(5) value_1 = operate(p);
(6) if (both value_0 and value_1 are 0's or 1's)                   //if outputs are fixed
        (6.1) detected++;                                          //increase detected
        (6.2) gat[temp->end].flag = value_0;                       //save the fixed value
(7) return;
```

〈그림 11〉 simulate_partition ( ) 알고리즘

그 다음 우선 분할 시뮬레이션에 들어가기 전에 simulate ( )로 일반모드 시뮬레이션을 실시하여 미지값이 발생하는 게이트를 조사하고, 분할 시뮬레이션 함수 simulate_partition( )으로 분할 목록을 참고하여 분할의 끝 게이트의 값이 미지값인 경우에 한하여, 시작 게이트의 값이 미지값인지 확인하여 분할의 시작 게이트와 끝 게이트의 값이 모두 미지값이면 부분분석 시뮬레이션 함수 p_simulate ( )를 호출하여 미지값을 정확하게 처리한다. 부분분석 시뮬레이션 함수는 시작 게이트의 값이 미지값이면 미지값 대신 0과 1을 대입하여 끝 게이트의 값이 변화하는지를 확인하는데, 끝 게이트의 값이 변화하지 않으면 이 분할의 값은 고정된 것으로 보고 끝 게이트의 값을 고정시킨다. 이렇게 각 분할 부분에 대하여 부분분석 시뮬레이션을 통해 정확한 값을 구하여 고정시켜 놓은 후, 마지막으로 정확하게 처리된 미지값을 가지고 일반모드의 3논리값 시뮬레이션을 하면 미지값들을 정확하게 처리할 수 있다.

분할모드 시뮬레이션을 실행하는데 있어서 분할의 깊이를 적절하게 설정하는 문제가 있는데, 깊이를 깊게 설정한다면 시뮬레이션의 정확도는 증가하지만 실행속도가 느려지고 메모리 소모가 많아진다. 따라서 사용 시스템의 성능과 사용자에 따라 깊이를 조절 할 수 있도록 하였다.

4. 분할 방법의 적용 결과

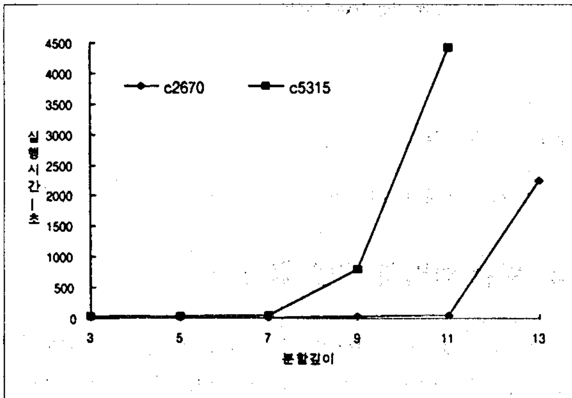
본 시뮬레이터는 800줄 정도의 C언어로 작성되었으며 ISCAS 85 벤치마크 회로[11]를 대상으로 100Hz Pentium 칩에서 시뮬레이션하였고 메모리는 600KB로 제한하여 분할 깊이와 소요 메모리 크기의 상관 관계를 밝혔다.

