계층적 설계에서의 타이밍 최적화를 위한 지능형 논리합성 알고리즘

이 대 희† · 양 세 양††

요 약

본 논문에서는 아키텍춰 수준에서 타이밍최적화를 효과적으로 수행하기 위한 지능적인 재합성 기술에 대하여 연구하였다. 구체적으로는 아키텍춰-수준에서 계층 구조를 가지는 회로 구조에 기존의 조합적 타이밍최적화 방법을 적용함으로서 발생하는 문제점을 해소시킬 수 있는 방법을 제시하였다. 접근 방법은 우선 설계자가 설계한 계층 구조를 유지시키는 방법으로 기존의 retiming 방법과 peripheral retiming 방법을 응용하여 서브컴퍼넌트 내 조합논리회로 부분을 확대하는 방법을 이용한다. 이와 같은 방법이 좋은 결과를 가져오지 못할 때 다른 접근 방법으로서 기존의 서브컴퍼넌트들로 이루어지는 경계를 새로운 경계를 가지는 새로운 서브컴퍼넌트들로 변형시켜 서브컴퍼넌트들 각각의 독립적인 타이밍최적화로 전체 회로에 대한 타이밍최적화를 이끌어 낼 수 있도록 한다. 본 논문은 아키텍춰-수준에서 계층적 구조를 가지는 회로에 대한 새로운 접근을 시도하고 있는데, 회로가 크고 복잡해짐에 따라 설계자가 실제 회로를 대부분 서브컴퍼넌트화하여 계층적 구조를 가지도록 설계하는 것이 일반적인 상황에서 이의 효능성을 실험적으로 입증할 수 있었다.

Intelligent Logic Synthesis Algorithm for Timing Optimization In Hierarchical Design

Dae-Hee Lee† · Sae-Yang Yang††

ABSTRACT

In this paper, an intelligent resynthesis technique for timing optimization at the architecture-level has been studied. The proposed technique can remedy the problem which may occur in combinational timing optimization techniques applied to circuits which have the hierarchical subblock structure at the architectural-level. The approach first tries to maintain the original hierarchical subblock structure while minimizing the longest delay by retiming and peripheral retiming. If this is unsuccessful, new block boundaries have been formed so that timing optimization for newly constructed individual subcomponent from the partition can result in minimizing the longest delay of whole circuit. This paper tries to find a new approach to timing optimization for circuits which have hierarchical structure at architectural-level, and has verified its effectiveness experimentally. We claim its usefulness from the fact that most designers design the circuits hierarchically due to the increase of design complexity.

※ 본 논문은 정보통신부의 대학기초연구지원사업(과제번호 : 96049-IT1-I1)의 연구비지원에 의한 연구결과입니다.
† 비 회 원 : 삼성전자
†† 정 회 원 : 부산대학교 컴퓨터공학과 교수
논문접수 : 1998년 9월 10일, 심사완료 : 1999년 3월 13일

1. 서 론

회로의 타이밍최적화를 위하여 아키텍춰-수준에서 효과적으로 수행할 수 있는 지능적인 합성기법에 대하여 연구하기 위하여, 우선 현재의 합성 기술에서 타이밍최적화를 수행하는 일반적인 기법을 언급하고, 이러한 방법의 문제점을 보기로 한다. 현재 하드웨어구술언어를 사용하여 설계를 하는 경우에 가장 일반적인 방법이 아키텍춰-수준에서 구조적 관점(structural view)으로 설계(이를 다른 말로는 레지스터전송-수준(register transfer level)에서의 설계라고도 함)하는 것이다. 이와 같은 설계에서는 설계 대상 회로를 하드웨어구술언어로써 서브컴퍼넌트들이 계층적(hierarchical)으로 연결된 구조로 구술하고, 기술라이브러리(technology library)내의 셀들을 사용한 서브컴퍼넌트들의 최적화된 구현은 논리합성툴을 이용하여 자동적으로 생성시키게 된다. 그러나 이와 같은 방식은 타이밍최적화에 많은 취약성을 가지고 있는 문제점이 있는데, 이에 대한 이유는 다음과 같다.

현재의 논리합성 기술의 한계로 인하여 논리합성의 수행은 각 서브컴퍼넌트 별로 독립적으로 수행하게 된다. 이는 곧 면적최소화 혹은 타이밍최적화가 각 서브컴퍼넌트 별로 독립적으로 수행되어 진다는 것인데, 면적최소화의 경우에는 각 서브컴퍼넌트들을 최소화시킴으로서 회로 전체의 입장에서도 최소화가 상당한 정도로 가능하게 되지만, 타이밍최적화의 경우에는 각 서브컴퍼넌트들을 독립적으로 타이밍최적화하는 것이 회로 전체의 입장에서의 타이밍최적화와는 전혀 별개가 될 수 있다는 것이다. 이를 좀 더 자세히 보기로 한다. 논리합성에서의 타이밍최적화는 회로내의 최장지연시간(longest delay)(회로내의 조합회로(combinational circuit)에서의 최장경로 상의 지연시간)을 최소화시켜 이에 상응하는 높은 클럭을 회로에 가함으로서 달성한다[1,2,3]. 그러나 실제 설계하는 회로의 대부분의 경우에 이와 같은 최장경로는 회로내의 서브컴퍼넌트 하나에 존재하는 것이 아니라, 최소한도 두 개 이상의 서브컴퍼넌트에 걸쳐서 존재하게 되는 것이 일반적이다. 이와 같은 상황에서 각 서브컴퍼넌트별로 독립적으로 각 서브컴퍼넌트의 최장지연시간을 최소화시킨다면 전체 회로의 최장지연시간은 오히려 증가할 수 있는 최악의 상황도 일어날 수 있다. 이와 같은 심각한 문제점에 대처하기 위하여 현재 상용화된 기술에서는 다음과 같은 해결책들을 제시하고 있다. 첫 번째 방법은 사용자가 하드웨어구술언어로 회로내의 각 서브컴퍼넌트를 구술시에 서브컴퍼넌트의 입출력 포트에 레지스터를 가능한 한 위치시키게 하여 회로내의 최장지연시간이 한 서브컴퍼넌트 내부에 존재하도록 하는 것인데[4], 이 방법은 회로의 함수적 기능에 따라서는 언제나 이와 같이 할 수 없는 경우가 자주 있다는 것과 설계자에게 이와 같은 코딩을 요구하여야 하는 제약을 발생시킨다는 다른 문제점들을 발생시킨다. 두 번째 방법은 사용자가 최장경로를 내포하고 있는 두 개 이상의 서브컴퍼넌트들을 하나의 블럭으로 묶어서 이 블록을 대상으로 논리합성 소프트웨어가 타이밍최적화를 수행시키게 하는 것이다[5]. 이 방법은 하나의 블록으로 합쳐지는 서브컴퍼넌트들을 사용자로 하여금 선정하게 하는 과정이 매우 번거럽고 많은 시간을 소비하게 한다는 것과 최장경로가 많은 수의 서브컴퍼넌트들에 걸쳐서 존재하는 경우에 이들을 합친 블록의 크기가 논리합성 툴이 타이밍최적화를 수행하기가 불가능할 정도로 커질 수 있다는 문제점을 야기시킨다.

