論 文
50D-1-7

사이클 기반 논리시뮬레이션 가속화 기법 연구

Acceleration Techniques for Cycle-Based Logic Simulation

朴 英 鎬* · 朴 恩 世**
(Young-Ho Park · Eun-Sei Park)

Abstract - With increasing complexity of digital logic circuits, fast and accurate verification of functional behaviour becomes most critical bottleneck in meeting time-to-market requirement. This paper presents several techniques for accelerating a cycle-based logic simulation. The acceleration techniques include parallel pattern logic evaluation, circuit size reduction, and the partition of feedback loops in sequential circuits. Among all, the circuit size reduction plays a critical role in maximizing logic simulation speedup by reducing 50% of entire circuit nodes on the average. These techniques are incorporated into a levelized table-driven logic simulation system rather than a compiled-code simulation algorithm. Finally, experimental results are given to demonstrate the effectiveness of the proposed acceleration techniques. Experimental results show more than 27 times performance improvement over single pattern levelized logic simulation.

Key Words : 사이클 시뮬레이션, 회로분할, 병렬패턴, 게이트 감축기법, 시뮬레이션 가속화

1. 서 론

종래의 전자회로의 동작 검증은 타이밍과 논리를 동시에 검증하는 사건 구동식 (event-driven) 시뮬레이션 방법을 사용하여 왔으나[1] 수백만 게이트급의 고집적 회로에서는 복잡한 타이밍 모델을 사용할 경우 전체 시뮬레이션 시간이 지나치게 많이 걸리는 단점이 있었다. 따라서, 최근의 동향은 회로의 타이밍과 논리를 분리하여 따로 검증하는 방법이 적용되고 있다. 즉, 타이밍 검증은 정적 타이밍 검증기 (static timing analysis)를 사용하고 논리검증으로는 사이클 기반 시뮬레이션 기법을 많이 사용하고 있다. 사이클기반 시뮬레이션은 내부 조합 논리 소자의 출력값을 연산할 때 각 논리 소자 및 연결 신호의 지연시간에 의한 변화를 고려하지 않고 클럭 신호가 활성화된 후의 플립플롭의 출력 및 주 입력 신호값들에 의한 정적 상태에서의 논리값 만을 고려함으로써 시뮬레이션 시간의 획기적인 단축을 얻게 된다. 이러한 방법은 조합 회로나 동기 회로에 한정되어 적용할 수 있다는 단점이 있지만, 최근의 설계동향이 거의 동기 회로이며 논리검증이 정적 타이밍 검증과 분리되어 수행되는 점을 감안할 때 앞으로도 많이 사용될 것으로 예상된다.

지금까지 연구되어 온 사이클 기반 시뮬레이션 알고리즘으로는 LCC (Levelized Compiled Code), [2], 병렬패턴 시뮬레이션 [3], Event-driven LCC [4] 및 BDD-based (Binary Decision Diagram) 시뮬레이션 [5,6] 등이 있다. 일반적으로 사이클 기반 시뮬레이션에서는 논리소자의 연산을 컴퓨터 언어를 사용하여 기술하고 이를 컴파일하여 사용하는 compiledcode 방식 [2,4]이 주로 사용되고 있으나, 본 논문에서는 논리소자의 연산은 기존의 논리표를 사용하는 table-driven 시뮬레이션 기법을 사용하였다. 그러나, 입력값의 변화에 의한 사건의 처리는 지연시간에 의한 사건 발생순서가 아닌 회로의 레벨에 따라 하나의 사이클에서 단 한번씩만을 수행하는 levelized simulation을 사용하였으며 이는 기존의 사건구동식 방식과 다른 사이클 기반 시뮬레이션의 특징을 나타낸다.

