캐리-세이브 가산기에 기초한 연산 하드웨어 최적화를 위한 실질적 합성 기법

(A Practical Synthesis Technique for Optimal Arithmetic Hardware based on Carry-Save-Adders)

김 태 환 † 엄 준 형 ††
(Taewhan Kim) (Junhyung Um)

요 약 캐리-세이브 가산기 (CSA)는 빠른 수행과 작은 면적을 가지는 연산 하드웨어 구현에서 가장 효과적으로 사용되는 연산 셀들 중의 하나이다. 현재 CSA 적용기술의 근본적인 약점은, 그 적용이 덧셈식으로 직접 변환되는 부분에 해당하는 회로에만 가능하다는 것이다. 이러한 제한점을 극복하기 위하여, 우리는새로운 몇가지 CSA 변환 기법들을 제안한다. 구체적으로 멀티플렉서를 포함한 연산에서의 CSA 변환, 다수 회로를 포함한 연산에서의 CSA 변환, 곱셈연산을 내포한 연산에서의 CSA 변환을 제안한다. 또한, 이러한 기법들을 실제의 회로 합성에서 효과적으로 적용하는 통합 알고리즘을 제안한다. 우리는 다양한 실험을 통하여, 제시된 기법들에 기반한 우리의 알고리즘이 기존의 CSA 방법들과 비교하여 실제적인 회로 합성에서 매우 효율적임을 보인다.

Abstract Carry-save-adder (CSA) is one of the most effective operation cells in implementing an arithmetic hardware with high performace and small circuit area. An fundamental drawback of the existing CSA applications is that the applications are limited to the local parts of arithmetic circuit that are directly converted to additions. To resolve the limitation, we propose a set of new CSA transformation techniques: optimizing arithmetics with multiplexors, optimizing arithmetics in multiple designs, and optimizing arithmetics with multiplications. We then design a new CSA transformation algorithm which integrates the proposed techniques, so that we are able to utilize CSAs more globally. An extensive experimentation for practical designs are provided to show the effectiveness of our proposed algorithm over the conventional CSA techniques.

1. 서 론

회로 최적화는 합성(synthesis)의 여러 단계에서 이루어진다. 본 논문은 회로의 지연시간 최소화 방법에 대해 다룬다.상위단계 합성(high-level synthesis)[1]에서, 지연시간은 컨트롤 스텝의 개수와, 사이클 시간의 두 가지를 의미한다. 스케쥴링(scheduling)을 수행할 때에는 자원의 제약 하에 지연시간을 최소화하거나, 지연시간 제약 하에 필요한 자원의 양을 최소화한다. 자원 할당/결합(resource allocation/binding) 단계에서는 주어진 제약시간을 만족하는 한계 내에서 회로의 데이타 플로우 그래프(CDFG)를 수행하기 위해 자원을 할당하고 결합한다. 한 연산에 대해 다른 종류의 하드웨어를 할당할 경우 CDFG는 다른 수행시간을 가질 수 있고, 이는 결국, 다른 지연시간을 갖는 회로를 생성하기 때문에, 지연시간을 충분히 최적화 하기 위해, 그리고 "실현 가능한" 지연시간을 갖는 회로를 만들기 위해 이상적으로는 자원의 할당/배정 문제를 스케쥴링 단계에서 고려하는 것이 바람직하다. 그러나, 실제적으로 이는 너무나 많은 계산양을 요구하게 되고, 따라서, 스케쥴링 단계에서는 각 연산이 특정한 자원에 할당되어 있다고 가정하고 스케쥴링을 수행하며, 자원 재할당 문제를 다음 단계인 할당/결합 단계에서 고려한다. 결과적으로, 이러한 방법 때문에 상위 단계에서 합성된 회로의 지연시간은 실제 지연시간

· 본 논문은 첨단정보기술 연구센터(AITrc)를 통하여 과학재단의 지원을 받았음.
† 종신회원 : 한국과학기술원 전자전산학과 전산학전공 교수
tkim@cs.kaist.ac.kr
†† 학생회원 : 한국과학기술원 전자전산학과 전산학전공
jhum@visisyn.kaist.ac.kr
논문접수 : 2000년 4월 17일
심사완료 : 2001년 8월 16일

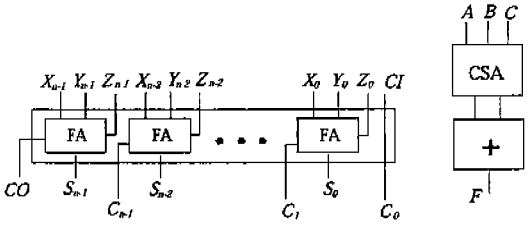
보다 길게 측정되거나 짧게 측정되는 잘못된 결과를 초래할 수도 있다. 지연시간 초과로 측정되었을 경우, 이는 스케줄러가 너무 느린 자원을 가정한 경우이고, 수행을 쓸모 없이 낭비하는 결과를 가지고 온다. 반대로, 지연시간이 실제의 지연시간보다 적게 측정되었을 경우 이는 스케줄러가 너무 빠른 자원을 가정한 경우이며, 자원 공유와 FSM의 지연시간으로 인해 생기는 멀티플렉서의 지연시간과 같은 지연시간 오버헤드(overhead)를 위한 충분한 여유 지연시간을 갖지 못하는 결과를 가지고 올 수 있다. 여기서 주목할 점은, 최종적인 회로의 사이클 시간은 대개의 경우 주어져 있기 때문에 지연시간을 실제의 시간보다 적게 측정하는 경우가 더욱 심각한 문제가 된다.

RT-단계 합성(register-transfer-level synthesis)에서의 입력은 상위단계 합성의 출력이거나, RT-단계의 코드로 주어진다. 두 경우 모두 회로 지연시간을 주어진 사이클 시간 제약에 만족되도록 최적화를 수행한다. 이 단계에서의 연산의 지연시간 최적화를 위해서는 여러 가지 방법이 있으며, 대표적으로 자원 선택과 연산트리 밸런싱을 들 수 있다.

