개선된 테스트 용이화를 위한 점진적 개선 방식의 데이타 경로 합성 알고리즘

(Stepwise Refinement Data Path Synthesis Algorithm for Improved Testability)

김 태 환 † 정 기 석 ††
(Taewhan Kim) (Ki-Seok Chung)

요 약 본 논문은 세 가지 중요한 설계 기준인 테스트 용이화, 설계 면적, 및 전체 수행 시간을 동시에 고려한 새로운 데이터 경로 합성 알고리즘을 제시한다. 우리는 테스트 용이화를 위한 선행 연구들에서 제시한 세 가지 기초적 척도들에 근거하여 새로운 테스트 용이화의 우수성에 대한 척도를 정의한다. 이 척도를 이용하여, 스케줄링과 할당의 통합된 형태의, 단계식이며 점진적 개선을 통한, 합성 알고리즘을 제시한다. 벤치마크 설계와 다른 회로의 예를 통한 실험에서, 우리는 설계 면적과 수행 시간에 대해 매우 적은 추가 부담으로, 회로의 테스트 용이화가 향상됨을 보인다.

키워드 : 아키텍처 설계, 테스트, 설계 자동화

Abstract *This paper presents a new data path synthesis algorithm which takes into account simultaneously three important design criteria: testability, design area, and total execution time. We define a goodness measure on the testability of a circuit based on three rules of thumb introduced in prior work on synthesis for testability. We then develop a stepwise refinement synthesis algorithm which carries out the scheduling and allocation tasks in an integrated fashion. Experimental results for benchmark and other circuit examples show that we are able to enhance the testability of circuits with very little overheads on design area and execution time.*

Key words : Architecture design, testing, design automation

1. 서 론

상위 수준 합성은 디지털 시스템의 행동 명세를 레지스터 전송 수준(RTL) 구조로 변환하는 설계 단계를 말한다.

상위 수준 합성에서 가장 중요하게 고려해야 될 사항은 면적, 수행능력, 핀의 수, 전력 소모와 시스템 분할등을 들 수 있다. 디지탈 시스템의 복잡도가 점차 증가하면서 시스템의 테스트가 중요한 주제가 되었다. 즉, 테스트하기 쉬운 데이타 경로를 합성하는 것이 시스템 설계에서 중요한 목적이다. 본 논문에서, 우리는 쉽게 테스트 할 수 있는 디지털 시스템 설계의 문제를 다룬다.

· 본 연구는 첨단정보기술 연구센터(AITrc)를 통하여 과학재단의 지원을 받았음.
† 종신회원 : 한국과학기술원 전자전산학과 교수
tkim@cs.kaist.ac.kr
†† 비 회 원 : 홍익대학교 컴퓨터공학과 교수
kchung@cs.hongik.ac.kr
논문접수 : 2001년 9월 14일
심사완료 : 2002년 3월 19일

상위 수준 합성에서 쉽게 테스트 할 수 있는 시스템 설계 문제를 위한 여러 시도가 있었다. Papachristou 등은 내장형 자체 테스트(BIST) 기술을 사용한 자체 테스트 가능한 설계를 위한 데이타 경로 합성 방법을 제안하였다[1]. 그들은 스케줄된 데이타 흐름 그래프를 자체 테스트 가능한 데이타 경로로 바꾸는 두 가지 할당 방법을 소개하였다. 그 중 하나는 그래프 이론적 휴리스틱 알고리즘에 기초한 반면, 다른 하나는 0-1 정수 선형 계획법 공식에 기초하였다. Papachristou는 또한 BIST 기술이 사용되었을 경우 주어진 스케줄에서 국지적인 변환으로 데이타 경로의 테스트 용이화를 향상시키는 재스케줄링 변환 방법을 제안하였다 [2]. Parulkar 등은 BIST 기술 적용을 위한 레지스터 사용량을 최소화하는 방법을 제안하였으며 [3], 그들은 또한, BIST를 위해 사용될 자원의 수를 줄이기 위해 입력으로 받은 데이타 흐름 그래프 (Data-flow Graph)를 변형하는 방법을 제시하였고 [4], BIST를 위한 스케줄링를 효과적

으로 하는 방법도 제시하였다 [5]. Mujumdar 등은 스캔 테스트 기술이 사용되었을 때 테스트 용이화를 향상시키기 위한 데이타 경로 할당 방법을 제안하였다[6]. 그들은 연산의 모듈 배정과 변수의 레지스터 배정을 달리 함으로써 설계에서 셀프 루프를 줄이려고 하였다. 그러나 [1]에서와 같이 그들의 접근은 데이타 흐름 그래프에서 연산 수행을 위한 스케쥴이 정해져 있다는 가정을 하고 있다. Avra는 pseudo-무작위 내장형 자체 테스트(PR-BIST) 기술이 사용되었을 때 데이타 경로의 테스트 용이화를 향상시키기 위해 가중값이 주어진 충돌 그래프 (conflict graph)의 노드 채색에 기초한 레지스터 할당 알고리즘을 제안하였다 [7]. 그 목적은 설계에서 셀프 루프의 수를 최소화하는 것이다. 그 방법은 연산들이 이미 스케쥴 되어 있고, 수행을 위해 모듈에 결합되었다고 가정한다. Lee 등은 쉬운 테스트 용이화를 위한 데이타 경로 스케쥴링과 데이타 경로 할당 방법을 제안하였다[8, 9]. 그들의 목표는 특정 테스트 정책들에 무관한 데이타 경로의 테스트 용이화를 향상시키는데 있었다. 그들은 데이타 경로의 테스트 용이화를 향상시키기 위해 두 가지 휴리스틱 방법을 제안하였다. 첫째는 관찰성(observability)과 제어성(controllability)의 향상이고 둘째는 순차적 깊이(sequential depth) 축소이다. 첫번째는 기본 입력으로부터 결함을 야기시키고 (제어성 향상), 고장의 영향이 기본 출력으로 전달시키는 (관찰성 향상) 테스트 시퀀스의 적용을 쉽게 하기 위한 것이다. 두번째는 고장의 영향이 관찰 가능할 수 있기 전에는 가능하면 적은 하드웨어 요소를 지나서 전달될 수 있도록 하기 위한 것이다. 그들은 두 가지 목적을 위한 스케쥴링 알고리즘 (이동도 경로 스케쥴링)을 개발하였다. 일단 이동도 경로 스케쥴링 알고리즘에 의해 얻어지면, 수정된 left-edge 알고리즘 [10]이 첫 번째 목적을 위한 레지스터 할당에 사용된다. 그리고 Stok의 모듈 할당 알고리즘 [11]이 두 번째 목적을 위해 사용된다. 그러나 스케쥴링, 레지스터 할당 모듈 할당을 분리된 단계를 통해 얻는 것은 테스트 용이화 향상의 가능성을 충분히 살리지는 못한다.