본 논문에서는 시뮬레이션의 속도를 향상시킬 수 있는 여러 가지 기법에 대해서 기술한다. 병렬 패턴을 이용한 논리연산 시뮬레이션을 기반으로 하여 게이트 수 감축, 게이트 타입 세분화와 함수 포인터 등의 기본적인 가속화 기법을 고안하였으며 또한, 순차회로에 대한 병렬패턴 시뮬레이션에서 속도향상에 가장 걸림돌이 될 수 있는 피드백 루프를 효과적으로 처리하기 위하여 세분화된 분할 기법을 개발하였다. 제안된 방식에 대한 ISCAS85 조합회로 [8]에 대한 실험결과 종래의 단일패턴 시뮬레이션에 비하여 최고 27배의 속도 향상을 얻을 수 있었다. 또한, 조합회로와 달리 순차회로의 경우, 피드백에 의한 상태 의존 (state dependency)이 있어 현재의 입력값만으로 회로의 출력 및 플립플롭의 값을 정할 수 없으므로 반복적인 시뮬레이션이 필요하다. 본 연구에서는 이러한 반복 연산시 과도한 시뮬레이션 시간을 줄이기 위하여 피드백 루프부분을 세분화 시켜 중첩되지 않는 각각의 피드백 루프로 하나씩 분할하였다. 이러한 시도는 유사한 분할 방식을 택하고 있는 Paris 방식 [3]에 비하여 ISCAS89 벤치마크 순차회로 [9]에 대하여 최고 20% 속도 향상을 얻을 수 있었다. 본 논문의 나머지 부분에서는 제안된 방식에 대하여 구

* 正 會 員 : 韓國電子通信研究員 交換素子研究所先任研究員
** 正 會 員 : (株)베라테스트 代表理事
接受日字 : 2000年 11月 4日
最終完了 : 2000年 12月 16日

체적인 알고리즘의 설명과 실험결과에 대한 비교를 통하여 사이클 기반 시뮬레이션 가속화의 가능성을 보여 준다.

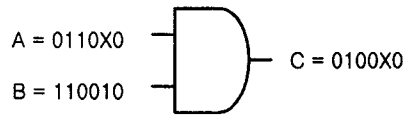
2. 사이클 기반 시뮬레이션 기본 가속화 기법

2.1 병렬 패턴 알고리즘

병렬 패턴의 사용은 한번의 시뮬레이션으로 동시에 여러 패턴에 대한 시뮬레이션 효과를 얻고자 함이다. 병렬 패턴을 나타내기 위하여 그림 1에서와 같이, 게이트 소자의 출력을 나타내는 변수 (컴퓨터 워드 크기)의 각 비트에 하나의 패턴을 대응시키는 방법을 사용하였다. 본 논문에서 사용되는 논리 값 체계가 4개 논리 값 (0, 1, X, Z)을 사용하므로, 하나의 논리 값을 나타내는데 최소 2 비트가 필요하게 된다. 따라서, 각 신호에 대하여 2개의 32 비트 워드변수를 사용하고 각 변수의 같은 위치의 비트 2개가 하나의 쌍으로 구성되게 하여 32개의 논리 값을 동시에 나타내도록 하였다. 그림 1은 각 논리값의 bit-encoding 방식을 보여준다. 병렬 패턴에서는 게이트 타입에 따라 정해진 비트 연산 (컴퓨터 언어의 연산자)을 수행하여 출력 값을 얻게 되므로 단일 패턴 연산에 비하여 논리 연산수가 병렬패턴 수 만큼 감소되므로 연산 시간이 크게 빠르게 된다. 비트 연산은 &(bit-wise AND), | (bit-wise OR), ^ (bit_wise exclusive OR), ~ (bit inversion)이 있다. 비트 연산을 통한 AND소자의 출력 값을 구하는 예제가 그림 2에 나타나 있다. 즉, AND 소자의 경우, 변수 1은 AND 연산 그리고 변수 2는 OR 연산을 수행하여 출력값을 구하도록 한다. 그 외의 다른 소자에 대하여도 비슷한 비트 연산을 수행한다. 게이트 레벨 논리소자의 연산에서 주의하여야 할 부분은 Z 논리값에 대한 처리이다. 예를 들어 1 <AND> Z의 연산결과는 X이어야 하므로 게이트 소자의 연산에서는 입력 신호에 Z 논리값이 있을 경우에는 이를 X로 변환한 후 연산을 하여야 한다.