논리-단계 합성(logic-level synthesis)에서는, RT-단계 합성을 통해 지연시간 제약이 여전히 만족되지 않았을 경우 논리 게이트 단위의 최적화가 수행된다. 네트워크 재구조화[2,3]나 게이트 사이징[4,5] 등의 방법이 주로 사용되다. 대개의 경우, 지연시간을 초과하는 정도가 높을 경우 상당히 긴 합성시간을 소비하며, 생성된 회로의 면적 또한 상당히 크다.

연산회로 최적화에 대해서는 많은 연구가 제시되어 있다[6,7,8,9,10,11,12]. [6]에서는 고속의 선형(linear) 계산 회로를 생성하기 위해 여덟 가지의 대수적인 변환을 제시하였다. [7]은 연산트리의 가장 긴 수행 경로를 줄이기 위해 연산 트리에서의 반복되는 연산식을 발견하기 위한 방법으로 인수분해방법을 적용하였다. 또한, 배분법칙을 포함한 경험적 변환 법칙을 적용한 방법이 [8]에 소개되고 있다. [9,10]은 피연산자의 임의의 도달시간을 가정하고 분배법칙과 교환법칙을 이용하여 연산트리 수행시간 감소 방법을 제시하였고, 덧셈과 뺄셈, 그리고 쉬프트(shift) 연산이 혼합된 연산트리에서의 수행시간 최소화 문제 해결에 응용하였다. [11]은 회로 성능의 향상을 위해 연산 단계에서 적용될 수 있는 상위단계의 많은 변환 방법을 소개하였다. 여기서는 동일한 회로면적 하에서 회로속도를 향상시키거나 동일한 지연시간 하에서 회로면적 또는 전력소모를 감소시키는 상위단계에서의 변환의 역할이 중요함을 강조하고 있다. [12]는 회로면적과 전력소모를 최소화하기 위해 규칙성을 이용한 대수적인 변환과 steepest decent에 기초한 연산변환 기법을 제시하고 있다. 위에 제시된 모든 방법들은 연산 최적화를 위해 캐리-세이브 가산기의 사용을 고려하지 않는 반면에, [14]는 기존의 방법들과는 달리 새로운 즉, 캐리-세이브 가산기 (CSA)[13]를 이용한 새로운 방법을 제안하였다. 이 방법은 회로와 사이클 시간이 주어졌을 때 연산 단계에서 지연시간을 최소화 한다는 점에서 RT-단계 합성에 해당되며, 상당한 효율을 보임이 [13]에 나타나 있다.

그러나, [14]에 제시된 방법은, 덧셈식으로 직접 변환되는 비교적 단순한 회로에만 적용이 가능하다는 단점을 가지고 있다. 실제로 산업 현장에서 쓰이는 많은 회로들은, 단순한 연산식으로 주어져 있지 않으며, 멀티플렉서를 포함하거나 다수의 계층을 갖는 연산식이거나 곱셈등을 포함하는 복잡한 구조를 갖는것이 보통이다. 이러한 회로들에 대해 [14]에 제시된 방법을 적용 가능하게 하기 위해, 우리는 새로운 몇 가지의 CSA변환 기법을 본 논문에서 제안하였다. 우리가 본문에서 제시하는 알고리즘은 기존의 [14]에서 제시된 방법과 간단하게 통합될 수 있으며, 기존의 방법으로는 CSA로 변환되지 못했던 회로의 여러 부분에 적용이 가능하기 때문에 보다 나은 성능의 회로로써 RT-단계에서 변환이 가능 하다는 데에 그 장점이 있다.



(a) (b)

그림 1 n-비트 CSA의 구조와 사용 예

우선, CSA의 구조와 기능에 대해 살펴보자. 그림 1(a)는 n-비트 CSA의 구조를 보이고 있다. n-비트 CSA는 n개의 서로 독립적인 full adder(FA)로 구성되어 있다. 이는 세 개의 n-비트 입력을 가지며, 두개의 출력을 생성한다. 일반적인 가산기(예를 들어, 리플-캐리 가산기(ripple-carry adder)나 캐리-예측 가산기(carry-lookahead adder))와는 다르게 CSA 내부에는 캐리 지연이 없다. 그러므로, CSA는 한개의 FA와 같은 지연시간을 갖는다. 그리고 이는 n의 크기와 관계없이

항상 일정한 값을 갖는다. 결과적으로, 충분히 큰 n에 대해 CSA는 보통의 가산기보다 훨씬 빠르고 또한 비교적 작은 회로면적을 갖게 된다.

우리는 CSA 트리를 CSA 연산기들과 하나의 일반적인 가산기로 이루어진 트리로 정의한다. CSA 트리는 임의의 주어진 덧셈 연산을 변환하여 두개의 피연산자를 생성하며, 이는 최종적으로 보통의 일반적인 가산기의 입력으로 연결된다. 예를 들어, 그림 1(b)에서 보이는 바와 같이 연산식 는 하나의 CSA와 하나의 일반적인 가산기의 트리로 변형되며, CSA는 A, B, C를 입력으로 갖고 두개의 CSA 출력이 일반적인 가산기의 입력으로 연결된다.

기존의 CSA 변환[14]은 다음 세 단계로 진행된다: (1) 우선, 변환될 (부분) 연산 트리를 회로 내에서 추출한다; (2) 추출된 트리를 덧셈 연산으로 변환시킨다; (3) 덧셈 연산을 CSA 트리 연산회로로 변환시킨다. 최소화하고자 하는 입력 연산회로가 non-cyclic CDFG로 주어졌을 때, 기존의 알고리즘은 더 이상의 변형될 트리가 발견되지 않을 때까지 위의 세단계를 반복하게 된다

위에 설명된 CSA 변환은 덧셈식에만 제한되지 않는다. 뺄셈이나 곱셈같은 모든 연산이 덧셈으로 변환되어 CSA 변환이 적용될 수 있다. 예를 들어 뺄셈의 경우, 연산식 는 로 변형될 수 있다. (이 경우, 2's complement가 이용되었다.) 또한, 곱셈식에도 CSA변환을 적용하기 위해 다음과 같은 방법이 사용된다. 우선, 곱셈은 두 가지 방법으로 분해된다: (1) 곱셈을 덧셈들의 합으로 완전히 분해하는 방법과 (2) 곱셈에 대한 Wallace 트리 합성 모델을 하나의 부분적인 곱의 모듈과 하나의 덧셈으로 분리하는 방법이다. (우리는 를 부분적으로 분해한 곱셈 연산자를 나타내기 위한 기호로 사용한다.) 이와 같은 두 가지 방법 외에, 만약 곱셈이 booth encoding 방법으로 합성되었다면 CSA 변환의 입력 피연산자로 booth encoder에 의해 생성된 부분적인 곱을 이용할 수 있으며, 이는 선택-1과 선택-2의 중간적인 변형 형태가 된다.