우리는 테스트 용이화의 개선을 위해 데이타 경로 합성의 새로운 접근을 제안한다. 우리의 합성 과정은 일반적이고 어떤 특정한 테스트정책을 가정하지 않는다. 스케쥴링과 할당 작업이 독립적으로 수행되는 이전 연구들([1, 2, 3, 5, 6, 7, 8, 9])과는 달리 우리의 방법은 스케쥴링과 할당 작업을 한 단계에서 동시에 수행함으로써 테스트 용이화에 스케쥴링과 할당의 영향을 더욱 충분히 그리고 효과적으로 살릴 수 있도록 하였다.

2. 합성 문제

우리는 점대점 (point-to-point) 배선 형태의 구조의 아키텍쳐 설계를 가정한다. 배선의 결정은, 연산은 수행을 위한 모듈에, 변수는 데이타 저장을 위한 레지스터에 배정됨에 따라 이루어진다. 우리의 알고리즘은 스케쥴되지 않은 데이타 흐름 그래프를 입력으로 받아서 데이타 흐름 그래프에 의해 명세 된 행동이 구현되는 데이타 경로 구조를 만들어낸다. 즉, 수행을 위한 데이타 흐름 그래프에서의 연산을 스케쥴하고, 비용 식

$$C = w_1 \cdot E + w_2 \cdot H - w_3 \cdot T \qquad (1)$$

을 최소화하기 위해 모듈, 레지스터, 배선을 할당하는 것이다. 여기에서 E와 H는 수행시간과 하드웨어 비용을 나타내며, T는 4절에서 좀 더 명확히 정의 되겠지만, 데이타 경로의 테스트 용이화의 우수성 척도이다. 그리고 w_1, w_2, w_3은 가중치 (≥ 0)이다.

우리는 다음 세 가지 관찰에 기초하여 우리의 알고리즘이 최소화하고자 하는 테스트 용이화 척도를 정의한다. 여기에서 처음 두 개의 관찰은 Lee의 알고리즘에서 최소화하고자 한 기준으로 사용된 것이고 [8, 9], 세 번째 관찰은 Mujumdar [6]와 Avra [7]의 알고리즘에서 최소화하고자 한 기준에 사용된 것이다.

(1) 데이타 흐름 그래프에서 변수는 기본 입력 (primary input), 기본 출력 (primary output), 임시 변수 (temporary variable) 세 가지로 분류된다. 기본 입력이 저장되는 레지스터를 *제어가능한 레지스터*, 기본 출력이 저장되는 레지스터를 *관찰가능한 레지스터*라고 한다. 직관적으로 제어가능한 레지스터와 관찰가능한 레지스터가 많으면 데이타 경로의 테스트 용이화가 향상되는 것은 당연하다.

(2) 데이타 경로에서 하나의 레지스터에서 다른 레지스터로의 *순차적 경로*는 이들을 잇는 경로이다. 일반적으로 레지스터 쌍들 간에는 많은 순차적 경로가 있다. 순차적 경로의 *길이*는 그 경로상의 모듈의 수로 카운트한다. 레지스터 쌍 간의 *순차적 깊이*는 그 쌍 사이의 가장 짧은 순차적 경로의 길이로 정의 된다 [9]. 당연히 고장이 발견되기 전에 그 영향의 전달 시간을 최소화하기 위해 제어가능한 레지스터와 관찰 가능한 레지스터 간의 순차적 깊이는 짧을수록 좋다.

(3) 레지스터 쌍 사이의 순차적 경로의 길이가 1 이고, 두 레지스터가 같으면 그 순차적 경로를 *셀프 루프*라고 한다. 데이타 경로에서 셀프 루프는 I-경로 구조 [1,2]를 사용하는 BIST 방법뿐만 아니라 PR-BIST 방법에서 테스트 문제를 매우 복잡하게 만든다. 당연히 우

리는 설계에서 자체 루프 수가 가능하면 적은 것을 원한다.

그림 1은 [8]에서 가져온 데이타 흐름 그래프의 예이다. 변수 u, v, z, y는 기본 입력이고, 변수 x, w는 기본 출력, a, b, c, d, e, f는 임시 변수이다. 그림 2는 그림 1의 데이타 흐름 그래프로 명세 된 행동을 구현한 데이타 경로를 보인 것이다. 이 예에서 레지스터 r_1은 기본 입력(u) 값이 저장되므로 제어가능한 레지스터이다. 그리고 레지스터 r_{11}은 기본 출력(x) 값이 저장되므로 관찰가능한 레지스터이다. 반면에 레지스터 $r_{4,\ 12}$는 기본 입력(y)와 기본 출력(w) 값이 모두 저장되므로 제어가능하고 관찰가능한 레지스터이다. 레지스터 r_1에서 r_{11}로의 가장 짧은 순차적 경로가 $r_1 \rightarrow m_6 \rightarrow r_9 \rightarrow m_{5,\ 7} \rightarrow r_{11}$이므로 그 순차적 깊이는 2가 된다.