본 논문은 이와 같은 현재의 타이밍최적화 방법의 문제점을 효과적으로 해결하는 아키텍춰-수준에서의 재합성에 의한 지능적 타이밍최적화 기법에 대한 것이다. 연구된 방법은 실제 회로 설계에서 순차논리회로가 대부분인 서브컴퍼넌트들이 조합논리회로와 레지스터로 이루어져 있다는 점에 착안해 아키텍춰-수준에서 레지스터를 적절히 분할하여 조합논리회로의 타이밍 최적화 방법을 효과적으로 이용하는 방법을 취하고 있다. 궁극적으로 이와 같은 레지스터와 조합논리부분을 분할하여 타이밍 최적화를 수행하는 방법의 효율성은 어떻게 효과적으로 레지스터 부분과 조합논리회로 부분을 분할하여, 새롭게 형성된 조합논리회로 부분을 기존의 효과적인 조합 논리회로 타이밍 최적화 방법을 통해 최적화 할 수 있는가 하는 데 있다. 이에 따라 본 논문에서는 아키텍춰-수준에서 새로운 경계를 결정하는 여러 방법들을 이용해 타이밍최적화 입장에서 가장 효과적인 경계를 결정하는 방법을 제시하고자 한다. 회로가 복잡하고 다양해짐에 따라 설계자들은 회로의 설계의 편리성을 위해 회로를 계층적 구조를 가지도록 하고 있기 때문에 회로가 계층적인 구조를 가지고 있을 때에 계층정보를 유지해 주는 것이 설계자가 회로를 검증하고 수정하는데 용이하게 된다. 본 논문에서는 이러한 계층정보를 효과

계층을 이용해 재사용 설계를 실현할 방법을 제시한다.

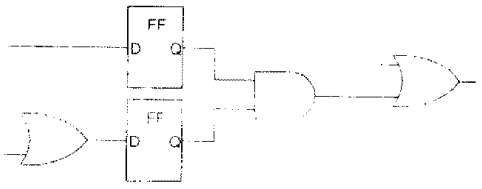
2. 아키텍춰-수준에서의 레지스터/조합 모듈 분할을 이용한 타이밍최적화

현재 대부분의 복잡한 회로는 VHDL이나 Verilog같은 하드웨어구술언어를 사용하여 아키텍춰-수준에서 구조적인 관점으로 설계가 이루어지고 있다. 이와 같은 설계에서는 설계 대상 회로의 서브컴퍼넌트들 간의 연결과 이들간의 신호 전송시의 타이밍 관계를 서브컴퍼넌트들의 함수적 기능과 함께 설계자가 구체적으로 하드웨어구술언어로써 언급하게 된다. 그러나, 기존에 연구되었던 타이밍최적화 방법들[1,2,3]은 회로가 계층적인 구조를 가지고 있지 않은 회로에 관한 연구가 대부분이었다. 따라서 계층적인 구조를 가진 회로는 각 서브컴퍼넌트별로 타이밍 최적화를 실행하는 방법을 취하거나 아니면 계층구조를 무시하고 모두 최하위 수준으로 만들어(flattening) 전체 회로에 대하여 타이밍 최적화를 실행하였는데, 이러한 접근의 문제점들은 이미 언급한 바와 같다. 우선 각 서브컴퍼넌트별로 타이밍 최적화를 실행하는 방법은 설계자가 정의한 계층구조를 그대로 유지한 상태로, 그 서브컴퍼넌트를 가지고 계층구조 속에서 타이밍 최적화를 실행하게 되는 것이다. 즉, 전체 회로 속의 각 서브컴퍼넌트(subcomponent)의 경계를 그대로 가지고 실행하여 그 서브컴퍼넌트 하나를 봤을 땐 주어진 지연시간의 제한 조건을 만족하거나 최적의 지연시간을 가지도록 하는 것이다. 이것은 각 서브컴퍼넌트마다의 타이밍최적화는 이루었지만 각 서브컴퍼넌트간의 지연시간에서 비롯된 전체 회로의 타이밍최적화를 이루었다고 전혀 보장할 수 없다. 이와 같은 상황의 예를 설명하면 다음과 같다. (그림 2.1)에서 보는 바와 같이 A 모듈, B 모듈 그리고 C 모듈을 서브컴퍼넌트로 가진 회로를 설계하고 이 회로의 타이밍 최적화를 위해 각 서브컴퍼넌트별로 타이밍 최적화를 실행하여 그림과 같이 회로를 결정하였다고 가정한다. 그렇게 함으로써 각 서브컴퍼넌트는 해당 서브컴퍼넌트 안에서 최적의 지연시간을 가지도록 합성되어졌거나, 주어진 지연시간을 만족하도록 줄여졌다고 가정한다. 그러나 실제회로에서 최장경로는 A, B 서브컴퍼넌트 사이의 연결 상에 있는 "path a"였고, 그 경로의 지연시간에 대한 고려가 타이밍최적화 과정에서 부족하게 되었다고 하면, 여전히 전체 회로는 주어진 지연시간을 만족시킬 수 없게 되거나 최악의 경우에는 타이밍최적화에 의하여 최장지연시간이 오히려 늘어날 수 있게 되는 것이다. 또한 계층구조를 무시한 채 모든 서브컴퍼넌트를 최하위 수준으로 만들어서 타이밍최적화를 실행하게 되면, 회로가 커짐으로 회로내의 너무 많은 최장경로군들이 존재해서 타이밍최적화의 실행시간이 비현실적으로 오래 걸리게 되어 실제 회로 설계에 대해서는 이를 적용하는 것은 불가능하다.

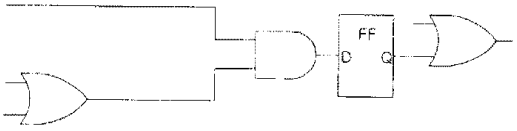
2.1 예비단계

[정의] : 동기 순차회로(이하 순차회로)는 게이트(node)들과 게이트들 사이의 연결(connection, 또는 edge)들로 구성된 가중 방향성 그래프(weighted directed graph)로 표현할 수 있으며, 이를 logic graph로 칭한다. 각 게이트들은 지연시간을 가지고 각 연결들은 거기에 존재하는 edge-triggered D 플립플롭의 수를 나타내는 음수가 아닌 정수 값의 weight를 가진다.