먼저 대분할 기법을 이용한 시뮬레이션과 소분할 기법을 이용한 시뮬레이션의 실행시간을 비교해보았는데 입력값의 3분의 1이 미지값을 포함하도록 임의로 선정된 1000개의 입력패턴을 입력하였다. 〈표 2〉와 〈그림 12〉를 통해서 알 수 있듯이 대분할 기법은 분할의 깊이가 깊어져 많은 소분할들이 병합됨에 따라 분할의 입력 개수를 n이라고 할 때 시뮬레이션 시간이 2^n의 형태로 증가하는 반면, 정확도는 그다지 많이 향상되지 않음을 알 수 있다. 이는 입력단에 미지수의 값이 증가함에 따라 그만큼 어떤 특정한 미지값이 정확하게 처리될 확률이 2^n의 형태로 작아지기 때문으로, 대분할 기법이 결국 NP-Complete 문제를 갖고있음을 나타내는 것이다. 따라서 본 분할기법 시뮬레이

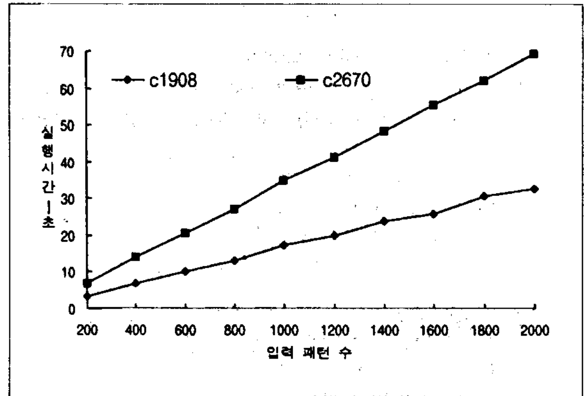
션에서는 시뮬레이션 시간이 2×n의 형태로 증가하는 소분할 기법을 사용하였다.

〈표 2〉 대분할 기법과 소분할 기법의 비교

분할 깊이	c2670 실행시간 [초] 소분할	대분할	c2670 처리된 미지값 소분할	대분할	c5315 실행시간 [초] 소분할	대분할	c5315 처리된 미지값 소분할	대분할
3	14.72	14.78	396	400	29.39	30.82	1,418	1,473
5	15.38	17.03	673	701	34.06	38.17	2,150	2,019
7	21.42	22.47	1,013	1,027	42.63	52.25	2,642	2,892
9	26.97	28.07	1,214	1,235	55.43	808.28	2,893	2,956
11	31.48	49.23	1,591	1,634	61.04	4,428.64	3,049	3,160
13	36.59	2,264.43	1,602	1,651	N/A	N/A	N/A	N/A



〈그림 12〉 대분할 시뮬레이션의 실행시간



〈그림 13〉 입력 패턴수와 실행시간의 관계

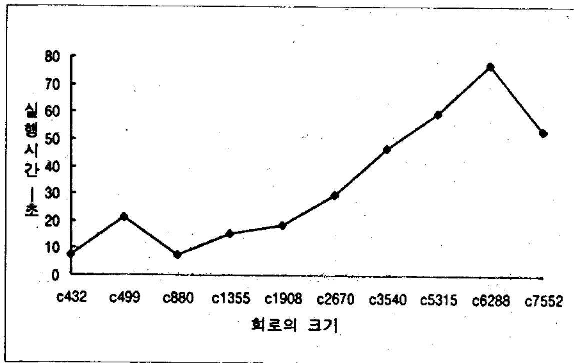
〈표 3〉 일반모드와 분할모드의 실행시간 비교

회 로	회로 입력수	분할 깊이	일반모드 [초]	분할모드 [초]	분할/일반	처리된 미지값
c432	36	10	1.21	7.25	5.99	126
c499	41	10	1.42	21.37	15.05	169
c880	60	10	2.09	7.41	3.54	279
c1355	41	10	2.75	15.05	5.47	163
c1908	33	10	3.46	18.68	5.40	634
c2670	233	10	5.98	29.78	4.98	1,406
c3540	50	8	6.42	46.70	7.27	374
c5315	178	10	10.22	59.78	5.85	2,941
c6288	32	10	10.01	77.40	7.74	377
c7552	207	5	14.12	53.41	3.78	1,342

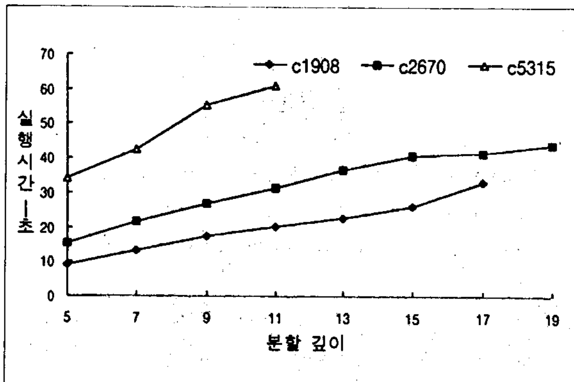
시뮬레이션 입력 패턴수와 실행시간의 관계를 살펴보기 위해 입력패턴을 200개에서 2000개까지 200개씩 증가시키면서 측정한 분할모드의 실행시간을 〈그림 13〉의 그래프로 나타내었다. 그래프에서 보여주듯이 입력 패턴수에 대해 실행시간은 거의 선형적으로 증가하므로 시뮬레이션 회수는 입력 패턴수와 거의 비례한다고 할 수 있다는 결론을 얻을 수 있다. 33개의 입력을 갖는 c1908과 233개의 입력을 갖는 c2670을 시뮬레이션했으며 미지값이 총 입력의 3분의 1이 되도록 하였다.