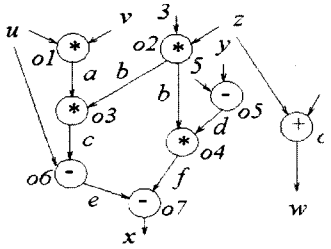

그림 1 데이타 흐름 그래프

3. 단계식 개선 합성 알고리즘

우리의 합성 알고리즘은 스케쥴 되지 않은 데이타 흐름 그래프를 입력으로 받는다. n과 m을 각각 데이타 흐름 그래프에서 연산과 변수의 수라고 두자. 처음에 우리는 단순히 n 개의 모듈과 m 개의 레지스터를 할당하고, 각 연산을 각 모듈에, 각 변수를 각 레지스터에 배정함으로써 데이타 경로를 만든다. 이 때 초기 설계는 가장 짧은 수행 시간과 가장 많은 설계 면적 (모듈, 레지스터, 배선의 최대치)을 갖는다. 그 다음, 그 초기 설계를 테스트 용이화 척도가 높도록 점진적으로 개선한다. 점진적 개선 단계를 거치면서 또한 전체 수행 시간과 설계 면적과의 변화를 고려한다.

제안된 알고리즘은 반복적으로 수행된다. 각 반복에서, 데이타 경로에서 각 두 개의 모듈이나 두 개의 레지스터를 선택하여 그것들을 합병함으로써 새로운 데이타 경로를 만든다. 두 모듈을 하나로 합병한다는 것은 그 두 개의 모듈에 배정된 모든 연산을 하나의 모듈로 재배정하는 것을 의미한다. 그래서 두 모듈을 하나로 합병하는 것은 이 모듈들의 모든 연산이 수행 중에 다른 클럭 스텝에 스케쥴 되어서 그 연산들이 같은 모듈을 공유할 수 있도록 하는 스케쥴링 제약조건을 가지게 된다. 레지스터의 합병의 경우에도 마찬가지이다. 결과적으로, 우리 알고리즘에서 그러한 스케쥴링 제약 조건이 데이타 흐름 그래프에 더해진다. (상세한 것은 5절에서 다룬다.) 설계에서 모듈이나 레지스터의 합병은 하드웨어 요소 수의 감소를 가져온다. 반면에 부과되는 스케쥴링 제약조건은 수행 시간의 증가를 가져올 수 있다.

알고리즘의 각 반복에서 우리는 모든 "합병가능한" 모듈과 레지스터 쌍들 중에서 결과로 나오는 데이타 경로의 테스트 용이화, 전체 수행 시간, 설계 면적의 척도에 대한 비용 함수에 따라 최선의 것을 선택한다. 즉, 각 반복에서, 알고리즘은 데이타 경로에서 모든 합병 가능한 모듈과 레지스터 쌍에 대해 증가되는 테스트 용이화 척도[1] ΔT를 계산하여 가장 큰 값을 가지는 k쌍을 선택한다. k는 테스트 용이화 척도와 수행시간과 하드웨어 비용 사이의 트레이드 오프 (trade-off)를 제어하기 위해 선택되는 매개변수이다. 즉, k값이 작다는 것은 테스트 용이화 척도의 증가를 하드웨어와 수행시간 감소보다 더 강조하는 것을 의미하고, k값이 크다는 것은 테스트 용이화 척도의 증가를 하드웨어와 수행시간 감소보다 덜 강조하는 것을 의미한다. 각 모듈과 레지스터의 k쌍에 대해서 알고리즘은 증가되는 수행시간 비용 ΔE와 증가되는 하드웨어 비용 ΔH를 계산하여 ΔE와 ΔH의 가중 합이 가장 작은 쌍을 선택한다. 이렇게 선택된 쌍은 합병되고, 데이타 흐름 그래프는 그 합병에서 나타나는 스케쥴링 제약 조건에 만족하도록 수정된다. 이 작업은 더 이상 전체 비용, 즉 식 1에서의 테스트 용이화 척도, 수행시간 비용, 하드웨어 비용의 가중 합의 증가가 없을 때까지 반복된다.

알고리즘 수행이 끝냈을 때, 데이타 경로와 이와 관련되어 점진적 개선 과정 중에 나타나는 부가적인 스케쥴

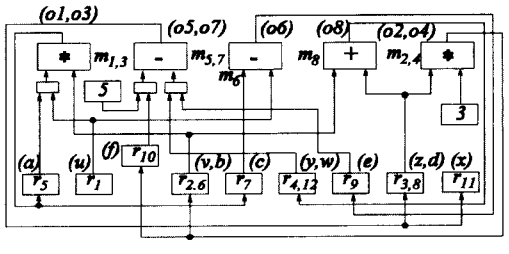

그림 2 그림 1의 데이타 흐름 그래프에서 우리의 알고리즘을 여섯 번 반복 후의 결과

1) 증가되는 비용의 계산은 4절에서 다룬다.