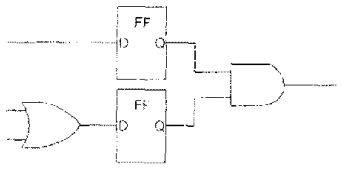
Retiming[6, 7]이란 순차회로 상의 플립플롭의 위치는 순차회로의 함수성(functionality)을 변화시키지 않으면서 이동시키는 변환이고, peripheral retiming[7]은 순차회로 상의 플립플롭들을 모두 주 출력, 혹은 주 입력 연결선으로 이동시키는 변환이다. 당연히 모든 순차회로가 항상 retiming, 혹은 peripheral retiming이 가능하지는 않다. (그림 2.1) (a)의 회로를 retiming한 회로가 (그림 2.1) (b)이고, (그림 2.1) (a)의 회로는 peripheral retiming 가능하지는 않으나, (그림 2.1) (c)의 회로를 peripheral retiming한 회로가 (그림 2.1) (d)이다. (그림 2.1) (c)의 회로는 retiming이 (따라서, peripheral retiming도) 가능하지 않은 회로이다.



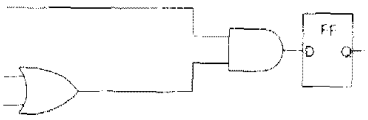
(a) retiming 가능 회로



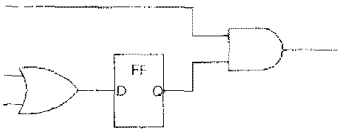
(b) retiming된 회로



(c) peripheral retiming 가능 회로



(d) peripheral retiming된 회로



(e) retiming 불가능 회로

(그림 2.1) Retiming과 peripheral retiming 예제 회로

2.2 알고리즘

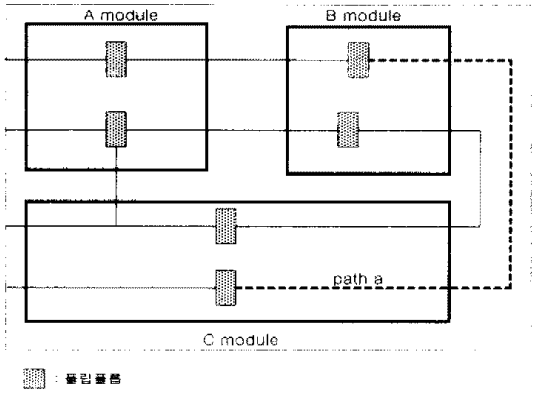
본 논문에서는 회로의 타이밍최적화를 실행하였을 때 서브컴퍼넌트들간을 고려하는 타이밍최적화를 실행할 필요성을 최소화하도록 하는 방법을 제안하였다. 우선 설계자가 정의한 아키텍춰-수준의 계층구조를 유지하면서 조합논리회로의 경계를 확대할 수 있는 방법을 생각하고, 그렇지 못할 경우 새로운 서브컴퍼넌트를 형성하여 조합논리회로의 경계를 확대할 수 있도록 하는 본 방법은 계층 구조를 가지는 아키텍춰-수준에서 구조적 관점으로 구술된 설계에서 각 서브컴퍼넌트들에 해당하는 순차논리회로들에 기존의 조합논리회로 타이밍 최적화 방법을 이용하기 위해 레지스터를 순차논리회로로부터 효과적으로 분할하여 분할된 블록별로 순차적 타이밍최적화를 적용하는 것인데, 전체 알고리즘은 아래와 같다.

```
retom(G(V, E, W))
{
        repeat
        {       create interconnection, weight table b/w component;
                search each primary input/output's weight ≥ 1
                in sub_component;
                if (exist) {
                        compute path delay sub_component;
                        search other path's weight ≥ 1;
                        if (yes)
                                peripheral retiming or/and retiming;
                        maintain sub_component structure;
                } else if (retiming possible) {
                        retime sub_component;
                        maintain sub_component structure; }
                else
                { get a constraint of sub_component information; }
                construct new boundary sub_component using
            a constraint;
                save a modified sub_component;
        } until (no more sub_component modification possible)

        while(sub_component)
        {       create temp sub_component after
                   dividing register from sub_component;
                run timing_optimizer with temp sub_component;
                reconstruct sub_component with register;
                if (retiming possible)
                        retimie sub_component;
        }
}
```

2.3 첫 번째 방법 : 기존 계층 구조를 유지

첫 번째 접근 방법은 우선 설계자가 설계한 아키텍쳐의 계층 구조를 유지시키면서 조합논리회로 부분을 확대할 수 있는 방향으로 접근하는 방법이다. 이 방법은 기존의 retiming[6,7]과 peripheral retiming[7]을 응용한 방법이다. 계층구조를 그대로 유지하는 것은 설계검증을 수행할 때 유리한데, 이것은 설계자의 의도대로의 구조를 계속 유지하고 있기 때문에 설계자가 다시 수정하거나 검증할 때 회로내 어디서 설계 오류가 생겼는가를 찾아내기가 용이하기 때문이다.

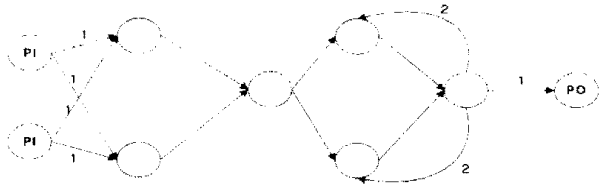


(그림 2.2) 계층구조로 이루어진 회로 구조의 예

이 방법은, 첫째로 각 서브컴퍼넌트의 주 입력 edge와 주 출력 edge를 검사하여 edge의 *weight* ≥ 1 여부를 가린다. 만일 *weight* ≥ 1이면 해당 서브컴퍼넌트의 최장지연시간은 이 서브컴퍼넌트 내부에만 존재하게 되고, 이 서브컴퍼넌트에 대한 타이밍 최적화를 실시하여 최소의 지연시간을 가지도록 하거나, 주어진 지연시간을 만족시키면 이 서브컴퍼넌트는 전체 회로의 지연시간에 영향을 주지 않게 된다. 즉, 이 서브컴퍼넌트와 연결된 다른 서브컴퍼넌트는 모두 레지스터를 거쳐 값을 주고받기 때문에 경로의 지연시간을 고려할 때 경로의 *weight*가 *0*인 경로에 대해서만 임계 지연시간을 고려하는 것에 위반되어 최장지연시간 계산에서 배제된다. (그림 2.3) (a)는 순차논리회로의 한 예이고 (그림 2.3) (b)는 이 회로의 그래프이다. 이 회로를 한 서브컴퍼넌트로 보면 이 서브컴퍼넌트의 주 입력단과 주 출력 단에는 레지스터가 모두 존재한다. 즉, edge의 *weight* ≥ 1을 만족하게 된다. 이런 서브컴퍼넌트는 임계경로가 서브컴퍼넌트 내부에만 존재하게 되고 그리하여 주 입출력에 있는 레지스터를 제거하여 내부가 조합논리회로만으로 구성되는 가를 검사한다. 이 회로는 내부에 cycle을 형성하는 경로가 있기 때문에 주 입력, 출력에 있는 레지스터를 제거하고 난 뒤 다시 cycle을 형성하는 경로 상에 있는 레지스터를 제거해 주면 된다. 그렇게 하고 나면 내부는 조합논리회로로 이루어지기 때문에 서브컴퍼넌트를 그대로 유지한 채 타이밍최적화를 수행할 수 있게 된다.