다음 〈표 3〉은 단순한 3논리값 시뮬레이션인 일반모드의 시뮬레이션과 분할모드 시뮬레이션과의 실행시간의 차이와 정확하게 처리되는 미지값의 개수를 서로 비교한 것이다. 입력값의 3분의 1이 미지값을 포함하도록 임의로 선정된 1000개의 입력패턴이 입력되었는데, c499를 제외하면 대체적으로 분할모드 시뮬레이션 방법이 단순한 일반모드 시뮬레이션보다 4~8배의 시간이 걸리며, 회로가 커짐에 따라 일반모드 시뮬레이션의 실행시간이 증가하고, 분할기법의 시뮬레이션의 실행시간도 따라서 증가한다는 것을 알 수 있다. 그리고 〈그림 14〉를 통해서 분할기법의 실행시간이 회로의 크기에 따라 지수적으로 증가하지 않고 선형적으로 증가함을 알 수 있다.

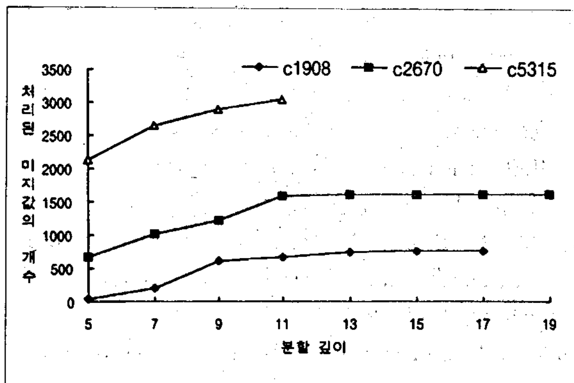
〈그림 15〉와 〈그림 16〉은 분할깊이에 따른 실행시간의



〈그림 14〉 회로의 크기와 실행시간의 관계



〈그림 15〉 분할깊이와 실행시간의 관계



〈그림 16〉 분할 깊이와 처리된 미지값 개수

지연과 처리되는 미지값의 개수를 비교한 것으로 c1908 회로와 c2670 그리고 c5315 회로를 비교의 대상으로 하였는데 〈그림 15〉에서 분할 깊이가 깊어짐에 따라 실행시간이 선형적으로 증가함을 볼 수 있다. 미지값의 개수는 〈그림 16〉을 통해서 알 수 있듯이 분할의 깊이가 깊어짐에 따라서 선형적으로 증가한다고 할 수 없는데, 이는 c1908 회로의 경우에는 분할 깊이 13에서 그리고 c2670 회로의 경우에는 분할깊이 11에서 이미 정확하게 처리 할 수 있는 미지값의 거의 대부분이 정확하게 처리되었기 때문에 더 이상 정확한 값을 구할 수 있는 미지값이 존재하지 않기 때문이다. 따라서 분할모드 시뮬레이션에서는 적절하게 설정된 분할 깊이가 불필요한 시뮬레이션 시간을 줄일 수 있으므로 분할 깊이의 적절한 선택이 시뮬레이션의 효율에 중요한 영향을 미친다는 것을 알 수 있다

c1908 회로에서는 분할 깊이 19에서 그리고 c5315 회로는 분할 깊이 13에서 600KB의 메모리로는 더 이상 분할 목록을 저장할 수 없었는데 이는 c1908회로가 c2670회로보다 더 많은 분할을 가졌기 때문으로, 회로의 크기뿐만 아니라 회로의 특성도 메모리의 사용량에 많은 영향을 준다는 것을 알 수 있다.