표 1 그림 2의 데이타 경로에서 몇 개의 모듈과 레지스터 쌍에 대한 비용 계산

쌍	Δt_1	$\Delta \hat{t}_2$	Δt_3	ΔT	ΔE	ΔM	ΔR	ΔI	ΔH	$\alpha \cdot \Delta E + \beta \cdot \Delta H$
⋮										
$m_{1,\ 3}$, $m_{2,\ 4}$	0	-4	+1	**+8.0**	+2	-1	-	0	-1.0	+3.0
$m_{5,\ 7}$, m_6	0	-3	+1	**+6.0**	0	-1	0	0	-1.0	**-1.0**
r_1, r_{10}	+1.0	-1	0	+3.0	+1	0	-1	0	-0.5	
r_1, r_{11}	-0.5	-2	0	+3.5	0	0	-1	0	-0.5	
*r_5, r_1										
5	+1.0	-1	0	3.0	0	0	-1	-1	-1.5	
r_5, r_{10}	+1.0	-2	0	**+5.0**	0	0	-1	0	-0.5	-0.5
⋮										

$\Delta T = \Delta t_1 - 2 \cdot \Delta \hat{t}_2 - 0 \cdot \Delta t_3$, $\Delta H = \Delta M + 0.5 \cdot \Delta R + \Delta I$

링 제약 조건이 더해진 데이타 흐름 그래프의 설계를 얻는다. 그 데이타 흐름 그래프는 ASAP (as soon as possible) 스케쥴링 알고리즘처럼 어떤 임계 경로 스케쥴링 알고리즘에 의해서도 스케쥴이 가능함을 보장한다.

예를 들어, 그림 2는 그림 1의 데이타 흐름 그래프에 대해 초기 데이타 경로로부터 알고리즘을 여섯 번 반복한 후의 데이타 경로를 보인 것이다. 이 때 우리는 그림 2의 데이타 경로에서 합병을 위한 모듈이나 레지스터 쌍을 선택하고자 한다.

표 1은 몇 개의 모듈과 레지스터의 쌍에 대해 증가되는 테스트 용이화 척도(ΔT), 수행시간 비용(ΔE), 하드웨어 비용(ΔH)의 상세한 계산을 보인 것이다. (*)표시된 쌍은 합병이 불가능한 것을 나타낸다. 모든 합병 가능한 쌍 중에서 ΔT 값이 가장 큰 k쌍을 선택한다. 표 1에서 선택된 쌍의 값은 ΔT열에서 강조된 글씨체로 보인 것이다. 그리고, 이 k쌍에서 $\alpha \cdot \Delta E + \beta \cdot \Delta H$값이 가장 작은 하나를 선택한다. 표의 마지막 열에서 강조된 글씨체로 나타낸 것이 최소 값으로 밝혀진 것이다.[2] 결과적으로 관련된 쌍 ($m_{5,\ 7}$, m_6)은 합병된다. 이후의 반복에서 레지스터 r_9와 $r_{3,\ 8}$, r_5와 r_{11}, r_{10}과 $r_{2,\ 6}$, $r_{5,\ 11}$과 r_7이 합병되어서 식 1에서의 전체 비용 C는 더 이상 감소되지 않게 된다.

4. 비용계산 공식

2절에서의 관찰에 따라 우리는 세 개의 테스트 용이화 척도 t_1, t_2, t_3를 다음과 같이 정의한다. S를 데이타 경로에서 레지스터의 집합을 나타낼 때, S에 속한 각 레지스터 r_i에 대해 $\delta(r_i)$를 정의하는데, 만일 r_i가, 제어가능만하거나 관찰가능만하면 $\delta(r_i)=1.0$, 제어와 관찰 모두 가능하면 $\delta(r_i)=1.5$(이 값은 당연히 $\delta(r_i)$ 보다는 크거나 같아야하고 $2\delta(r_i)$보다는 작거나 같아야한다. 따라서, 안정된 값으로 우리는 1.5를 취하였다. 하지만 설계 목적에 따라 값을 다르게 조정할 수 있다.) 그 외의 경우는 $\delta(r_i)=-1.0$(이 값 또한, 0 보다는 같거나 작은 값이 되어야하며, 본 연구에서는 임의로 $-\delta(r_i)$로 두었다.) 그러면 t_1을 다음과 같이 둔다.

$$t_1 = \sum_{r_i \in S} \delta(r_i) \tag{2}$$

S에 속한 각 레지스터 r_i와 r_j의 쌍에 대해 $sdep(r_i, r_j)$를 정의하는데, r_i가 제어가능하고, r_j가 관찰가능할 경우 $sdep(r_i, r_j)=r_i$에서 r_j로의 순차적 깊이, 그 외의 경우 $sdep(r_i, r_j)=0$ 그러면 t_2를 다음과 같이 둔다.

$$t_2 = \sum_{r_i, r_j \in S} sdep(r_i, r_j) \tag{3}$$

S에 속한 각 레지스터에 대해 $slp(r_i)$를 r_i를 통과하는 셀프 루프의 수로 정의하면 t_3을 다음과 같이 둔다.

$$t_3 = \sum_{r_i \in S} slp(r_i) \tag{4}$$

우리는 γ_1, γ_2, γ_3이 음수가 아닌 가중치 인자라고 할 때 설계의 전체적 테스트 용이화 척도 T는

$$T = \gamma_1 \cdot t_1 - \gamma_2 \cdot t_2 - \gamma_3 \cdot t_3 \tag{5}$$

로 정의한다. T값이 크다는 것은 제어와 관찰이 가능한 레지스터 수를 최대화하고 순차적 깊이, 셀프-루프의 수는 최소화하는 것을 의미하며, 이는 쉽게 테스트 가능한 설계임을 의미한다. 따라서, T의 식에서 t_2(순차적 깊이) 와 t_3(셀프 루프의 수)는 최소화하도록 마이너스가 붙어있다.

2) $\alpha=2$, $\beta=1$로 두었다.