논리값	0	1	X	Z
워드변수1	0	1	0	1
워드변수2	1	0	0	1

그림 1 논리값 bit-encoding 예
Fig. 1 Bit encoding of 4-valued logic



	워드변수 W1	워드변수 W2
A = 0110X0	011000	100101
B = 11001X	110010	001100
C = 0100X0	AND operation 010000	OR operation 101101

그림 2 2-입력 AND 소자의 병렬 패턴 연산
Fig. 2 Parallel pattern logic evaluation for a 2-input AND gate

2.2 회로 내의 논리 소자 수 감축

사이클 기반 시뮬레이션에서는 회로 내부의 타이밍을 배제하고 회로 출력 (주출력 또는 플립플롭 입력)의 논리 값만을 관측하므로 입, 출력 관계만 보장되면 회로 내부 구조가 바뀌어도 상관이 없다. 따라서, 회로의 정적 상태 동작의 변화가 없는 한에서 게이트를 없애거나 게이트 타입을 바꾸는 방법으로 많은 수의 게이트를 줄여 시뮬레이션 시간의 단축을 꾀할 수 있다. 일반적으로 시뮬레이션 속도는 회로내의 소자 수의 제곱에 비례하므로 소자 수를 줄일 수 있으면 시뮬레이션 시간의 감축을 얻을 수 있다.

예를 들면, 버퍼의 경우 입력의 값이 출력에 그대로 전달되기 때문에 버퍼가 없어도 논리 값에는 영향이 없다. 또는 AND와 AND 게이트가 연결되어 있으면, 입력 쪽 게이트를 없애고 그 입력을 다음AND 게이트의 입력으로 직접 연결해도 논리 값의 변화는 없다. 그리고, INVERTER의 경우에는 INVERTER를 입력 쪽에 붙이는 경우와 출력 쪽에 붙이는 경우가 있다. 전자의 경우의 예는 AND-INVERTER의 형태로 이 때 NAND하나로 나타낼 수 있다. 후자의 경우에는 INVERTER를 다음 단 게이트의 입력에 붙여서 ~IN_NOT이라는 새로운 게이트 타입을 정의한다. 게이트 감축 방법을 살펴보면 우선 입력이 한 개인 모든 게이트는 감축이 가능하고 입력 핀이 여러 개인 게이트의 경우는 게이트 사이의 연결구조에 따라 감축 가능 여부가 결정된다. 이때 단지 게이트를 없애고 입, 출력을 연결만 하면 되는 경우가 있고 또는 게이트 형태를 변경시켜야 하는 경우도 있다. 감축 가능한 게이트 구조의 예를 그림 3에서 설명하고 있다.

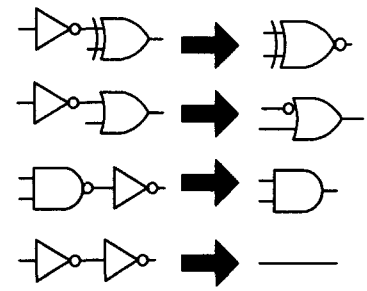


그림 3 회로 크기 감축의 예
Fig. 3 An example of circuit size reduction

2.3 게이트 소자 양식 세분화 및 function pointer 사용

앞서 설명한 회로 크기 감축의 INVERTER를 없애는 과정에서 INVERTER를 입력 쪽에 붙이는 방법과 출력 쪽에 붙이는 방법이 있었다. 이 중 출력 쪽에 붙이는 방법을 사용하여 ~IN_NOT이라는 새로운 게이트 타입을 정의하였는데, 여기서는 입력 수에 따라서, 그리고 입력에 붙은 INVERTER의 수에 따라서 게이트 타입을 세분화한다. 그러면 2-입력 AND 게이트는 "AND2" 라는 타입으로 정의되고, 3입력 OR 게이트 중에서 입력에 INVERTER가 하나 붙어있으면 "OR2_IN1_NOT" 이라는 타입으로 정의된다. 이러한 세분화된 게이트 타입은 각각의 출력 연산 함수를 갖게 된다. 그러

나, 많은 게이트 타입을 사용할 때 발생할 수 있는 문제는 원하는 출력 연산 함수를 호출하기 위해 많은 비교 판단과정이 필요하게 된다.