두 번째로 다시 서브컴퍼넌트 내부의 retiming 가능 여부를 가리고 다음 단계인 타이밍 최적화로 가는데 (그림 2.3)의 회로에서는 더 이상의 retiming은 필요 없는 경우가 된다. 한편, 주 입력 edge와 주 출력 edge 모두에 레지스터가 존재하지 않고 어느 한 쪽에만 있는 경우라든지 원래는 없지만 내부의 레지스터를 옮길 수 있을 때 주 입, 출력 edge 어느 한 쪽으로만 옮겨지는 경우[1]가 있다. 이런 경우엔 어느 쪽으로 옮길 지는 해당 서브컴퍼넌트와 연결되어 있는 다른 서브컴퍼넌트의 타이밍 정보를 참조하여야 한다.

(a) Logic Circuit



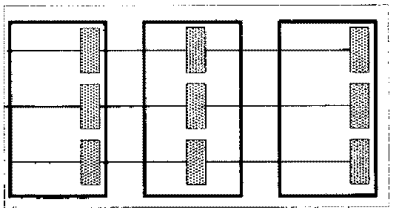
(b) Logic Graph

(그림 2.3) 예제 순차회로

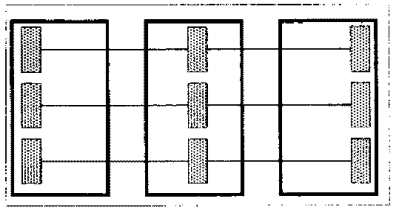
(그림 2.4) (a)에서와 같이 전후 서브컴퍼넌트들 역시 주 출력 edge쪽에만 레지스터가 있다고 한다면 해당 서브컴퍼넌트 역시 주 출력 edge쪽으로 옮기는 retiming을 실행하여야 한다. 물론 (그림 2.4) (b)처럼 어느 쪽으로 옮길 지가 명확하지 않은 경우는 서브컴퍼넌트간의 최장지연시간을 계산해 작은 쪽으로 옮긴 후 두 서브컴퍼넌트들을 합쳐서 새로운 서브컴퍼넌트

1) 주 입력에서 주 출력에 이르는 path의 weight가 1인 경우, 한 쪽으로만 옮겨진다.

로 형성하는 방법을 선택한다. 이런 방법으로 retiming을 실행한 후, 해당 서브컴퍼넌트들의 타이밍최적화를 위해서 그 서브컴퍼넌트의 주 입력 edge와 주 출력 edge에 있는 레지스터를 모두 제거했을 때 조합논리회로가 되면 이상적으로 타이 최적화를 실행할 수 있다. 그러나 그렇지 못할 경우, 다시 이 서브컴퍼넌트 내부를 조사하여 레지스터가 존재할 경우엔 peripheral retiming을 실행하여 해당 레지스터 모두를 입출력 주변으로 옮기는 시도를 한다. 여기서도 peripheral retiming이 성공적으로 수행되면 회로 내부가 조합논리회로로만 구성되어 바람직하지만 그렇지 못할 경우, 즉 모든 레지스터에 대해 실행되지 못할 경우, 특히 레지스터에 cycle이 형성되어 있는 경우에는 위에서 구술한 retiming 방법이 적용되기 어렵게 된다(그림 2.5). 그러나 이런 경우에는 그 레지스터를 그대로 둔 상태로 제거하는 방식을 사용한다. 즉, 레지스터의 입력과 출력을 해당 서브컴퍼넌트의 주 입력과 주 출력으로 임시적으로 할당하는 방법을 생각할 수 있다. 이렇게 하면 역시 조합적 타이밍 최적화 알고리즘을 이용할 수 있게된다. 그렇지 못한 경우에는 조합논리회로 타이밍최적화 알고리즘을 이용하지 않고 retiming을 이용하여 최대한의 타이밍 최적화를 실행하거나 아니면 조합적 타이밍최적화 알고리즘을 이용하기 위해 서브컴퍼넌트를 세분화하는 방법을 사용한다.



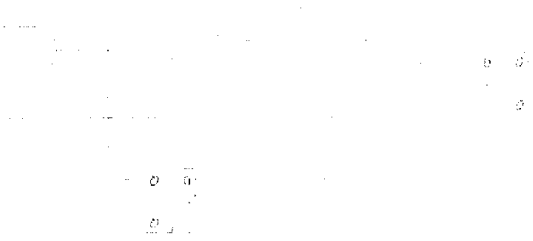
(a) 사례 1



(a) 사례 2

(그림 2.4) 계층적 회로 구조의 예들

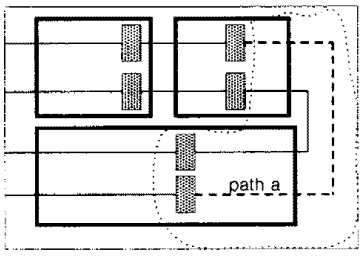
(그림 2.5)에서 보는 것처럼 주 출력 단에 있는 레지스터는 retiming으로 옮겨 놓았다. 그러나 cycle을 형성하고 있는 경로가 있는 회로의 경우 실제 레지스터의 retiming이 어렵다. 이런 경우는 cycle을 형성하는 게이트들과 같은 서브컴퍼넌트에 할당시켜 놓는 방법을 사용한다.



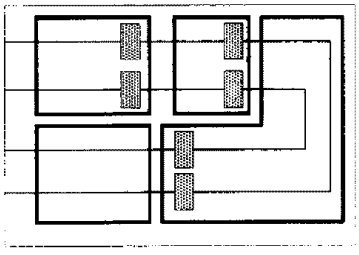
(그림 2.5) 순차회로의 다른 예

2.4 두 번째 방법 - 새로운 서브컴퍼넌트의 설정

첫 번째 접근에서 만족할 만한 결과를 도출하지 못하였을 경우, 즉 각 서브컴퍼넌트의 주 입력 edge와 주 출력 edge에 레지스터가 없거나 주 입력 edge와 주 출력 edge로 레지스터를 옮길 수 없는 경우에[2] 각 서브컴퍼넌트별로 타이밍최적화를 실행하면 앞에서 언급한 (그림 2.2)와 같은 경우가 생길 수 있다. 이런 경우를 방지하기 위해 이번에는 레지스터를 중심으로 새로운 서브컴퍼넌트를 설정하는 방법을 사용한다(그림 2.6).