주어진 데이타 경로에 대해 테스트 용이화 척도 t_1, t_2, t_3값이 계산된다. 만약 두 개의 하드웨어 모듈이나 두 레지스터가 합병되면 이러한 테스트 용이화 척도의 값들은 다시 계산된다. Δt_1, Δt_2, Δt_3는 결과로 얻어지는 데이타 경로에서 이런 척도의 값들의 증가치를 나타낸다. 당연히 Δt_1값은 상수 시간에 계산할 수 있다. 즉 레지스터 r_i와 r_j가 합병되면 r_{ij}가 합병 후에 얻어지는 레지스터를 나타낼 때 $\Delta t_1 = \delta(r_{ij}) - \delta(r_i) - \delta(r_j)$이다. 만약 두 모듈이 합병되면 $\Delta t_1 = 0$이다. Δt_3는 합병되기 전의 두 모듈이나 두 레지스터를 지나가는 셀프 루프의 수와 합병된 후의 모듈이나 레지스터를 지나는 셀프 루프의 수와의 차이를 계산함으로써 선형 시간에 계산할 수 있다. 그러나 Δt_2계산은 그래프에서 모든 노드 간의 가장 짧은 경로의 길이를 정하는 것과 같다. 이 계산 시간은 n이 그래프에서 노드의 수를 나타낼 때, 시간 한계는 $O(n^3)$이다. 알고리즘의 속도를 향상시키기 위해서 Δt_2값을 합병된 모듈이나 레지스터를 지나는 제어가능한 레지스터에서 관찰가능한 레지스터로의 가장 짧은 순차적 깊이와 합병되기 전의 두 모듈이나 레지스터에 대한 가장 짧은 순차적 깊이의 합과의 차이로 근사하여 선형 시간에 계산한다. 이 근사 값을 $\Delta \hat{t}_2$로 나타낸다. 그러면 결과로 얻어진 데이타 경로에서 테스트 용이화 척도의 증가치 ΔT는 다음과 같이 정의된다.

$$\Delta T = \gamma_1 \cdot \Delta t_1 - \gamma_2 \cdot \Delta \hat{t}_2 - \gamma_3 \cdot \Delta t_3 \qquad (6)$$

표 1은 그림 2의 데이타 경로에서 몇 개의 모듈과 레지스터 쌍에 대해 Δt_1, $\Delta \hat{t}_2$, Δt_3, ΔT값을 보여준다. ΔT를 계산하기 위해 $\gamma_1 = 1$, $\gamma_2 = 2$, $\gamma_3 = 0$으로 두었다.

주어진 데이타 경로 그래프와 그와 관련된 데이타 경로에 대해, 수행 시간 비용 E는 데이타 흐름 그래프에서 경로의 각 연산의 수행시간에 의한 임계 경로의 길이로 정의하고 (즉, $E = L$, 여기서 L은 임계경로의 길이), 하드웨어 비용 H는 설계에서 사용된 모듈, 레지스터, 배선 수의 가중 합으로 정의한다. (즉, M, R, I는 각각 모듈, 레지스터, 배선의 수를 나타낼 때, $H = \eta_1 \cdot M + \eta_2 \cdot R + \eta_3 \cdot I$가 되며, η_1, η_2, η_3는 가중치이다.)

결과적으로 한 쌍의 모듈이나 레지스터가 합병될 때 ΔE는 임계 경로 길이의 증가치와 같고 ΔH는 하드웨어 비용의 감소치와 같다. ΔE는 음수가 아니고, ΔH는 양수가 아니므로 $\alpha \cdot \Delta E + \beta \cdot \Delta H$는 가능하면 작은 것이 좋다.

표 1은 그림 2의 데이타 경로에서 몇 개의 모듈과 레지스터 쌍에 대해 ΔE와 ΔH값을 보여준다. ΔM, ΔR, ΔI는 각각 모듈, 레지스터, 배선의 감소치를 나타낸다. ΔH, $\alpha \cdot \Delta E + \beta \cdot \Delta H$를 계산하기 위해, $\eta_1 = 1$, $\eta_2 = 0.5$, $\eta_3 = 1$이고 $\alpha = 2$, $\beta = 1$로 두었다.

5. 스케쥴링 제약조건 추가

3절에서 지적한 바와 같이 두 개의 모듈이 하나로 합병될 때 이 두 모듈에서 수행되었던 연산은 하나의 모듈을 공유할 수 있도록 다른 클럭 스텝에 스케쥴 되어야 한다. 마찬가지로, 두 레지스터가 하나로 합병될 때는 이 두 레지스터에 저장된 변수들의 수명(lifetime)에 영향을 미치는 모든 연산들이 이러한 변수들이 하나의 레지스터를 공유할 수 있도록 스케쥴 되어야 한다. 우리는 두 모듈이나 레지스터가 합병될 때 나타나는 부가적인 스케쥴링 제약 조건을 결정하는 알고리즘을 제시한다. 그러한 스케쥴링 제약 조건은 각 반복에서 데이타 흐름 그래프에 더해진다. 결과적으로 각 반복에서 데이타 흐름 그래프와 관련된 설계의 총 수행 시간의 최소값은 데이타 흐름 그래프의 임계 경로 길이와 같다.(최소 수행 시간을 갖는 스케쥴을 얻기 위해 ASAP 스케쥴링 알고리즘을 사용한다.)

두 모듈 m_i와 m_j를 합병한다고 하자. O_{i_1}, O_{i_2}, ..., O_{i_s}는 모듈 m_i에서 수행되도록 스케쥴 된 s개의 연산이고, O_{j_1}, O_{j_2}, ..., O_{j_t}는 모듈 m_j에서 수행되도록 스케쥴 된 t개의 연산이라고 하자.