Table-driven 시뮬레이션에서 게이트 연산루틴으로 분기하기 위해 게이트 종류에 따라 switch ~ case 문이나 if ~ else if 구문을 사용한다. 이러한 방식은 처음부터 비교를 해서 원하는 게이트 형태를 만날 때까지 순서대로 비교하게 되므로 최악의 경우에는 모든 게이트의 종류만큼을 비교해야 한다. 하지만 function array의 pointer를 사용하게 되면 불필요한 많은 비교를 없앨 수 있다. 회로 내의 게이트 수가 많을수록, 또 게이트 타입이 많을수록 비교 판단의 횟수가 증가하게 되는데 함수 포인터를 사용하여 이러한 문제를 해결하였다. 함수 포인터 방법은 각 출력 연산 함수의 주소를 배열에 저장하고, 배열의 인자로써 미리 정의된 게이트 타입의 값을 사용한다. 이렇게 되면 배열이 차지하는 메모리 소모만이 있을 뿐, 비교 판단에 소요되는 시간을 줄일 수 있다.

2.4 결과 및 토의

본 장에서 기술한 가속화 기법에 대하여 ISCAS85 벤치마크 회로에 대하여 실험을 하였으며 전체적인 실험 환경은 SparcStation20 (RAM 128 Mbytes)이다. 표 1에서는 게이트 수 감축의 효과를 보여준다.

표2의 결과는 적용한 가속화 기법들에 의한 결과 향상 정도를 보여준다. 실험에 사용한 입력 패턴의 수는 70,560 개이고, 표 2에 나타난 값들의 단위는 초(sec)이다.

실험1: 단일 패턴에 게이트 수 감축도 하지 않고, 출력 함수를 호출하기 위해 if ~ else if 구문을 사용
실험2: 실험1에서 단일 패턴 대신 병렬 패턴을 사용
실험3: 실험2에서 if ~ else if 대신 함수 포인터를 사용
실험4: 실험3에서 게이트 수 감축

단일 패턴을 사용하며 아무런 가속화 기법을 적용하지 않을 때보다 병렬 패턴을 사용하며 앞서 설명한 가속화 기법을 사용할 때의 속도가 적어도 10배 이상, 최고 26배 이상 빨라졌다는 것을 알 수 있다. 가속화 기법을 적용하기 위한 전처리 과정의 시간은 무시할 수 있을 정도이다. 따라서, 시뮬레이션 횟수는 회로가 클수록 더 증가하므로 이러한 기법들의 사용이 많은 입력 패턴을 사용할수록 더 유리하다는 것을 알 수 있다.

표 1. 회로 크기 감축 결과
Table 1 Experimental results of circuit size reduction

	전체 게이트 수	fanout	buffer	inverter	감축후의 게이트수	감축비율 (%)
C2670	2096	454	272	321	1049	50.0
C3540	2320	579	223	490	1028	44.3
C5315	3414	806	313	581	1714	50.2
C6288	3936	1456	0	32	1448	36.8
C7552	5128	1300	535	876	2417	47.1

표 2. 가속화 기법 적용시의 시뮬레이션 시간 비교
Table 2 Comparisons of simulation CPU time for each acceleration technique

	실험1	실험2	실험3	실험4	Speedup (실험1/실험4)
c2670	241.45	36.88	16.83	9.03	26.7
c3540	251.30	43.93	20.05	10.20	24.6
c5315	461.17	104.67	51.13	29.52	15.6
c6288	450.32	126.67	59.05	37.83	11.9
c7552	682.15	197.87	99.22	53.68	12.7

3. 피드백 루프 분할 방법

앞에서의 가속화 기법 중에서 병렬 패턴 시뮬레이션은 컴퓨터 워드(32bit) 크기만큼 여러 개의 입력을 주어서 시뮬레이션 할 수 있다. 그러나 플립플롭 (이후 FF라 쓴다) 은 이전 상태를 기억하는 동작이 있으므로 병렬패턴 시뮬레이션에서 효율적으로 처리하기 위하여는 특별한 방법이 필요하다. 특히 피드백 루프가 있는 경우는 이전 상태에 대한 의존도가 있으므로 이 문제를 해결하기 위해 반복적인 시뮬레이션을 하게 되는데 본 장에서는 전체 회로에서 피드백 루프를 세분화시키는 방법에 따라 시뮬레이션 속도를 향상시킬 수 있는 방법에 대해 논의한다 [3,7].