(a) before

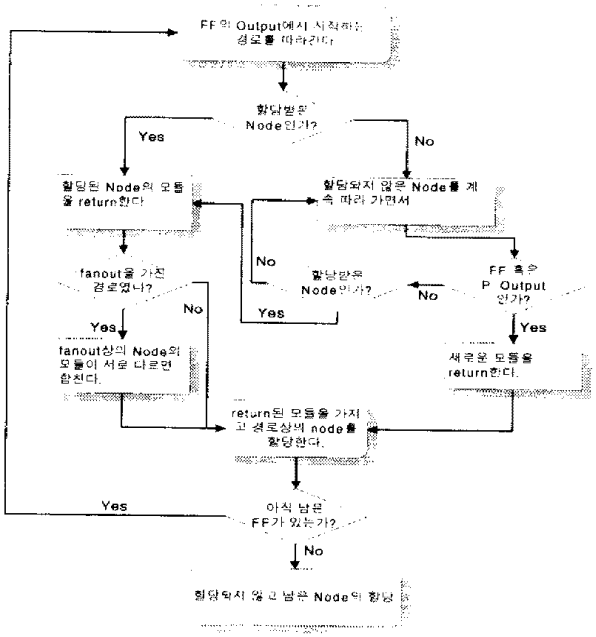


(b) after

(그림 2.6) 서브컴퍼넌트의 재 구성 예

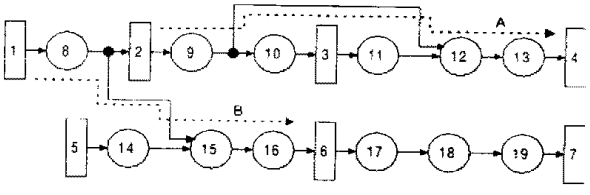
2) 레지스터를 옮길 수 없다는 것은 주 입력에서 주 출력에 이르는 edge의 weight = 0일 경우이거나, 그 레지스터를 중심으로 cycle을 형성하는 경로가 있는 경우를 말한다.

(그림 2.6) (a)와 같이 "path a"의 지연시간이 문제가 될 수 있음은 (그림 2.1)에서 이미 설명한 바와 같다. 이 경우, 설계자가 만든 서브컴퍼넌트사이에 있는 "path a"의 지연 시간을 효과적으로 줄이기 위해서는 레지스터를 경계로 하여 (그림 2.6) (b)와 같이 새로운 서브컴퍼넌트를 형성하는 방법을 취한다. 그렇게 만든 서브컴퍼넌트는 주 입력, 또는 주 출력 edge에만 레지스터가 위치하거나 조합논리회로만으로 구성되기 때문에 첫 번째 접근 방법에서처럼 레지스터 부분을 분할하면 전적으로 조합논리회로로 구성된다. 이 서브컴퍼넌트에 대하여 각각 조합논리회로 타이밍 최적화를 실행하는 방법을 사용한다. 그러므로 이 때 새로운 서브컴퍼넌트를 설정하는 기준은 서브컴퍼넌트사이의 연결은 레지스터를 통해서 이루어지도록 서브컴퍼넌트를 분할하는 것이다. 이런 기준을 가지고 서브컴퍼넌트를 새로이 분할하게 하면 기존의 서브컴퍼넌트사이의 레지스터를 거치지 않고 연결되던 경로를 한 서브컴퍼넌트로 할당하게 된다.



(그림 2.7) 새로운 서브컴퍼넌트 설정을 위한 분할의 흐름도

전체적인 흐름은 (그림 2.7)과 같다. 위와 같은 흐름을 따라갈 때 cycle의 존재 여부와 fanout의 존재 여부가 관건이 된다. 첫 번째, 분할되어야 하는 서브컴퍼넌트의 내부에 cycle을 형성하는 경로가 있지 않다면 나누는 방법을 일관되게 적용할 수 있다. 즉, 경계가 되는 레지스터의 출력에 있는 부분을 기준으로 새로운 서브컴퍼넌트를 형성하는 방법을 사용하면 된다. 그렇게 해서 그 레지스터로부터 시작되는 경로[3]를 따라가면서 할당되지 않은 레지스터를 만날 때까지 확장하여 그 서브컴퍼넌트의 경계로 만들도록 한다. 그리고 다시 그 레지스터의 출력에 있는 부분을 다시 새로운 서브컴퍼넌트의 시작으로 할당한다. 두 번째, cycle이 형성되어 있는 경우엔 cycle이 형성되어 있는 경로에 해당하는 부분을 어느 서브컴퍼넌트에 할당할 것지는 cycle을 형성하고 있는 경로의 특성에 따라 달라진다. 우선 cycle을 형성하고 있는 경로 상에 있는 모든 회로를 같은 서브컴퍼넌트에 할당하도록 한다. 이 방법은 회로의 함수적 행태를 그대로 유지할 수 있기 때문에 타이밍최적화를 수행할 때 유리할 것이다. 여기서 cycle이 형성되는 경로 상에 있는 레지스터는 어느 서브컴퍼넌트에 할당하게 할 것인지는 경로가 단순히 cycle만을 형성하고 있는 단일 경로인지 아니면 그 경로를 포함하는 다른 경로가 존재하는지의 여부에 따라 달라진다. 다른 경로가 있다면 레지스터도 cycle 경로를 할당한 서브컴퍼넌트에 같이 할당될 것이고 그렇지 않은 경우는 같이 할당하지 않아도 무방하다. 세 번째, 경로 상에서 fanout이 발생한 경우인데 이런 경우 각 fanout경로가 여러 서브컴퍼넌트에 할당되어져 있는 경우가 발생할 수 있다. 이런 경우는 다시 이미 할당되어 있는 fanout경로를 거슬러 올라가 해당하는 서브컴퍼넌트를 모두 하나의 서브컴퍼넌트로 합치는 방법을 사용해야한다. 이렇게 하여 나누어진 서브컴퍼넌트별로 레지스터를 떼어내 조합논리회로 부분만을 타이밍최적화를 실행하고 다시 레지스터를 붙이는 과정을 반복한다.

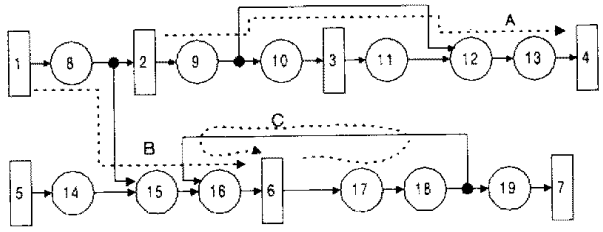


(그림 2.8) 분할을 위한 예제 회로

이를 예제회로로써 설명하면 다음과 같다. (그림

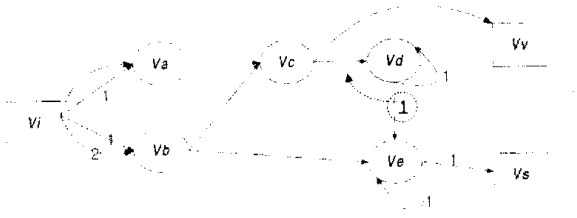
3) 게이트(G)와 연결(C)의 교대 순서를 경로(path)라고 한다. 임의의 경로 P는 $\{C_0, G_0, C_1, G_1, \cdots, G_n, C_{n+1}\}$로 표기한다.