먼저 모듈 m_i와 m_j에 스케쥴 된 연산이 하나씩인, $s = 1$, $t = 1$일 경우를 생각해 보자. 그러면 이 두 모듈의 합병은 연산 o_i과 o_j이 수행 중에 다른 클럭 스텝에 스케쥴 되어야 하는 스케쥴링 제약 조건이 추가된다. 만약 이 연산들이 이미 데이타 흐름 그래프에 따라 다른 클럭 스텝에 수행되도록 제한되어 있다면 더 추가되는 스케쥴링 제약 조건은 없다. 그렇지 않은 경우에는 두 가지 가능성이 있을 수 있다. 즉, o_i이 o_j보다 먼저 수행될 경우와 o_j이 o_i보다 먼저 수행될 경우이다. 바꾸어 말하면, 우리는 첫 번째 경우에 o_i에서 o_j으로의 간선을 데이타 흐름 그래프에 추가할 수 있고, 두번째 경우에는 o_j에서 o_i으로의 간선을 추가할 수 있다. 이러한 간선을 *수행시간 disjoint 간선*이라고 한다. 수행시간 disjoint 간선은 이 간선으로 연결된 두 연산 사이에 수행 의존성이 있음을 나타낸다. 즉, 그 연산들이 다른 클럭 스텝에서 수행되어야 함을 의미한다.[3] 우리의 알고리즘은 이 두 가능성 중에 결과로 생기는 데이타 흐름 그래프의 임계 경로의 길이가 적게 증가하는 것을 선택한다.

3) 처음 데이타 흐름 그래프의 간선을 *데이타 의존성 간선*이라고 한다.

결과로 생기는 데이타 흐름 그래프의 임계 경로 길이는 수행시간 disjoint 간선을 추가하기 전의 데이타 흐름 그래프의 임계 경로의 길이와, 이 간선으로 연결되는 두 연산을 지나는 임계 경로의 길이의 비교를 통해 상수시간에 계산할 수 있다.

두 모듈 m_i와 m_j를 공유하는 연산들이 o_{i1}, o_{i2}, ..., o_{ik}와 o_{j1}, o_{j2}, ..., o_{jl}인 일반적인 경우를 생각해 보자. 두 모듈 m_i와 m_j를 합병할 경우 두 모듈을 공유하는 모든 연산들이 수행시 다른 클럭 스텝에 스케쥴 되어야 한다. o_{i1}, o_{i2}, ..., o_{ik}는 이미 모듈 m_i를 공유하고 있으므로 수행시 그들 간에 순차적 순서가 있다. 일반성을 잃지 않고, 그 순서가 o_{i1}→o_{i2}→...→o_{ik}라고 가정할 수 있다. 마찬가지로 o_{j1}, o_{j2}, ..., o_{jl} 연산 사이에서도 수행시에 순차적 순서가 있고, 이 순서가 o_{j1}→o_{j2}→...→o_{jl}라고 가정한다. 이 두 개의 순차적 순서를 하나로 합병하려고 한다. 앞 문단에서 o_{i1}과 o_{j1} 중 어느 것이 먼저 수행되어야 할지를 결정하기 위한 과정을 사용하여 o_{i1}과 o_{j1}을 검사한다. 첫 번째 경우에는, 설명했던 과정을 사용하여 o_{i2}와 o_{j1}에 대해 수행시의 순차적 순서를 결정하게 된다. 두 번째 경우에는 o_{i1}과 o_{j2}의 순차적 순서를 결정하게 된다. 당연히, 연산 o_{i1}, o_{i2}, ..., o_{ik}와 o_{j1}, o_{j2}, ..., o_{jl}의 순차적 순서를 반복적으로 결정할 수 있다. (이 아이디어는 합병정렬 (merge sorting) 알고리즘과 같다.) 레지스터 합병시 스케쥴링 제약 조건의 추가 과정도 모듈의 합병에 대한 것과 마찬가지다.

6. 실험 결과

우리의 알고리즘은 C 언어로 구현되어 Sun workstation에서 수행되었다. 모든 실험에서, 우리는 γ_1=1, γ_2=2, γ_3=1, η_1=1, η_2=0.5, η_3=1로 하였다. 표 2는 [8]의 예 *ex3*에 대해 제어 변수 k를 변화해 가면서 구한 결과를 보여준다. 우리 알고리즘의 매 반복 수행 과정에서 k값도 변화하는데, 그 값은 ρ=k/(그 반복과정에서 합병가능한 쌍의 수)에 의해 이루어진다. 예를 들어, ρ=0.2의 의미는 유효 쌍들 중에서 가장 큰 테스트 용이화 측정치를 가지는 상위 20% 쌍만을 의미한다. *클럭 스텝 수, 모듈 수, 레지스터 수, 배선 수*는 각각 사용된 클럭 스텝, 모듈, 레지스터, 배선의 수를 나타낸다. *IO 레지스터*는 제어가능한 레지스터, 관찰가능한 레지스터, 제어가능하면서 관찰 가능한 레지스터 수를 나타낸다. *순차적 깊이*는 가장 긴 순차적 깊이의 값, 평균 값, 가장 짧은 순차적 깊이의 값을 나타낸다. 표의 결과에서 보여주듯이 테스트 측정 값과, 회로 수행시간, 하드웨어량 (특히 배선수)이 ρ 값에 따라 상충적으로 제어됨을 보여준다.