3.1 D-FF 의 연산 방법

32 비트 워드의 병렬 패턴은 오른쪽에서 왼쪽으로 시간이 증가하면서 32개의 연속적인 시간 프레임에 대응된다. 그림 4에서 저장 기능이 있는 FF은 입력 패턴을 왼쪽으로 1 비트 위치를 이동시켜 출력을 결정한다. 그림 4는 FF의 초기 값은 'X'(unknown)일 때 입력값과 출력값을 나타내고 있다. 1비트를 왼쪽으로 쉬프트한 후에 LSB(Least Significant Bit)는 FF의 이전에 저장된 값('X')으로 채워지게 되고 MSB(Most Significant Bit)의 값은 다음 시뮬레이션 단계를 위해 FF에 저장되어 진다.

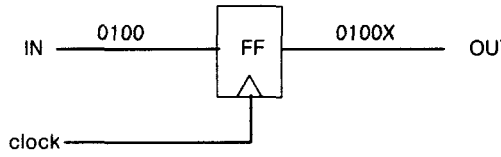


그림 4 32-시간 프레임에서의 플립플롭 논리 연산
Fig. 4 Logic evaluation of a flipflop in 32 time frames

3.2 피드백 루프 연산 방법

대부분의 순차회로에서는 피드백 루프를 가지고 있으므로 병렬패턴 시뮬레이션에서는 그림 5에서와 같이 정확한 결과값을 나타낼 때까지 반복하여 피드백 루프를 시뮬레이션해야 한다 [3]. 그림 5에서와 같이 첫 번째 시뮬레이션 단계에서 FF의 초기 값은 모두 X이고 3번의 반복적인 시뮬레이션으로

안정된 상태의 정확한 결과 값을 얻을 수 있다. 최악의 경우 반복시뮬레이션의 수는 최대 단일패턴 시뮬레이션 회수인 32번이지만 평균적으로 32 보다 훨씬 적은 횟수의 반복 시뮬레이션으로 올바른 출력 값을 얻을 수 있다. 그러나, 이러한 방식은 회로의 피드백 루프의 형태에 따라 성능을 저하시킬 수 있다. 따라서, 보다 세밀한 회로 분할이 필요하게 된다.

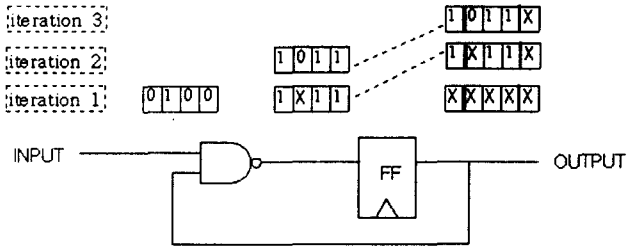


그림 5 피드백 루프의 시뮬레이션 방법

Fig. 5 Logic simulation in a feedback loop

3.3 회로 분할 방법

본 논문에서는 각각의 피드백 루프와 조합회로의 부분에 대하여 세분화된 분할을 시도하여 불필요한 조합회로의 반복 연산 부분을 제거하고 피드백 루프의 반복 부분을 줄여 성능을 개선시키고자 한다. 회로를 SCC(Strong Connectivity Component) 와 피드백 루프가 없는 부분으로 크게 두가지로 구분한다. SCC는 방향이 있는 그래프에서 한 소자에서 모든 다른 소자로 가는 경로가 있으면 strongly connected 됐다고 한다. SCC는 여기서는 서로 수렴하거나 중첩된 피드백 루프를 하나로 취급하는 개념이다. 그림 6에서 A1, A2, A3, A4는 피드백이 없는 부분이고 B1, B2, B3, B4는 SCC로 피드백 루프가 있는 부분이다.