2.8)의 회로 예제는 cycle이 없는 회로이다. 그림에서 보는 것과 같은 경우 각 레지스터가 다른 서브컴퍼넌트에 할당되어 있는 회로라고 하고 번호순서대로 레지스터를 따라가면서 할당한다고 하자. 그러면 우선 1번 레지스터의 출력에서 시작되는 경로를 따라가면서 다시 레지스터를 만날 때까지 확장하여 간다. 이렇게 해서 2번, 6번 레지스터를 만날 때까지 확장하여 하나의 서브컴퍼넌트를 형성한다. 그리고는 2번 레지스터의 출력에서 시작되는 경로를 따라가 3번 레지스터와 4번 레지스터를 만날 때까지 확장하여 다른 하나의 서브컴퍼넌트를 형성한다. 이제 3번 레지스터의 출력에서 시작되는 경로를 따라간다. 여기서 노드 12까지 오면 이 12번 노드는 이미 경로 A를 통해 새로운 서브컴퍼넌트를 할당받은 노드다. 이런 경우 3번에서 시작하여 4번에 이르는 경로중 일부가 이미 할당받은 경우가 되고 이런 경우, 이미 할당받은 서브컴퍼넌트에 같이 할당한다. 이런 식으로 5번 레지스터에서 6번 레지스터에 이르는 경로도 다른 경로 B에 의해 이미 할당된 부분 경로를 가지므로 하나의 서브컴퍼넌트로 같이 할당한다. 그렇게 되면, 노드 8, 14, 15, 16을 포함하는 하나의 서브컴퍼넌트, 노드 9, 10, 11, 12, 13을 포함하는 하나의 서브컴퍼넌트, 그리고 노드 17, 18, 19를 포함하는 하나의 서브컴퍼넌트로 할당받게 된다.

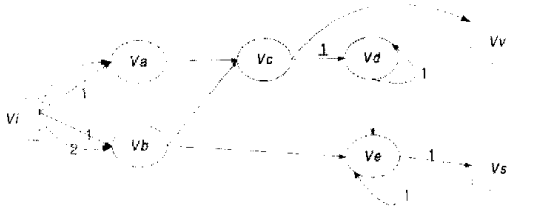


(그림 2.9) 분할을 위한 또 다른 예제 회로

(그림 2.9)는 (그림 2.8)과 유사한 회로이다. 그러나 이 회로엔 노드 16, 17, 18이 거치는 경로 C에 cycle이 존재하고 있다. 그래서 위 회로의 서브컴퍼넌트 할당은 노드 8, 14, 15, 16을 포함하는 서브컴퍼넌트와 노드 9, 10, 11, 12, 13을 포함하는 서브컴퍼넌트는 위와 같지만 노드 17, 18, 19를 포함하는 서브컴퍼넌트는 달라진다. 노드 17, 18을 포함하는 경로 C가 cycle을 이루고 있기 때문에 경로 C에 있는 노드들은 같은 서브컴퍼넌트에 할당해야 한다. 즉, 이 서브컴퍼넌트는 16이 포함되어 있는 서브컴퍼넌트와 합쳐지게 해야 한다. 그래서 위 회로는 노드 8, 14, 15, 16, 17, 18, 19를 포함하는 서브컴퍼넌트와 노드 9, 10, 11, 12, 13을 포함하는 서브컴퍼넌트, 2개로 나누어진다. 이렇게 나누어진 서브컴퍼넌트별로 레지스터를 분리한 조합논리회로 부분만 조합적 타이밍최적화를 실행하고 다시 레지스터를 붙이는 과정을 반복한다. 그리고 타이밍최적화를 실행하고 추가적으로 수행할 수 있는 과정이 있는데, 레지스터를 다시 붙여 retiming 방법을 재적용시키는 것이다. 즉, 떼어낸 레지스터는 각 서브컴퍼넌트의 주 입력 단이나 주 출력 단에 놓여지게 되고 그럴 경우 서브컴퍼넌트 안에서의 최장경로 상에 레지스터를 다시 적절하게 위치를 옮김으로써 최장지연시간을 다시 단축시킬 수 있다. (그림 2.10) (a)를 설명하면, (그림 2.10) (a)가 본 논문에서 제안하고 있는 방법을 따라 조합논리회로 부분의 타이밍최적화를 수행하고 난 뒤 다시 레지스터를 붙인 회로의 그래프라고 하자. 이 회로를 다시 retiming이 가능한가를 조사하여 보면, 다시 붙인 레지스터들이 주로 주 입력/출력 edge에 있게 되고 서브컴퍼넌트 내의 최장경로 상에도 놓여있게 되고 최장경로상의 레지스터를 조합논리회로 부분으로 다시 옮길 수 있어 해당 서브컴퍼넌트의 지연시간은 더욱 줄어들게 된다. 즉, (그림 2.10) (a) 그래프에서 (Vd, Ve) 경로 상에 있는 레지스터를 (Vc, Vd) 경로로 옮겨 (그림 2.10) (b)와 같은 그래프를 얻을 수 있으며, 이런 경우 더 좋은 타이밍 최적화의 결과를 얻을 수 있다.



(a) Before retiming



(b) After retiming

(그림 2.10) 새 분할에 의한 조합적 타이밍최적화 후의 retiming 예제

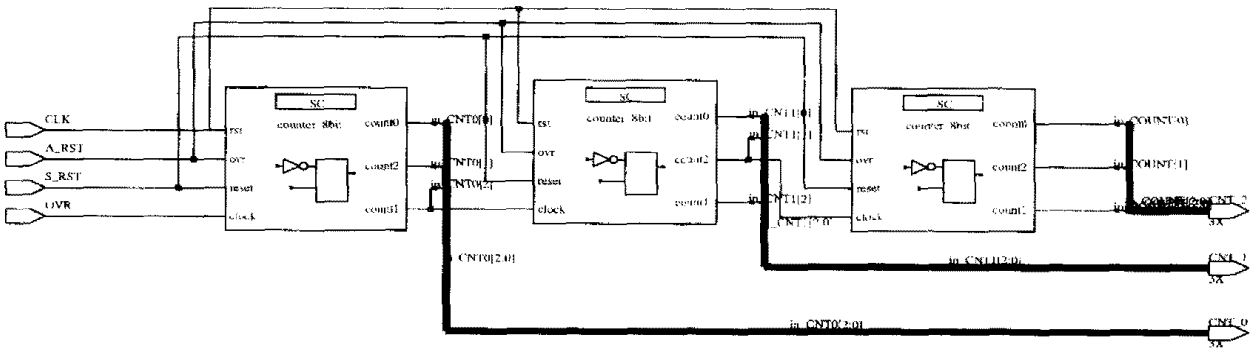
3. 실 험

본 논문에서 새롭게 제안된 방법의 효능을 실험적으로 검증하기 위하여 제안된 알고리즘을 툴로 구현하여 실험을 수행하였다. 실험환경 구축은 한국전자통신연구원에서 개발한 자동설계 시스템인 LODECAP(Logic Design Capture)[8]을 이용하였다.