표 2 제어변수 k를 각 변화된 값에 대해, [8]의 예 *ex3*에 대한 결과

	ρ=k/(# of pairs)			
	0.2	0.3	0.4	0.5
클럭 스텝 수	4	4	4	4
모듈 수	3*, 3+, 2	3*, 3+, 2-	3*, 3+, 2-	3*, 3+, 2
레지스터 수	7	6	5	5
배선 수	34	28	29	27
IO 레지스터	2/1/2	3/2/1	2/1/2	3/2/1
순차적 깊이	2/0.88/0	2/1.08/0	2/1.08/0	2/1.24/1

표 3 [8]의 예 *ex3*에 대한 결과

	Lee(MPS)	Lee(S1)	Ours1	Ours2
클럭 스텝 수	4	4	4	4
모듈 수	3*, 3+, 2-	3*, 3+, 2	3*, 3+, 2-	3*, 3+, 2-
레지스터 수	5	6	5	6
배선 수	32	33	29	29
IO 레지스터	2/1/2	3/2/1	2/1/2	3/2/1
순차적 깊이	2/1.24/0	2/1.08/0	2/1.08/0	2/1.24/1
셀프루프 수	8	6	4	2

표 3은 [8]에서의 세 번째 예 (*ex3*)에 대한 결과를 보여준다. *셀프-루프 수*는 설계에서 셀프-루프의 수를 보여준다. 표 3에서는 우리의 알고리즘이 테스트 용이화 척도에 대한 서로 다른 가중치를 사용하여 각각 Ours1과 Ours2로 표시된 두 가지 설계를 얻은 것을 보여준다. (Ours1의 경우 ρ=0.4, Ours2의 경우 ρ=0.3으로 두었음.) Lee의 알고리즘에 의해 만들어진 결과와 비교하여 우리의 알고리즘은 다른 척도에 대해서는 같거나, 셀프-루프의 수를 확실히 줄였음을 볼 수 있다. 그림 3과 그림 4는 표 3의 Ours2에 의해 만들어진 *ex3*의 스케쥴링 결과와 데이타 경로 구조를 보여주고 있다.

표 4는 [8]의 *DiffEq.*의 예에 대해 반복적인 루프 후에 데이타 흐름 그래프에서 순환하는 (cyclic) 데이타 흐름을 끊었음을 볼 수 있는 결과를 보여준다. 우리는 Lee의 알고리즘에서와 같이 첫 클럭 스텝의 판독주기 (read cycle) 전에 남아있어야 하는 모든 변수들을 주 입력으로, 마지막 클럭 스텝의 기록주기 (write cycle) 이후에 남아있어야 하는 모든 변수들을 기본 출력으로 가정하였다. 우리의 알고리즘이 하드웨어 (모듈, 레지스터)와 수행 시간의 변화 없이 순차적 깊이를 13%까지, 셀프-루프를 60%까지, 배선 수를 23%까지 줄였음을 볼 수 있다.

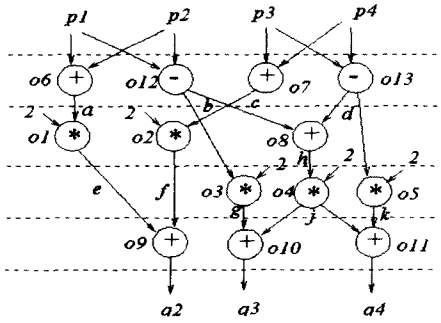

그림 3 예 *ex3*에 대한 표 3의 Ours 2의 스케쥴 결과

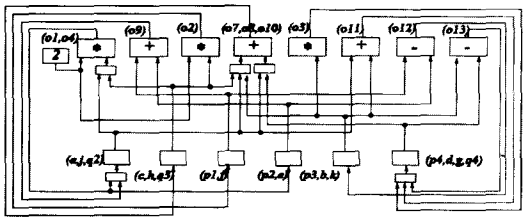

그림 4 예 *ex3*에 대한 표 3의 Ours 2의 데이타 경로 결과

표 4 [8]의 예 *DiffEq.*에 대한 결과

	Lee(MPS)	Lee(S1)	Ours
클럭 스텝 수	4	4	4
모듈 수	2*, 1+, 1-, 1<	2*, 1+, 1-, 1<	2*, 1+, 1-, 1<
레지스터 수	6	6	5
배선 수	25	26	21
IO 레지스터	3/2/1	3/2/1	3/2/1
순차적 깊이	2/1.24/0	2/1.17/0	2/1.08/0
셀프루프 수	4	5	2

표 5 [8]의 예 *DiffEq.*에 대해 자체 루프에만 가중치를 두어 얻은 결과

	Ours1	Ours2	Mujum	Mujum
클럭 스텝 수	4	4	-	-
모듈 수	2*, 2(-,-,<)	2*, 3(+,-,<)	2*, 2(+,-,<)	2*, 3(+,-,<)
레지스터 수	5	5	5	5
MUX 입력수	6	5	6	5
자체 루프 수	1	1	1	1

표 5는 *DiffEq.*의 예에 대해, Mujumdar [8]의 결과와 공정하게 비교하기 위해 자체 루프 수 만을 최소화하도록 가중치를 주어 얻어진 결과를 보여주고 있다. 결과적으로 우리는 Mujumdar의 것과 같은 수의 자체 루프를 가지는 설계들을 생성하였다.