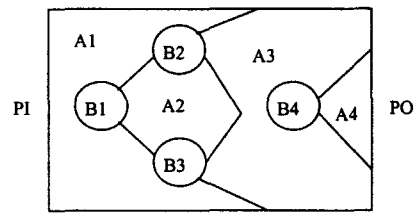


그림 6 피드백 루프가 있는 회로의 분할

Fig. 6 Circuit partition in a circuit with feedback loops

SCC 분할부분은 위에서 설명한 피드백 루프 시뮬레이션 방법으로 반복 수행해서 값을 안정화시키면서 시뮬레이션을 수행하여 간다. 각 분할 간의 시뮬레이션의 순서는 DFS (depth first search) 알고리즘을 사용하여 그 순서를 정할 수 있다. 그림에서의 시뮬레이션 순서는 A1 -> B1 -> A2 -> { B2, B3 } -> A3 -> B4 -> A4 의 순서로 시뮬레이션 해야 한다. {B2, B3}는 B2, B3 가 동시에 시뮬레이션 된다는 의미를 나타낸다. SCC와 피드백이 없는 부분을 분할하는 방법을 살펴보면, 역시 회로를 DFS 방법으로 탐색하면서 입력 게이트의 관계를 고려하면서 분할을 해나간다. 분할하는 알고리즘을 살펴보면 아래와 같다

gate[]는 회로의 모든 논리 소자를 나타낸 값.
gate[].part는 gate의 partition 정보를 저장한다.
gate[].feed_end는 피드백 루프의 끝을 표시한다.

```
Function Gate_Partition(gate[], start gate address)
max_partA = 1 , max_partB = 1;
initialize All PI's part to A(0) ;

while ( Depth First Search )
{
   if (gate[input] is already visited) then
      gate[input].part = B(max_partB++);
   if (gate[current] is already visited &&
         gate[current].part==B(i)) then
      gate[current].feed_end = TRUE;
   if (all gate[input].part is same) then
      gate[current].part = gate[input_gate].part;
   else {
     switch ( input gate partition type)
      case A(i) meet A(j)   (i≠j)
         gate[current].part = A(max_partA++);
      case A(i) meet B(j)
         if ( find a part_B(j) in fanout gate ) then
            gate[current].part = B(j) ;
         else   gate[current].part = A(max_partA++) ;
      case A(i) meet [B(j) &&
               (gate[current].feed_end==TRUE)]
         gate[current].part = A(max_partA++);
      case B(i) meet B(j) (i≠j)
         if (find a part_B(i) and part_B(j) in fanouts) then
            gate[current].part = B(max_partB++);
            convert partition B(i) and B(j) to B(max_partB);
         else if (find a part_B(i) in fanout gate) then
            gate[current].part = B(i);
         else if (find a part_B(j) in fanout gate) then
            gate[current].part = B(j);
         else if (can not found both part_B(i) and partB(j)
                  in fanout gate) then
            gate[current].part = A(max_partA++);
       case : B(i) meet [B(j) &&
            (gate[current].feed_end==TRUE)]
         if (find a part_B(i) in fanout gate) then
            gate[current].part = B(i);
         else  gate[current].part = A(maxpartA++);
       case : [B(i) && (gate[current].feed_end==TRUE)]
            meet [B2 && ( gate[current].feed_end==TRUE)]
         gate[current].part = A(maxpartA++);
   }
}
```

제안된 분할 알고리즘을 적용한 예는 그림 7과 같다. 먼저 모든 주입력 신호들을 A0로 파티션을 초기화한다. 1번과 6번 게이트에서 입력이 이미 방문했던 곳임을 발견하고 4번 게이트와 8번 게이트에 피드백 루프가 있음을 찾을 수 있다. 4번 gate.part = B1, 8번 gate.part = B2로 분할한다. 이제 1번 게이트에서 보면 입력 게이트의 gate.part는 A0 와 B1 이다. fanout 게이트를 살펴보면 4번 게이트에서 gate.part = B1을 만날 수 있으므로 1번 게이트의 분할은 gate.part = B1으로