그러나, [14]에서는 연산트리에 곱셈이 포함되었을 경우 곱셈이 연산 트리의 가장 하단부에 위치 하였을때만 최적화가 가능하고, 이와 같은 단점을 해결하기 위해 본 논문에서는 곱셈을 배분법칙을 이용하여 연산트리를 변형한 후 CSA최적화를 수행하는 방법을 제시한다. 또한, 우리의 CSA 기법은 일반적인 회로에서 자주 나타나는 멀티플렉서(multiplexor)를 가진 연산 트리 그리고 다중설계 (multiple design)를 포함한 연산 트리등 기존의 방법으로는 적용이 용이하지 않은 [14]의 CSA 변환의 단점을 극복한다. 다시 말하면, 우리는 다음의 세 가지의 CSA 기법: (a) 멀티플렉서를 가진 연산 트리의 최소화; (b) 다중설계를 포함한 연산 트리의 최소화; (c) 곱셈을 포함한 연산 트리의 최소화를 이용하여 전체적 회로에 대해 기존에 제시된 CSA변환을 충분히 적용할 수 있도록 위의 세가지 기법들을 잘 혼합하여 사용할 수 있는 알고리즘을 제시한다.

2. 새로운 CSA 기법들

본 단락에서는 먼저 새로운 CSA 최적화 기법들에 대해 소개하고 있다. 이들을 함께 혼합하여 사용하는 전반적 알고리즘은 단락 3에 제시한다.

2.1 멀티플렉서를 가진 연산 트리의 최소화

회로 설계 내의 조건문 (예: if, case)이나, 자원 공유(resource sharing)는 회로 내에 멀티플렉서를 유도하게 된다. 그림 2(a)와 (b)는 각각 조건문에 대한 VHDL 회로 설계와 그에 따른 변환된 회로 그래프 보여준다. 그림 2(b)의 점선은 회로의 가장 긴 수행경로(critical path)를 나타낸다. 우리는 이 경우 CSA 변환은 다음의 세 단계로 이루어진다.

* 단계 1 (연산 올림): 주요경로 위의 연산기를 그림 2(c)에서 보이는 바와 같이 멀티플렉서 위로올려, 새로운 연산 트리를 생성한다. 이 경우 회로면적은 증가하지만 회로의 지연시간에는 변함이 없다.

* 단계 2 (트리 변형): 주요경로 위의 조건분기(conditional branch)에 정의된 연산 트리는 그림 2(d)에서처럼 CSA 트리로 변환된다(즉, tree 1). 이때, 오른쪽의 조건 분기가 또한 연산 트리를 이루면 (즉, tree 2), 이 또한 CSA 트리로 변환된다. 이러한 변환은 회로의 지연시간이나 회로면적을 감소시킴과 동시에 다음단계에서 야기될 수 있는 회로 면적 증가를 최소화시키는데 도움을 준다.

* 단계 3 (연산 내림): 조건분기에 변환된 트리의 마지막 가산기를(즉, op1, op2) 멀티플렉서를 경유하여 아래로 내리고, 그림 2(e)에서 보이는 것처럼 통합한다. 이 과정에서 멀티플렉서는 추가로 하나 더 생성되지만, 회로의 지연시간은 그대로 유지된다.

2.2 다중설계를 포함한 연산 트리의 최소화

연산식을 포함한 아주 복잡한 회로 설계는 연산이 여러 개의 부분 회로로 나누는 것은 자연스러운 설계이다. 예를 들어, 그림 3(a)는 두개의 회로 A,B를 가지고 있는 연산 회로를 보여준다. 기존의 CSA 변환을 두개의 회로에 차례로 적용한 결과는 그림 3(b)에 보여진다. 결과적으로, 두개의 덧셈 연산기중 하나는 부분회로 A에 위치해 있고 또 다른 하나는 부분회로 B에 위치해 있

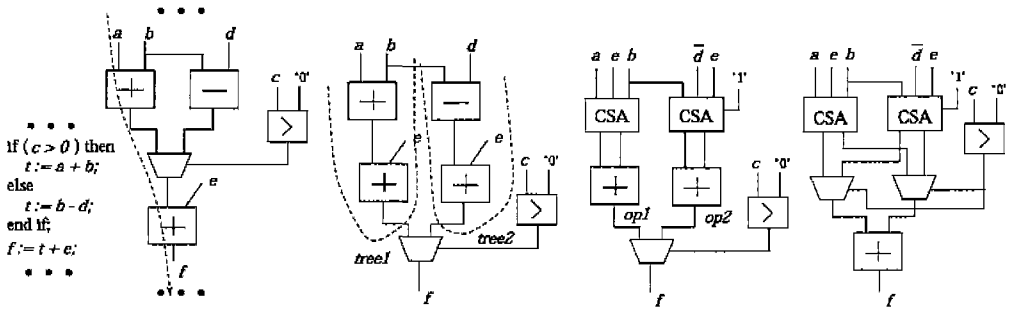


그림 2 멀티플렉서를 포함한 연산회로의 CSA 변환의 예: (a) 조건분기를 포함한 VHDL 코드의 부분; (b) (a)에 대한 회로; (c) 단계 1 적용으로 생성된 회로; (d) (c) 회로에 단계 2 적용으로 생성된 회로; (e) (d) 회로에 단계 3 적용으로 생성된 회로

다. 이는 만약 연산식이 n개의 회로에 체인처럼 연결되어 있다면 이러한 기존의 CSA 적용은 각 회로에 하나씩의 덧셈을 생성하여 전체적으로 n개의 덧셈을 생성할 수 밖에 없다. 따라서, 우리의 목적은 이러한 (캐리-지연이 있는) 덧셈들을 CSA들로 변환시켜 이 덧셈의 개수를 1개까지 줄이자는 것이다. 이를 위하여 우리는 캐리-벡터 포트라 불리는 여분의 특별한 포트를 이용한다.