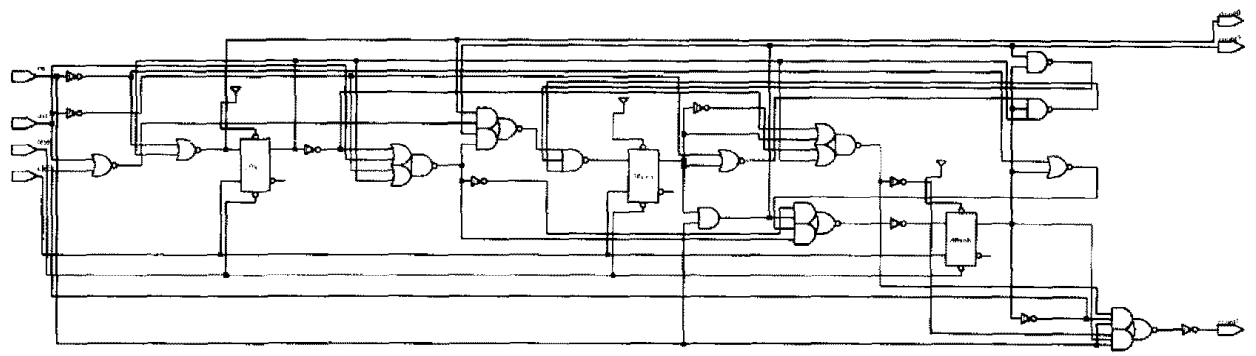
본 논문에서 실험을 위해 사용한 예제 중 하나가 cascaded counter로서 이의 구조는 (그림 3.1)과 같고 그 중 한 서브컴퍼넌트의 내부는 (그림 3.2)와 같다. 이 회로는 VTI사의 CMOS gate array(VGT350)[9] 기술 라이브러리로 각 서브컴퍼넌트가 합성되어져서 mif 형태[1][8] (그림 3.3)로 표현되어 있다.

(그림 3.3)에서 보는 봐와 같이 [TITLE] 다음에 나오는 CELL counter_8cascade는 이 회로의 이름이고, 다음의 CONT cnt_0[0] in_cnt0[0] OUT은 회로의 주출력에 해당한다. 각 게이트는 [BODY]안에서 CELL in01d1 s6/s3/s128와 같은 형태로 정의되며, in01d1은 어떤 종류의 게이트인가를 s6은 그 게이트의 식별인자이며, /s3/s128은 이 회로가 계층구조를 가지고 있다는 것을 알려주고 있다. 즉, 이 정보를 가지고 서브컴퍼넌트를 나눌 수 있게 되고 새로운 서브컴퍼넌트를 설정할 때도 유용하게 사용되는 정보이다. 여기서 s28은 게이트가 속한 서브컴퍼넌트의 식별인자이며, s128은 서브컴퍼넌트 안에서의 식별인자가 된다. 게이트의 입/출력은 CONT zn w129 OUT /s3/w127로 정의되며 zn은 매핑된 라이브러리에서 제공하는 입/출력 이름이며, w129는 입/출력 단자와 연결되어 있는 선(wire)의 식별인자이며, OUT은 zn이 출력이라는 것은 나타내며, /s28/w127은 위 게이트의 설명 때처럼 계층정보를 알려주는 것이다.

주어진 회로의 실험 결과는 다음의 <표 3.1>과 같다.



(그림 3.1) 실험대상 예제회로 : cascaded counter



(그림 3.2) (그림 3.1)의 cascaded counter의 서브컴퍼넌트의 내부

1) 한국전자통신연구원 LODECAP 내부 네트리스트 형식

```
[TITLE]
CELL counter_8cascade
CONT CNT_0[0] in_CNT0[0] OUT
CONT CNT_0[1] in_CNT0[1] OUT
...
CONT CNT_1[1] in_CNT1[1] OUT
CONT CNT_1[2] in_CNT1[2] OUT
CONT OVR w108 IN
CONT S_RST w106 IN
CONT A_RST w105 IN
CONT CLK w103 IN
[BODY]
CELL in01d1 s6 /s3/s128
CONT i in_COUNT[2] IN /s3/w108
CONT zn w129 OUT /s3/w127
...
CELL dfbtnb s16 /s3/s118
CONT d w145 IN /s3/w122
CONT cp w103 IN /s3/w131
CONT sdn w144 IN /s3/w132
CONT cdn w105 IN /s3/w130
CONT q w138 OUT /s3/w107
CONT qn w146 OUT /s3/w140
...
CELL in01d1 s36 /s2/s127
CONT i w159 IN /s2/w114
CONT zn in_CNT1[1] OUT /s2/w115
...
CELL oa03d1 s63 /s2/s100
CONT c w168 IN /s2/w124
CONT b2 w180 IN /s2/w125
CONT b1 w160 IN /s2/w126
CONT a2 w158 IN /s2/w127
CONT a1 in_CNT0[2] IN /s2/w117
CONT zn w185 OUT /s2/w128
CELL in01d1 s64 /s1/s128
CONT i in_CNT0[2] IN /s1/w108
CONT zn w187 OUT /s1/w127
...
[END]
```