〈표 4〉 미지값의 비율과 실행시간의 관계

(단위 : 초)

회로 / 미지값 비율	c2670		c5315		c6288	
	일반모드	분할모드	일반모드	분할모드	일반모드	분할모드
1/3	5.82	27.91	10.38	58.24	10.32	68.35
1/7	5.27	19.01	10.16	40.05	10.16	54.89
1/11	5.21	15.76	10.05	33.02	10.16	48.62
1/15	5.21	13.46	10.05	29.06	10.10	41.64
1/19	5.21	12.19	10.01	27.08	10.05	40.43

〈표 4〉는 입력되는 미지값의 개수와 실행시간의 상관관계를 밝힌 것으로 입력되는 미지값의 수가 적어짐에 따라 일반모드의 시뮬레이션 시간은 거의 변함이 없지만 분할모드 시뮬레이션의 시간은 상당히 줄어듦을 알 수 있다. 이는 적은 미지값을 갖는 시뮬레이션에서 분할모드 실행시간이 줄어드는 본 시뮬레이터의 특성을 말해주고 있는데, 본 시뮬레이터는 각각의 입력에 대하여 미지값이 출력된 경우에만 분할 시뮬레이션을 하며 나머지는 일반모드로 빠르게 시뮬레이션하므로 적은 미지값을 갖는 시뮬

레이션에서 상대적으로 빠르게 동작하는 특징이 있다. 예를 들어 T_n을 일반모드의 시뮬레이션 시간이라고 하고 T_p를 분할모드의 시뮬레이션 시간이라고 한다면 P번의 시뮬레이션 중 p번의 경우에 한하여 미지값이 출력되었을 때 P>>p 인 경우에 본 시뮬레이터의 분할모드 시뮬레이션 시간 $T_{new} = P \times T_n + p \times T_p$ 은 전체를 분할기법에 의해 시뮬레이션한 시간 $T_{old} = P \times T_p$ 보다 훨씬 적게 된다.

〈표 5〉 분할기법과 논문[9]의 비교

회로	실행시간 [초]		처리된 미지값	
	분할기법	논문 [9]	분할기법	논문 [9]
c1908	3.87	2,447.64	337	376
c2670	5.47	3,823.91	257	321
c6288	13.21	7,764.87	568	579

마지막으로 논문[9]와 본 논문의 시뮬레이션 방법을 〈표 5〉를 통해서 비교해 보았다. 3분의 1이 미지값으로 설정된 200개의 패턴을 입력하였으며, 본 논문의 시뮬레이션 깊이는 10으로 설정하여 논문[9]의 시뮬레이션 방법과의 실행시간 차이와 처리되는 미지값의 차이를 비교하였다. 본 논문의 방법은 논문[9]의 방법에 비해서 정확도는 다소 떨어지지만 실행시간이 매우 짧음을 알 수 있는데, 이는 본 분할기법이 소분할 기법을 사용하였고 앞서 살펴본 바와 같이 각각의 입력에 대하여 미지값이 존재하는 경우에만 분할 시뮬레이션 하기 때문으로 게이트의 수가 많고 분할이 많이 발생하는 회로에서 효과적으로 사용될 수 있음을 보여주고 있다.