그림 3(c)는 주어진 포트 r외에 또 다른 포트 즉, 캐리-벡터 포트 s를 할당한 예이다. 초기에는 포트 s의 값이 0으로 주어진다. 결과적으로, 우리는 회로 A의 연산트리에 대해 마지막 일반적인 가산기를 사용하지 않고 CSA 트리를 생성할 수 있고, 두개의 출력, 즉 마지막 CSA의 두 출력을 r과 s에 연결함으로써 변형된 최종적인 CSA 트리를 생성하게 되어, 트리의 마지막에 하나의 가산기만이 할당되었다. 따라서, 기존의 회로 지연 시간에 비해 작은 지연시간을 갖는 연산트리를 생성하게 된다. 여기서, 우리가 각 부분 회로의 연산 트리를 추출하기 위해 사용한 알고리즘과 추출된 연산 트리를 덧셈식으로 변환하기 위해 사용한 알고리즘은 [14]에서 제시한 것을 이용한다.

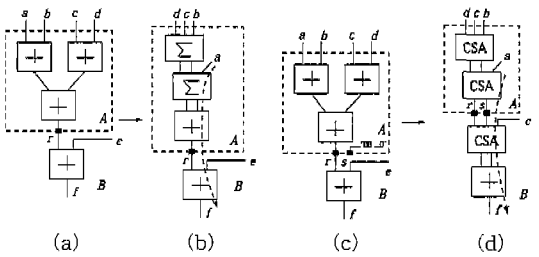


그림 3 여분의 한 포트를 이용한 CSA 변환과 기존의 CSA 변환의 예

2.3 곱셈을 포함한 연산 트리의 최소화

CSA들로 변형되기 위해 선택된 연산트리는 곱셈연산을 포함하는 경우가 많다. 이 경우, 트리 추출에서 합당한 연산트리가 되기 위해서는 곱셈는 트리의 마지막 노드에 위치해야 한다는 조건이 있다[14]. 그림 4(a)는 곱셈을 포함한 기존의 연산트리 추출의 예를 보여준다. 그림 4(a)의 트리중 op1과 op2이 추출되고, 따라서 그림 4(b)의 CSA 트리로 변형되었다. (여기서, 는 두개의 출력을 가지는 부분적인 곱셈 연산을 나타낸다.) 결과적으로 기존의 CSA 변환을 그대로 적용할 경우, op1, op2, op3를 모두 함께 CSA로 변형할 수 없다.

이러한 경우, CSA 변환의 적용 범위를 넓히기 위해 회로의 주요 경로 위에 곱셈 연산이 있다면 연산식에 대해 배분법칙을 적용하는 것이 필요하다. 예를 들어, 그림 4(a)의 원래 연산식에 배분법칙을 op2과 op3에 적용하여 그림 4(c)와 같이 재구성할 수 있으며, 이로부터 주요 경로 위의 모든 연산들을 함께 CSA로 변환할 수 있다. 주의할 점은, CSA 변환으로 인한 회로 면적의 증

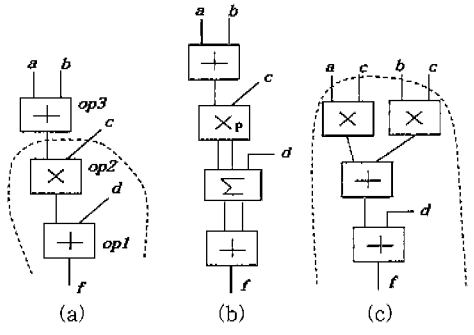


그림 4 곱셈을 포함한 CSA 변환을 보여주는 예

가가 클 수 있으므로, 이런 배분법칙을 이용한 CSA 변환 기법은 곱셈이 상수 입력을 가질 경우에 적당하다. 그러나, 실제적으로 이러한 변환이 부득이 사용되어야 하는 경우에는, 한 예로서, 지연시간 제약이 엄격한 회로의 경우, 제안한 기법이 매우 유용하다.

3. 통합 알고리즘

우리의 통합 알고리즘의 목적은, [14]에서 제시된 CSA 기법이 단순한 연산트리에만 적용 가능하기 때문에 앞의 단락에서 제시된 새로운 CSA 변환 방법을 혼합하여 주어진 회로내의 연산 트리를 찾아 CSA를 적용하는 과정을 좀더 광범위하게 수행함으로써, 통합적인 최적화를 하여 궁극적으로 최소 지연시간을 가지는 회로를 만드는 것이다. 본 단락에서는, 앞 단락에서 제시된 새로운 CSA 적용기법의 여러 가지 조합을 알아보고, 이를 사용하기 위한 단계적 향상 알고리즘을 제안한다.

3.1 사용될 CSA 변환 기법들

본 논문에서 제시하는 알고리즘에서는 단락 2의 CSA 변환기법들과 기존의 CSA 변환기법을 이용한 6가지의 CSA 변환기법을 사용한다:

* 타입 1 : 기존의 CSA 기법[14] (변환될 CSA 트리는 멀티플렉스를 가지지 않아야 하고, 하나의 설계안에 있어야 하며, 곱셈은 트리의 leaf 노드에 위치해야 한다.)

* 타입 2 : 2.1 단락에서 소개된 멀티플렉서를 포함한 연산 최적화

* 타입 3 : 2.2 단락에서 소개된 다중 설계를 포함한 연산 최적화

* 타입 4 : 2.3 단락에서 소개된 곱셈을 포함한 연산 최적화

* 타입 5 : 타입 4의 수행된 후, 타입 1의 수행

* 타입 6 : 타입 4의 수행된 후, 타입 3의 수행

여기서, 타입 3과 타입 4를 함께 사용하는 타입의 변환은 사용되지 않는다. 이는 거의 모든 실질적 회로에서 멀티플렉서를 가진 회로의 데이타 및 컨트롤 흐름은 회로 설계자 측면에서 쉽게 알 수 있도록 멀티플렉서와 관련된 모든 연산 부분이 하나의 회로 안에 놓여지는 것이 상례이기 때문이다.