(그림 3.3) (그림 3.2) 회로의 네트리스트 표현 형식

〈표 3.1〉 (그림 3.1) 회로의 실험 결과

	원래 회로	구조 유지	새 구조 형성
지연 시간	28.61	24.99	22.81

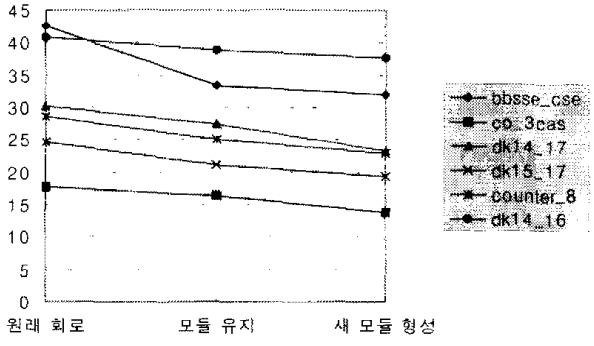
(그림 3.4)는 초기 회로의 최장지연시간을 얻기 위하여 정적 타이밍분석기(static timing analyzer) dtv[8]를 수행시켜 얻어진 결과이다. 보는 바와 같이 초기 회로의 최장지연시간은 **28.61**이다. 그리고 우선 설계자가 설정한 계층구조의 서브컴퍼넌트를 유지한 채 실행하는 LODECAP의 타이밍최적화를 실시하여 얻은 최장지연시간은 **24.99**였다. 최장지연시간은 초기 회로보다 12.7 % 줄었으나 이 최장경로는 서브컴퍼넌트간에 걸쳐 있는 경로이다. 이것은 새로운 레지스터의 경계를 이용하여 서브컴퍼넌트를 새로이 설정하여 타이밍 최적화를 실행하였을 때 더 좋은 결과를 기대할 수 있게 하고 실제 결과도 **22.81**로 20.3% 감소하였다. 이는 최장경로가 새로운 서브컴퍼넌트 내에 있는 경로 상에서 나타나게 되어 좀 더 좋은 결과를 보장할 수 있게 된 결과이다. 그 외 다른 회로들의 실험결과는 <표 3.2>와 (그림 3.5)와 같다.

```
[PATH]
[PORT] RST          F 0.00
q  @ /s3/s130       R 3.36
z  @ /s3/s135       R 5.51
zn @ /s3/s139       F 6.72
zn @ /s3/s142       R 14.62
zn @ /s1/s102       F 15.07
zn @ /s1/s108       R 18.61
zn @ /s1/s111       F 19.07
zn @ /s1/s115       R 24.13
zn @ /s1/s117       F 24.58
zn @ /s1/s139       R 27.23
zn @ /s1/s154       F 27.79
zn @ /s1/s158       R 28.61
[PORT] Z[4]         R 28.61
[ENDPATH]
```

(그림 3.4) 초기 회로의 타이밍 분석 결과

〈표 3.2〉 실험 결과

	원래 회로	모듈 유지	새 모듈 형성
bbsse_cse	42.55	33.26	32.06
co_3cas	17.71	16.48	13.90
dk14_17	30.24	27.50	23.28
dk15_17	24.69	21.21	19.25
counter_8	28.61	24.99	22.81
dk14_16	40.87	38.86	35.75



(그림 3.5) 실험 결과

보이는 바와 같이 최장지연시간이 서브컴퍼넌트간의 경로에서 나타나는 경우 본 논문에서 제안한 방법으로 타이밍 최적화를 실행한 경우 기존의 방법에 의한 결

서부터 얻을 수 있음을 알 수 있다.

4. 결 론

본 논문에서는 아키텍춰 수준에서 계층 구조를 가지는 회로 구조에 기존의 조합적 타이밍최적화 방법을 적용함으로서 발생하는 문제점들을 해소시키기 위하여 회로 구조에 존재하는 레지스터를 순차논리회로인 서브컴퍼넌트로부터 효과적으로 분할하는 방법을 제시하고 있다. 접근 방법은 우선 설계자가 설계한 계층 구조를 유지시키는 방법을 먼저 취하는데, 이 방법은 기존의 retiming 방법과 peripheral retiming 방법을 응용하여 서브컴퍼넌트 내 조합논리회로 부분을 확대하는 방법을 이용한다. 이와 같은 방법에 의하여 타이밍최적화가 좋은 결과를 가져오지 못할 때에는 다른 접근 방법으로서 기존의 서브컴퍼넌트들로 이루어지는 경계를 새로운 경계를 가지는 새로운 서브컴퍼넌트들로 변형시켜 서브컴퍼넌트들 각각의 독립적인 타이밍최적화로 전체 회로에 대한 타이밍최적화를 이끌어 낼 수 있도록 하는 방법을 제시하였다.

기존의 연구들이 비 계층구조를 가지는 순차적, 혹은 조합적 논리회로에서의 타이밍 최적화를 위한 방법으로 연구가 집중되어 온 반면, 본 논문은 아키텍춰-수준에서 계층적 구조를 가지는 회로에 대한 새로운 접근을 시도하고 있는데, 설계되는 회로가 크고 복잡해짐에 따라 설계자가 실제 회로를 대부분 서브컴퍼넌트화 계층적 구조를 가지는 구조로 설계하는 것을 고려해 볼 때 이의 효능성은 매우 크다는 것을 알 수 있다.

참 고 문 헌

[1] K. J. Singh, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli, *"Timing Optimization of Combinational Logic,"* in Proc. of ICCAD, pp. 282-285, 1988.

[2] Y. Kukimoto, R. K. Brayton, and P. Sawkar, *"Delay-Optimal Technology Mapping by DAG Covering,"* in Proc. of DAC, pp.348-351, 1998.

[3] D. S. Kung, *"A Fast Fanout Optimization Algorithm for Near-Continous Buffer Libraries,"* in Proc. of DAC, pp.352-355, 1998.

[4] HDL Coding Style (in FPGA Design Ultrapack CDROM), Actel 1998.

[5] *Logic Synthesis Using Synopsys*, P. Kurup and T. Abbasi, Kluwer Academic Publishers 1995.

[6] C. E. Leiserson and J. B. Saxe, *"Retiming Synchronous Circuitry,"* in Algorithmica, Vol.6, pp. 5-35, 1991.

[7] S. Malik, E. M. Sentovich, R. K. Brayton, and A. Sangiovanni-Vincentelli, *"Retiming and Resynthesis : Optimizing Sequential Networks with Combinational Techniques,"* in IEEE Trans. on CAD/ICAS, Vol.CAD-10, No.1, pp.74-84, Jan., 1991.

[8] *LODECAP Ver. 2.0*, 반도체연구단, 한국전자통신연구소, 1996.

[9] *VGT350 LIBRARY*, VLSI Technology INC., Nov., 1991.

이 대 희

1995년 2월 부산대학교 컴퓨터공학과 졸업(공학사)

1997년 2월 부산대학교 대학원 컴퓨터공학과 졸업(공학석사)

1997년 3월~현재 삼성전자 근무

관심분야 : VLSI 설계 및 CAD

양 세 양

e-mail : syyang@hyowon.pusan.ac.kr

1981년 2월 고려대학교 전자공학과 졸업(공학사)

1985년 2월 고려대학교 대학원 전자전산기공학과 졸업(공학석사)

1990년 5월 메사츄세츠대학교 대학원 전기컴퓨터공학과 졸업(공학박사)

1990년 2월~1991년 2월 Microelectronics Center of North Carolina(MCNC) VLSI-CAD 그룹, 선임연구원

1991년 3월~현재 부산대학교 컴퓨터공학과 재직(현 부교수)

관심분야 : VLSI CAD, VLSI testing, FPGA 응용