정하고. 3번 역시 gate.part = B1, 4번은 이전에 방문했던 곳이고 이미 분할이 B1으로 되어 있으므로 feed_end = TRUE, gate.part = B1, 5번 게이트에서 보면 입력 게이트의 분할은 part A0와 [part B1 && feed_end == TRUE] 이다. 그러므로 새로운 part A1으로 분할한다. 6, 7, 8번 게이트 역시 같은 방법으로 분할할 수 있다. 분할이 끝나면 연산 순서를 결정하여야 되는데 회로를 각 분할 부분에 대하여 다시 DFS 하면 쉽게 시뮬레이션 순서를 결정할 수 있다. 즉, 그림 7의 회로분할에 대한 결과는 그림 8과 같이 A0 -> B1 -> A1 -> B2 -> A3 순서가 된다.

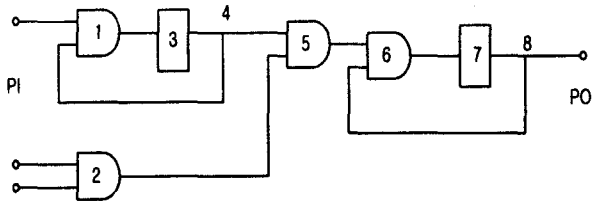


그림 7. 제안된 피드백 회로 분할의 예
Fig. 7. Example of circuit partitioning

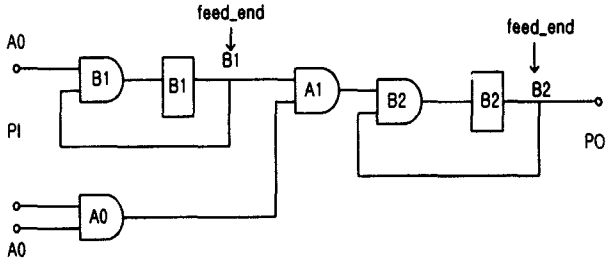


그림 8 제안된 방식의 피드백 회로 분할
Fig. 8 The result of circuit partitioning using proposed algorithm

3.4 실험 결과 및 분석

본 논문에서 설명한 피드백 루프 분할 방법을 적용하였을 때 시뮬레이션 속도를 비교하였다. 시뮬레이션에 사용된 시스템 환경은 Intel Pentium celeron 333MHz, 메인 메모리 64MB에서 실험하였다. 실험에 사용된 회로들은 12개의 ISCAS89 벤치마크 회로 [9]를 사용하였고 이 회로는 FF이 포함된 순차 회로이다. 입력 패턴은 36,000 임의의 패턴(pseudo-random pattern)을 가하였고, 아래의 모든 회로는 단일 클럭으로 동작하는 회로로 가정하였다. 표 3의 결과를 보면 기존의 PARIS 방식보다 본 논문에서 설명 한 피드백 루프 분할 기법을 적용한 시뮬레이션 기법의 성능이 대부분 빠른 속도로 동작하는 것을 알 수 있다. 표3이 결과는 단순히 회로 분할에 의한 속도 비교를 위하여 2장에서 기술한 가속화 기법을 제외하였다.

표 3. 피드백 루프 분할에 의한 속도 비교
Table 3 Comparisons of simulation speed using proposed partitioning algorithm

circuit	전체 게이트수	SCC수	Paris	제안방식	speedup
s298	142	12	1.14	0.96	15 %
s344	195	6	1.36	1.27	6 %
s349	196	6	1.36	1.25	8 %
s382	188	6	1.23	1.12	8 %
s420	253	16	0.93	0.89	4 %
s444	211	6	1.25	1.10	12 %
s526	223	15	1.41	1.22	13 %
s1423	753	6	4.51	4.13	8 %
s35932	18148	18	134.35	106.76	20 %