3.2 회로 최적화 알고리즘

회로의 가장 긴 수행경로에 있는 연산들에 대해, 우리는 각 변환 타입 k()에 대해 적용 가능한 연산 트리들을 모두 추출한다. 각 변환 타입 k에 대해, 모두 개의 변환 가

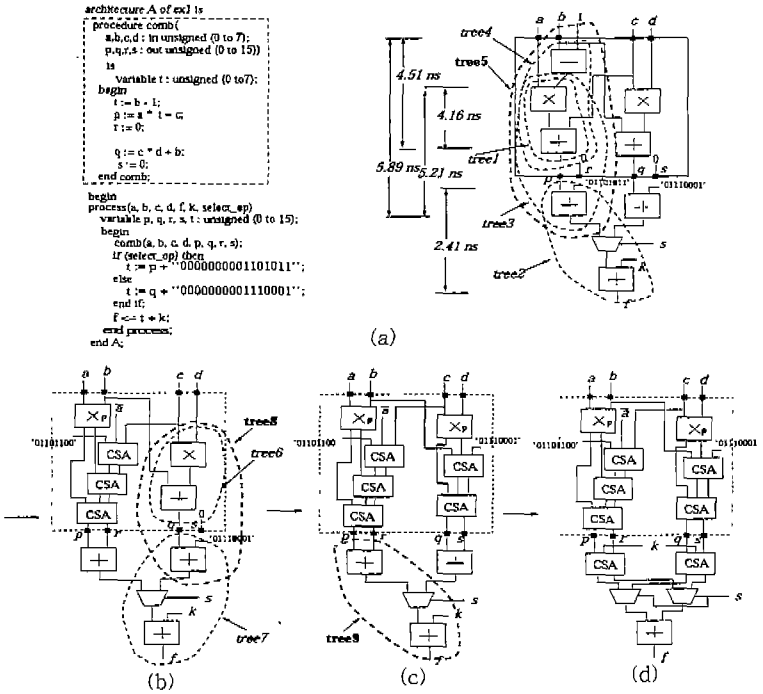


그림 5 논문에 제시된 알고리즘의 흐름을 보여주는 예

능한 부분 연산 트리가 존재한다고 하자. 우리는 개의 후보 중에 가장 좋은 부분 연산 트리를 선택하기 위한 비용함수 C를 정의한다. 각 개의 후보들에 대해, 우리는 그에 대응하는 변환 타입 k를 적용하고 다음의 비율 C = (지연 시간의 감소량)/(회로 면적의 증가량)을 계산한다.

이러한 비율은 회로 면적의 증가에 대한 회로 지연시간 감소의 효율성을 나타낸다. 즉, 비용 C는 회로 면적을 그다지 증가시키지 않으며 회로의 지연시간을 최소화하는 연산 트리와 변환 타입을 선택하기 위한 의도로 만들어졌다. 변환 타입 i에 해당하는 개의 후보 트리들 중에서 우리는 가장 큰 C값을 갖는 후보 트리를 선택한다. 다시 이러한 선택된 6개의 후보 중에서, 가장 큰 C 값을 갖는 하나의 부분 연산트리를 선택하고, 선택된 트리를 그에 해당하는 변환 기법에 의해 변환한다. 이러한 과정은 반복적으로 수행되며, 지연시간을 더 이상 감소시킬 수 없을 때 종료된다.

예로서 우리의 변환 과정을 보이기 위해, 그림 5(a)의 회로를 보자. 그림 왼쪽은 (합성 가능한) VHDL 코드를 보여주며, 오른쪽의 그림은 이것의 RT-단계의 회로 그래프이다. 이 회로는 (모듈 단위의) 한 부분 회로를 포함하고 있다. 는 회로의 입력이며 f는 회로의 출력이다. 가장 긴 수행 경로 위의 연산들로부터 우리의 알고리즘은 각 변환 타입들에 대해 C 값을 최소로 갖는 연산트리들을 각각 선택한다. 그림 5(a)의 회로 그래프 안의 점선으로 이루어진 각 원들은 변환 가능한 연산 트리들을 나타낸다. 이중에 우리는 C값이 가장 큰 (tree5, type6)를 선택하였다. 따라서, tree5는 type6을 적용하여 그림 5(b)에서 보이는 바와 같이 변환된다. 연산트리 tree5의 상수 피연산자 '01101011'는 -a를 ,로 변환할 때 생긴 상수 1과 더해지고, 이 연산자는 부분 회로 안에서 초기에 할당되는 CSA의 입력으로 연결된다. 그리고, 이때 입력의 도달시간이 늦은 피연산자는 이후에 할당되는 CSA의 입력으로 연결된다. 이제, 그림 5(b)내의 변환된 회로의 가장 긴 수행경로는 회로의 오른편을 지나는 것이 된다. 결과적으로, 세 쌍의 (tree, type)가 그림에서 보여지는 바와 같이 선택되고, 이중 tree8과 type3이 선택된 후, tree8은 type3에 의해 변형된다. 그림 5(c)는 그 결과 생성된 회로이다. 마지막으로 tree9와 type2의 쌍이 선택되고 그림 5(d)에서 보이는 회로로 변환된다. 제안한 전체적인 알고리즘의 흐름은 다음과 같이 요약된다.