5. 결론

단순히 0과 1외에 미지값 X를 추가한 3논리값을 이용하는 시뮬레이션은 미지값의 X값 전파 문제로 인하여 실제 시스템의 출력값과 시뮬레이션의 결과와의 사이에 많은 차이를 보일 수 있다. 그러므로 적절한 미지값의 처리 방법을 통하여 이 문제를 극복하고, 보다 나은 정확성을 보장해주는 시뮬레이터를 개발하는 것이 본 논문의 목적이었고 그러한 결과로 본 분할기법 시뮬레이션 프로그램을 만들었다. 어떤 시뮬레이터이든지 그 정확성과 속도 그리고 메모리에는 반비례의 상관 관계가 존재하며 어떠한 방법으로 시간과 메모리 그리고 정확도를 적절하게 설정하느냐 하는 것은 좋은 시뮬레이터를 결정하는데 중요한 요소로 작용한다. 본 시뮬레이터는 우선 메모리를 600KB로 제한하여 효율적인 메모리 사용과 분할 깊이의 적절한 절충을 고려했으며, 분할 깊이를 선택함으로서 정확도와 실행시간을 최적의 상태로 조절할 수 있도록 하였다. 그리고 본 시뮬레이터의 분할모드 시뮬레이션 특징으로 첫째 소분할 기법을 이용하여 정확도를 어느정도 유지하면서 실행시간을 대폭 감소시켰다는 점과, 둘째 각각의 입력에 대하여 미지값이 출력된 경우에만 분할 시뮬레이션을 하고 나머지는 일반모드로 빠르게 시뮬레이션함으로써 적은 미지값을 갖는 시뮬레이션에서 상대적으로 빠르게 동작하면서 절약되는 시간 동안 분할 깊이를 깊게 함으로써 그 정확도를 더욱 향상시킬 수 있다는 점을 들 수 있다.

참고문헌

[1] S. Szygenda, "TEGAS2－Anatomy of a General Purpose Test Generation and Simulation System for Digital Logic," Proc. of Design Automation Conference, pp. 116-127, 1972.

[2] S. Szygenda and E. W. Thompson, "Modeling and Digital Simulation for Design Verification and Diagnosis," IEEE Transactions on Computers, vol. 25, no. 12, pp. 1242-1253, Dec, 1976.

[3] A. Miczo, "Digital Logic Testing and Simulation," Harper & Row, 1986.

[4] E. Horbst, "Logic Design and Simulation," North-Holland, 1986.

[5] Y. H. Jea and S. A. Szygenda, "Mapping and Algorithms for Gate Modeling on a Digital Simulation Environment," IEEE Transactions on Circuits and Systems, vol. 26, no. 5, pp. 304-315, May 1979

[6] M. A. Breuer, "A Note on Three-Valued Logic Simulation," IEEE Transactions on Computers, pp. 399-402, April, 1972.

[7] M. A. Breuer and A. D. Friedman, "Diagnosis & Reliable Design of Digital Systems," Computer Science Press, 1976.

[8] M. Abramovici, M. A. Breuer and A. D. Friedman, "Digital System Testing & Testable Design," IEEE Press, pp. 43-46, 1990.

[9] S. J. Chandra and J. H. Patel, "Accurate Logic Simulation in The Presence of Unknowns," IEEE Proc. of International Conference on Computer-Aided Design, pp. 34-37, 1989.

[10] H. P. Chang and J. A. Abraham, "The Complexity of Accurate Logic Simulation," Proc. International Conference on Computer-Aided Design, 1987.

[11] F. Brglez and H. Fujiwara, "A Neutral Netlist of 10 Combinational Benchmark Circuit and a Target Translation in FORTRAN," Proc. of International Symposium on circuits and Systems, pp. 695-698, 1985.

● 저자소개 ●

오상호

1997년 8월 연세대학교 전기공학과 졸업예정(공학사)
현 재 연세대학교 전기공학과 재학중
관심분야 : 알고리즘, 데이터처리, DFT.

조동균

1997년 3월 연세대학교 전기공학과 졸업예정(공학사)
현 재 연세대학교 전기공학과 재학중
관심분야 : 데이터처리, 유・무선통신.

강성호

1986년 2월 서울대학교 제어계측공학과 졸업(공학사)
1988년 5월 The Univ. of Texas at Austin
Electrical and Computer Eng. 졸업(공학 석사)
1992년 5월 The Univ. of Texas at Austin
Electrical and Computer Eng. 졸업(공학 박사)
1989년 11월～1992년 8월 미국 Schlumberger Inc. Research Scientist
1992년 9월～1992년 10월 미국 The Univ. of Texas at Austin Post Doctoral Fellow
1992년 8월～1994년 10월 미국 Motorola Inc. Senior Staff Engineer
1994년 9월～현재 연세대학교 전기공학과. 조교수
관심분야 : CAD, 테스팅, ASIC설계