4. 결 론

사이클 기반 시뮬레이션은 고도로 집적화 된 동기 회로의 논리 검증을 위한 매우 빠른 시뮬레이션 방법을 제공한다. 본 논문에서는 이러한 사이클 기반 시뮬레이션에서의 성능 향상을 위해서 여러 가지 가속화 기법을 제시하였고, 각각의 가속화 기법들을 사용하여 구현한 결과를 통해서 본 논문에서 제시한 가속화 기법들이 CBS 속도 향상에 효과가 있음을 보였다. 특히, 본 논문에서는 병렬패턴을 이용한 사이클 기반 시뮬레이션 기법과 세분화된 피드백 루프 분할에 의한 시뮬레이션 속도를 향상시키는 방법에 대해서 설명하였다. 앞으로 계속해서 연구되어져야 할 사항은 실제 고집적 회로에서 존재하는 다중 클럭 영역에 대한 효과적인 시뮬레이션 기법의 개발이다. 가장 합리적인 가정으로는 각 클럭의 주기가 서로 조화 관계 (harmonically related)가 있는 경우이며 이때 각 클럭의 주기의 최대값에 해당하는 사이클을 기준으로 시뮬레이션이 가능하다. 또한, 내장된 메모리의 사용이 점점 증가되어지기 때문에 메모리를 효율적으로 사용하는 시뮬레이션 기법의 개발이 요구된다.

감사의 글

본 연구는 1999년 한양대학교 교내연구비 지원에 의하여 이루어진 결과이며 지원에 감사드립니다.

참 고 문 헌

[1] N. Murgai and M. Fujita, "Some Recent Advances in Software and Hardware Logic Simulation", Int. Conf. VLSI design, pp.232-238, Jan. 1997.

[2] Z. Barzilai, L. Carter, B. Rosen, and J. Rutledge , "HSS: A high-speed simulator", IEEE Transactions on Computer Aided Design, vol. CAD-6, NO. 4, pp.601-617, July. 1987.

[3] N. Gouders and R. Kaibel, "PARIS: A parallel pattern fault simulator for synchronous sequential circuits", Proc. International Conference on Computer-Aided

Design, pp. 542-545, Nov. 1991.
[4] Z. Wang and P. M. Maurer, "LECSIM: A Levelized Event-Driven Compiled Logic Simulator", ACM/IEEE Proc. Design Automation Conf., pp. 491-496. 1990.
[5] P. McGeer, ".Fast Discrete Function Evaluation using Decision Diagrams", IEEE Int. Conf. on Computer-aided Design, pp.402-407, 1995.
[6] P. Ashar and S. Malik, "Fast Functional Simulation Using Branching Programs", IEEE Int. Conf. on Computer-aided Design, pp.408-412, 1995.
[7] C. J. DeVane, "Efficient Circuit Partitioning to Extend Cycle simulation Beyond Synchronous Circuits", IEEE/ACM Proc. International Conference on Computer-Aided Design, pp. 154-161, Nov 1997.
[8] F. Brglez and H. Fujiwara, "A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran," Proc. IEEE Int. Symp. Circuits Syst., 1985.
[9] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits", Proc. IEEE Int. Symp. Circuits and Systems, 1989, pp.1929-1934.

저 자 소 개

박 영 호 (朴 英 鎬)

1985년 대전산업대 전자계산학과 졸업 1983년~현재 한국전자통신연구원/교환전송기술연구소/시스템종합팀/선임기술원. 관심분야 : CAD, 컴퓨터네트워크
E-mail : yhpark@etri.re.kr

박 은 세 (朴 恩 世)

1957년 11월 11일 생. 1980년 서울대 전기공학과 졸업. 1982년 한국과학기술원 전기및전자공학과 졸업(석사). 1989년 미국 텍사스주립대 전기및컴퓨터공학과 졸업(공박). 1982년-1995년 한국전자통신연구원 책임연구원. 1989년-1990년 미국 멘토그래픽스사 연구원. 1990-1991년 미국 모토롤라사 연구원. 1995년-2000년 한양대 부교수. 현재 (주)베라테스트 대표이사
Tel : 02-522-9097, Fax: 02-522-9020
E-mail : espark@veratest.com