CSA 변형의 통합적 알고리즘:
/* T: 현 회로의 지연시간 */
* 논리최적화를 적용한 현 회로의 지연시간/면적 계산;
while (회로지연시간 > clock_period) {
* (tree, type)의 모든 가능한 쌍을 알아낸다;
* 각 tree에 type을 시도하여 C 값이 가장 최소인 (tree, type)를 선택하여 회로를 변형한다;
* 회로의 지연시간/면적을 갱신한다;
}

본 알고리즘은 기본적으로 greedy 한 성격을 지니고 있다.그 이유는 다음의 두 가지 사실에서 정당성을 찾을 수 있다: (1) 어떤 조건이 주어져 있을때, 예를 들어 회로 면적의 한계가 주어져 있다고 할 때 이용자는 알고리즘의 수행 중에 반복적으로 조건을 체크하여 알고리즘 수행 과정에 생성되는 결과를 선택할 수도 있을 것이다. 이를 위해서는 생성되는 결과가 이전의 반복에서 생성된 어떠한 결과보다도 좋은 결과를 가져야 한다. 본 알고리즘은 이러한 기능을 해결하는데 용이할 뿐 아니라 while-루프를 반복하는 중에 cost 함수를 동적으로 바꾸면서 알고리즘을 수행할 수도 있게 한다; (2) 우리는, 표 5에서 보이는 실제적으로 쓰이는 큰 회로에 대해 실험한 결과에서 이러한 greedy 최적화는 상당히 잘 동작함을 볼 수 있다. 이는 반복적인 수행에서 변환되는 영역이 전체 회로의 가장 긴 수행경로의 길이에 30~60%에 해당하기 때문이며, 즉, 어느 정도의 "지역적이지 않은" 최적화를 수행하기 때문이다.

우리는, 알고리즘 수행시의 모듈 선택과 논리 최적화 수행을 빠르게 하기 위해, while-루프 안의 두 번째 문장 수행에서는 부분 측정방법을 시도하였다. CSA 수행은 규칙적인 구조를 가지기 때문에, 변형된 트리에서의 지연시간과 회로 면적의 변화를 빠르게 계산하는 것이 비교적 쉽다. 하지만, 각 반복 끝에서 이루어지는 전체 회로 지연시간/면적 갱신에서는 전체적인 모듈 선택과 논리 최적화를 시도하였다. 실험에서 우리는 알고리즘의 각 반복에 적용된 부분 측정방법이 상당히 효율적일 뿐만 아니라, 회로의 질적인 면에서도 나빠지지 않음을 발견하였다.

4. 실험 결과

우리는 흔히 사용되는 여러 가지의 연산식에 알고리즘을 적용해 보았다. 자원선택, 그리고 논리 최적화를 위해 Synopsys의 Design Compiler[15]를 이용하였고 우리의 알고리즘과 [14]에 의한, 그리고 CSA를 이용하지 않은 회로들과 비교하였다.

* 멀티플렉서를 포함한 회로에 대한 실험: 그림 6에서 보이는 mux_1, mux_2, mux_3 회로에 우리의 알고리즘을 적용해 보았다. 그 결과는 표 1에 요약되어 있다. 우리는 곱셈연산의 입력으로 8-비트의 피연산자를 사용하였으며 다른 연산들의 입력으로는 16-비트의 피연산자를 사용하였다. 표의 두 번째, 세 번째, 네 번째의 열은 각각 CSA를 사용하지 않고 최적화된 회로, [14]의 적용 후 생성된 회로, 우리의 알고리즘에 의해 생성된 회로를 나타낸다. mux_1와 mux_3는 첫 번째는 회로면적 제약 하에 지연시간을 최적화하였으며, 두 번째는 지연시간 제약 하에 회로면적을 최적화하였다. mux_2 역시 두 번 실험되었는데 한번은 곱셈을 부분적인 곱셈기와 일반적인 덧셈기로 분해한 방법을 이용하였으며 다른 실험에서는 곱셈을 완전히 덧셈으로 분해하는 방법을 이용하였다. 두 경우 모두 지연시간을 최소화하였다.

* 다중설계를 포함한 회로에 대한 실험: 제시된 알고리즘을 그림 6의 hier_1, hier_2, hier_3 회로에 적용시켜 보았으며, 그 결과는 표 2에 정리되어 있다. 결과의 비교는 다중설계를 포함한 회로에 대한 최적화가 매우 효과적임을 보이고 있다.

* 곱셈을 포함한 회로에 대한 실험: 형식을 갖는 연산식에 우리의 알고리즘을 적용해 보았으며 그 결과는 표 3에 정리되어 있다. 이러한 비교는, 우리의 CSA 적용방법이 회로의 지연시간을 좀더 감소시킬 수 있음을 보여준다. 그러나, 이것은 본문에 설명된 바와 같이, 연산 트리의 마지막에 위치하지 않은 곱셈에 대해서도 CSA최적화를 적용하기 위해 배분법칙을 사용하였으며, 이는 곱셈의 개수를 증가시켰기 때문에 상당한 회로면적 증가 부담을 가지고 왔다. 회로 면적의 증가는 x가 간단한 상수일 때, 즉 일 경우 최소화 된다. 결과적으로, 이와 같은 변환 방법은 회로의 지연시간을 감소시키기 위한 최후의 방법으로 이용됨이 바람직하다.

* 멀티플렉서, 부분 회로, 그리고 곱셈을 포함한 회로의 변환: 그림 6에 보이는 mix_1, mix_2, mix_3과 그림 5(a)에 보이는 ex1 회로에 본문에서 제시된 알고리즘을 적용해 보았다. mix_1은 부분 회로와 멀티플렉서를 포함한 회로이고, mix_2는 멀티플렉서와 곱셈 연산을 포함한, mix_3은 부분 회로와 곱셈 연산을 포함한 회로이다. 그리고, ex1은 다중설계, 멀티플렉서, 곱셈 연산을 모두 포함하고 있다. 표 4에서 보이는 결과는 여러 변환 기법들을 동시에 수행한 CSA 변환 방법이 기존의 CSA 변환의 한계를 극복하는 매우 효과적인 해결책임을 보이고 있다.