마지막으로, 제안한 테스트 용이화를 위한 측정이 회로의 테스트에 관계되는지를 보기 위해, Lee와 우리의 알고리즘으로 생성된 회로들에 대해 고장 시뮬레이션을 시행하였다. 알고리즘들로부터 RTL (Register Transfer Level) 설계를 생성해 내고, 그 설계들로부터 2-비트와 4-비트 게이트 단계 회로들을 만들어 내었다. 그 다음 *SIS* [12]를 사용하여서 논리 최적화를 수행하고, 그 최적화된 게이트 회로들에 대해서 *STEED* [13]를 이용해서 고장 검출율을 구하였다. 표 6은 2-비트 설계에서의 우리의 제안한 방법과 Lee의 방법을 사용했을 때 고장 검출율에 대한 비교를 보여준다. 결과로부터 우리는 100%의 고장 검출율을 가지는 회로를 생성함을 보여준다. 또한, 표 7은 2-비트 설계에서의 우리의 제안한 방법과 Lee의 방법을 사용했을 때 고장 검출율에 대한 비교를 보여주고 있다. *ex3*의 경우, 우리는 Lee의 경우 보다 1.2% 높은 고장 검출율을 가지면서, 11 배나 빠른 테스트 시간을 가진다. *DiffEq.*의 경우 Lee의 경우 보다 무려 84배 빠른 테스트 시간을 가지고도 고장테스트 수에서는 3배 정도가 많음을 볼 수 있다.

표 6 2-비트 설계에서의 고장 검출율 결과

		고장 테스트 수	고장 검출율 (%)	테스트 시간
ex3	Ours	370	100%	3.1s
	Lee(MPS)	384	99.74%	2.5s
DiffEq.	Ours	284	100%	0.3s
	Lee(MPS)	363	100%	2.4s

표 7 4-비트 설계에서의 고장 검출율 결과

		고장 테스트 수	고장 검출율 (%)	테스트 시간
ex3	Ours	1098	99.45%	7.9hr
	Lee(MPS)	1031	98.25%	88.6hr
DiffEq.	Ours	370	100%	3.1s
	Lee(MPS)	970	100%	261.5s

7. 결 론

본 논문에서 우리는 세 가지 중요한 설계 기준인 테스트 용이화, 수행 시간, 설계 면적을 동시에 고려한 새로운 데이타 경로 합성 알고리즘을 제시하였다. 스케쥴

링과 할당 작업을 독립적으로 수행한 테스트 용이화를 위한 합성의 선행 연구들과는 달리, 우리의 합성 알고리즘은 테스트 용이화에 대한 스케쥴링과 할당의 효과를 더욱 충분하고 효과적으로 살리기 위해 스케쥴링과 할당 작업이 합해진 형태를 고려하였다. 그리고 실험적 결과는 우리의 통합된 알고리즘이 매우 좋은 설계를 만들어 냄을 보여 주었다. 그러나, 좀더 안정된 알고리즘이 되기 위해서는 설계에 따라 가장 적합한 (제안한 비용 공식의) 가중치 값들과 ρ 값을 결정하는 것이 계산 시간을 줄이고 최적 결과를 유도하는데 중요한 요소가 된다. 가중치를 정하는 원칙은 설계자의 경험과 설계 테스트 타입 (예: BIST, scan-test)에 따라 달라지며, 가중치의 상대적 값의 변화가 설계 결과에 미치는 영향에 대해서는 반복적인 알고리즘 수행을 통한 관찰에 현재 의존하기 때문에 이에 대한 체계적 연구가 앞으로 필요하다 하겠다.

참 고 문 헌

[1] C. A. Papachristou, S. Chju, and H. Harmanani, "Data Path Synthesis Method for Self-Testable Designs," *Proc. Design Automation Conference*, pp.378-384, 1991.

[2] C. A. Papachristou, "Rescheduling Transformation for High Level Synthesis," *Proc. International Symposium on Circuits and Systems*, pp.766-769, 1989.

[3] I. Parulkar, S. Gupta, and M. A. Breuer, "Data path allocation for synthesizing RTL designs with Low BIST area overhead," *Proc. Design Automation Conference*, pp.395-401, 1995.

[4] I. Parulkar, S. Gupta, and M. A. Breuer, "Introducing Redundant Computations in a Behavior for Reducing BIST Resources," *Proc. Design Automation Conference*, pp.548-553, 1998.

[5] I. Parulkar, S. Gupta, and M. A. Breuer, "Scheduling and Module Assignment for Reducing BIST Resources," *Proc. Design, Automation and Test in Europe*, pp. 66-73 1998.

[6] A. Mujumdar, K. Saluja, and R. Jain, "Incorporating Performance and Testability Constraints during Binding in High-level Synthesis," *IEEE Trans. on CAD*, Vol.15, No.10, October 1996.

[7] L. Avra, "Allocation and Assignment in High-Level Synthesis for Self-testable Data Paths," *Proc. International Test Conference*, pp.463-472, 1991.

[8] T-C. Lee, W. H. Wolf, and N. K. Jha, "Behavioral Synthesis for Easy Testability in Data Path Scheduling," *Proc. International Conference on Computer-Aided Design*, pp.616-619, 1992.

[9] T-C. Lee *et al.*, "Behavioral Synthesis for Easy Testability in Data Path Allocation," *Proc. International Conference on Computer Design*, pp.29-32, 1992.

[10] F. J. Kurdahi and A. C. Parker, "REAL: A Program for Register Allocation," *Proc. 24th Design Automation Conference*, pp.210-215, 1987.

[11] L. Stok, "An Exact Polynomial Time Algorithm for Module Allocation," *Proc. The fifth Int. Workshop on High-Level Synthesis*, pp.69-76, 1991.

[12] E. M. Sentovich, *et al.*, "Sequential Circuit Design using Synthesis and Optimization," *Proc. International Conference on Computer Design*, pp.328-333, 1992.

[13] A. Ghosh, S. Devadas, A. R. Newton, "Test Generation for Highly Sequential Circuits," *Proc. International Conference on Computer-Aided Design*, pp.362-365, 1989.

김 태 환

1993년 미국 일리노이대 전산학 박사. 현재 한국과학기술원 전산학과 부교수

정 기 석

1998년 미국 일리노이대 전산학 박사. 현재 홍익대학교 컴퓨터공학과 조교수