* 실제 큰 회로에 대한 실험: 우리는 상위 단계의 몇 개의 benchmark[16] 회로들에 우리의 알고리즘을 적용하였다. MAHA[17]는 6개의 조건 블록으로 이루어져 있다. 우리는 이 회로를 하나의 부분 설계는 가장 바깥의 조건 블록에서의 왼쪽 분기의 세 개의 조건 블록을 포함하며, 다른 하나의 부분 설계는 나머지 회로를 포함하는 두개의 설계로 나누었다. Differential equation.2와 2nd-order iir filter.2는 differential equation과 2nd-order iir filter를 두 번 unrolling 하여 생성된 회로이다. 우리는 MAHA를 제외한 표 5에 보이는 각 회로를 회로 실행 지연시간의 대략의 값을 이용하여 데이타 플로우 그래프의 맨 위부터 아래까지를 4개의 부분 회로로 나누었다. (Synopsys의 Behavioral Compiler[18]를 사용하여 얻어진 지연시간 정보에 따라 회로를 나누었다.) 표 5의 두 번째 열은 각 회로의 연산의 전체 개수를 나타낸다. MAHA의 결과의 비교는 우리의 알고리즘이 회로 면적을 증가시키며 빠른 지연시간을 갖는 CSA 회로를 생성함을 보여주고 있다. 이때 회로 면적의 증가는 멀티플렉서를 포함하는 연산의 중복 생성으로 인해 발생한다. 전체적으로, MAHA 알고리즘으로 인한 지연시간의 향상은 매우 크며, [14]의 알고리즘으로 만들어진 회로와 비교하여 평균 18%, 그리고 CSA를 사용하지 않은 회로와 비교하여 30%의 지연시간 향상이 있었음을 실험에서 볼 수 있다. 이때, 회로면적의 증가는 5% 미만이다.

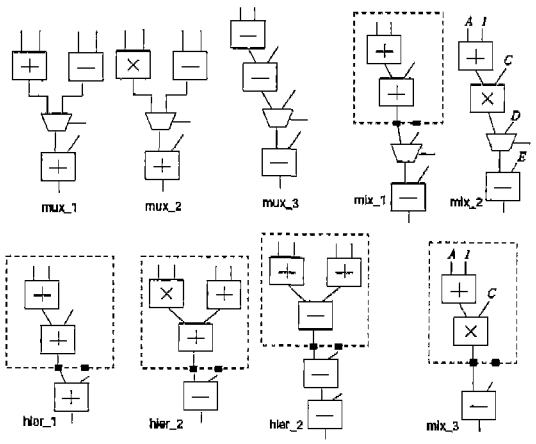


그림 6 멀티플렉서(mux_1, mux_2, mux_3), 다중설계 회로(hier_1, hier_2, hier_3), 그리고 형식의 곱셈의 회로와 멀티플렉서, 다중설계, 곱셈을 포함한 회로

표 1 멀티플렉서를 포함한 회로에 대한 결과 비교

Design	RTL Design (timing, area)	[14] (timing, area)	Ours (timing, area)	improvement over RTL design	improvememt over [14]
mux_1 (timing opt.)	3.26 ns 4742 units	3.26 ns 4742 units	2.09 ns 3565 units	36% faster 25% smaller	36% faster 25% smaller
mux_1 (area opt.)	3.26 ns 4742 units	3.26 ns 4742 units	2.94 ns 2312 units	10% faster 51% smaller	10% faster 51% smaller
mux_2 (partial)	4.78 ns 5036 units	4.78 ns 5036 units	4.20 ns 4265 units	12% faster 15% smaller	12% faster 15% smaller
mux_2 (full)	4.78 ns 5036 units	4.02 ns 5325 units	3.26 ns 5439 units	32% faster 8% larger	19% faster 2% larger
mux_3 (timing opt.)	2.89 ns 5756 units	2.81 ns 5004 units	2.55 ns 4115 units	12% faster 29% smaller	9% faster 18% smaller
mux_3 (area opt.)	2.89 ns 5756 units	2.95 ns 4439 units	3.01 ns 2541 units	4% slower 56% smaller	2% slower 43% smaller

표 2 다중설계를 포함한 회로에 대한 결과 비교

Design	RTL Design (timing, area)	[14] (timing, area)	Ours (timing, area)	improvement over RTL design	improvement over [14]
hier_1 (timing opt.)	3.03 ns 2919 units	3.06 ns 2261 units	2.57 ns 2017 units	25% faster 31% smaller	12% faster 11% smaller
hier_1 (area opt.)	3.03 ns 2919 units	3.06 ns 2472 units	2.88 ns 1365 units	14% faster 53% smaller	6% faster 40% smaller
hier_2 (timing opt.)	4.42 ns 4822 units	3.80 ns 4750 units	3.29 ns 4591 units	26% faster 5% smaller	13% faster 3% smaller
hier_2 (area opt.)	4.42 ns 4822 units	4.30 ns 3406 units	3.99 ns 3416 units	10% faster 29% larger	7% faster same
hier_3 (timing opt.)	4.18 ns 3993 units	3.18 ns 3907 units	2.45 ns 3632 units	41% faster 9% smaller	23% faster 7% smaller
hier_3 (area opt.)	4.18 ns 3993 units	3.58 ns 3255 units	3.60 ns 2390 units	14% slower 40% smaller	same 27% smaller

표 3 형식의 곱셈 연산식을 포함한 회로에 대한 결과 비교

Dcsign	RTL Design (timing, area)	[14] (timing, area)	Ours (timing, area)	improvement over RTL design	improvement over [14]
(a+b)*c-d (timing opt.)	5.14 ns 5066 units	4.08 ns 3813 units	3.75 ns 6052 units	27% faster 19% larger	8% faster 59% larger
(a+39)*c-d (timing opt.)	4.80 ns 4290 units	3.93 ns 3277 units	3.66 ns 4360 units	24% faster 2% larger	7% faster 32% larger
(a+7)*c-d (timing opt.)	4.66 ns 4308 units	3.87 ns 3369 units	3.58 ns 4033 units	23% faster 6% smaller	7% faster 20% larger
(a+1)*c-d (area opt.)	4.12 ns 4892 units	3.43 ns 3430 units	3.35 ns 3217 units	19% faster 34% smaller	2% faster 6% smaller

표 4 멀티플렉서, 다중설계, 를 모두 포함한 회로에 대한 결과 비교

Design	RTL Design (timing, area)	[14] (timing, area)	Ours (timing, area)	improvement over RTL design	improvement over [14]
mix_1 (timing opt.)	3.41 ns 4456 units	3.20 ns 4399 units	2.49 ns 4051 units	27% faster 11% smaller	22% faster 8% smaller
mix_2 (timing opt.)	4.49 ns 5296 units	4.27 ns 4676 units	3.34 ns 5129 units	26% faster 3% smaller	22% faster 10% larger
mix_3 (timing opt.)	4.66 ns 4329 units	4.36 ns 4147 units	3.33 ns 3607 units	29% faster 17% smaller	24% faster 13% smaller
test_1 (timing opt.)	5.68 ns 9911 units	5.48 ns 8124 units	4.83 ns 8356 units	15% faster 16% smaller	12% faster 3% larger
test_1 (area opt.)	5.68 ns 9911 units	5.88 ns 6560 units	5.49 ns 6331 units	3% faster 36% smaller	7% faster 3% smaller

표 5 큰 회로에 대한 결과 비교

Design	#ops	RTL Design (timing, area)	[14] (timing, area)	Ours (timing, area)	improvement over RTL design	improvement over [14]
MAHA	16	5.83 ns 9986 units	5.92 ns 10141 units	4.31 ns 10835 units	26% faster 9% larger	27% faster 7% larger
Differential equation.2	20	12.92 ns 22171 units	9.89 ns 20235 units	9.51 ns 0586 units	26% faster 7% smaller	4% faster 2% larger
5th-order elliptic filter	33	10.66 ns 16187 units	8.12 ns 14096 units	7.20 ns 13396 units	32% faster 17% smaller	11% faster 5% smaller
2nd-order iir filter.2	18	9.94 ns 29424 units	9.76 ns 26468 units	8.31 ns 28738 units	16% faster 2% smaller	15% faster 9% larger
average					30% faster 9% smaller	18% faster 5% larger

5. 결 론

이 논문은 캐리-세이브 가산기 (CSA)를 이용하여 연산 회로를 최적화하는 새로운 알고리즘을 제시하였다. 구체적으로, 우리는 멀티플렉서를 포함한 연산회로 최적화, 다중설계를 지닌 연산회로 최적화, 곱셈를 포함한 연산회로 최적화 기법을 소개하였으며, 이 기법들을 효과적으로 통합 회로 전체에 걸쳐 광범위한 CSA 사용을 이루는 알고리즘을 제시하였다.

우리의 알고리즘이 기존의 CSA를 사용하지 않은 방법, 또한 CSA를 사용한 기존의 방법으로 만들어진 회로보다 지연시간을 얼마나 향상시켰는가를 실험에서 보였다. 그러나, 이러한 향상은 경험이 풍부한 설계자의 수작업에 의해 만들어진 CSA 회로보다 항상 좋은 성능을 가지고 오는 것을 의미하지는 않는다.하지만, 회로 설계자들이 CSA 변환을 직접 수행 하는데에 있어 많은 수작업은 지루할 뿐 아니라 에러를 생성하기 쉽지만, 우리의 방법은 CSA 변환이 자동적으로 수행되고, 표준적인 HDL의 합성 환경의 안에서 수행 가능하다.

참 고 문 헌

[1] D. Gajski, N. Dutt, A. Wu, and S. Lin, High-Level Synthesis -Introduction to Chip and System Design}, Kulwer Academic Publishers, 1992.

[2] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincente, Logic Minimization Algorithms for VLSI Synthesis, Kulwer Academic Publishers, 1984.

[3] R. K. Brayton, R. Rudell, A. L. Sangiovanni-Vincentelli, and A. R. Wang, "MIS: A Multiple-level Logic Optimization Systems," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. CAD-6, pp. 1062-1082, 1987.

[4] J. Grodstein, E. Lehman, H. Harkness, B. Grundmann, and Y. Watanabe, "A Delay Model for Logic Synthesis of Continuously-Sized Networks," Proc. of IEEE International Conference on

Computer-Aided Design, pp. 458-462, 1995.
[5] S. S. Sapatnekar and W. Chuang, "Power vs. Delay in Gate Sizing: Conflicting Objectives?," Proc. of IEEE International Conference on Computer-Aided Design, pp. 463-466, 1995.
[6] M. Potkonjak and J. Rabaey, "Maximally Fast and Arbitrarily Fast Implementation of Linear Computations," Proc. of International Conference on Computer-Aided Design), pp. 304-308, 1992.
[7] D. Lobo and B. Pangrle, "Redundant Operation Creation: A Scheduling Optimization Technique" Proc. of Design Automation Conference, pp. 775-778, 1991.
[8] A. Nicolau and R. Potasma, "Incremental Tree Height Reduction for High-level Synthesis," Proc. of Design Automation Conference, pp. 770-774, 1991.
[9] R. Hartley and A. E. Casavant, "Tree-Height Minimization in Pipelined Architectures," Proc. of International Conference on Computer-Aided Design, pp. 112-115, 1989.
[10] R. Hartley and A. E. Casavant, "Optimized Pipelined Networks of Associative and Commutative Operators," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 13, No. 11, November 1994.
[11] K. K. Parhi, "High-Level Algorithm and Architecture Transformations for DSP Synthesis," Journal of VLSI Signal Processing, No. 9, pp. 121-143, 1995.
[12] M. Janssens, F. Catthoor, H. De Man, "A Specification Invariant Technique for Regularity Improvement between Flow-Graph Clusters," Proc. of European Design Automation Conference, pp. 138-143, 1996.
[13] N. Weste and K. Eshraghian, Principles of CMOS VLSI Design-A Systems Perspective, Addition-Wesley Publishers, 1985.
[14] T. Kim, W. Jao, and S. Tjiang, "Circuit Optimization using Carry-Save-Adder Cells," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 17, No. 10, pp. 974-984, Oct. 1998.
[15] Synopsys Inc., Design Compiler Manual, 1998.
[16] P. Paulin and J. Knight, "High-level Synthesis Benchmark Results using a Global Scheduling Algorithm," Logic and Architecture Synthesis for Silicon Compilers, North-Holland, pp. 211-228, 1898.
[17] A. C. Parker, J. Pizarro, and M. J. Mlinar, "MAHA: A Program for Datapath Synthesis," Proc. of IEEE Design Automation Conference, pp. 461-466, 1986.
[18] Synopsys Inc., Behavioral Compiler Manual, 1998.

김 태 환
정보과학회논문지 : 시스템 및 이론
제 28 권 제 1 호 참조

엄 준 형
서울대학교 수학교육과 학사. 한국과학기술원 수학과 석사. 한국과학기술원 전산학과 박사과정. 관심분야는 Combinatorial Optimization, Behavioral and Logic Synthesis