캐리-세이브 가산기를 이용한 지연시간 최적화를 위한 연산기 합성

김태환 엄준형 김영태 여준기 홍성백
한국과학기술원 전산학과
첨단정보기술 연구센터(AITrc)

A Timing-Driven Synthesis of Arithmetic Circuits using Carry-Save-Adders

{tkim, jhum, young, orualy, hongsup}@vlsisyn.kaist.ac.kr
Taewhan Kim Junhyung Um Youngtae Kim Chungi Lyuh Sungpack Hong

Dept. of Computer Science and Advanced Information Technology Research Center(AITrc)
Korea Advanced Institute of Science and Technology

요약

캐리-세이브 가산기(CSA)는 연산식의 빠른 수행을 위해 가장 일반적으로 쓰이는 연산기중에 하나이다. 일반적인 CSA 적용의 근본적인 한계로는, 연산 회로중에 바로 덧셈 연산으로 변환되는 부분만이 적용이 가능하다는 사실이다. 이러한 제한점을 극복하기 위하여, 우리는 간단하고도, 효율적인 CSA 변환 방법을 제시한다. 이들은 (1)*멀티플렉서를 포함한 최적화*, (2)*회로 경계를 포함한 최적화*, (3)*곱셈기를 포함한 최적화*이다. 이러한 방법을 포함하여, 우리는 전체적인 회로에서 CSA를 충분히 사용할 수 있는 새로운 지연시간 최적화를 목표로 하는 CSA 변환 방법을 만들어 내었다. 실험에서는 실제적인 여러 회로에 대해 제시된 방법이 효율적임을 보였다.

1 서론

회로의 지연시간은 합성의 여러 단계에서 최적화 되어야 할 중요한 요소중에 하나이다. 본 논문에서 제시된 방법은, 회로와 사이클 시간이 주어졌을때 연산기 단계에서 지연시간을 최적화 한다는 점에서 RTL합성에 해당된다. 그러나, 우리는 지연시간 최적화를 위해 간단한 연산기 조작이나 상수 지연등의 기존의 자주 이용되는 방법과는 다른, 새로운 빠른 연산기 캐리-세이브 가산기(CSA) 를 이용한[1], 트리 변환등의 방법과는 완전히 다른 새로운 방법을 제시한다.

n비트 CSA는 n개의 서로 독립적인 full adder(FA)로 구성되어 있다. 이는 세개의 n 비트 입력을 가지며, 두개의 출력을 생성한다. 그 중 하나는 n 비트의 sum 벡터이며, 또다른 하나는 n 비트의 carryout 벡터이다. 일반적인 가산기와는 다르게, CSA는 내부적인 carry 지연을 갖고 있지 않다. 우리는 *CSA tree*를, CSA 연산기들로 이루어져 있고, 최종적으로 보통의 일반적인 가산기 한개로 이루어진 트리로 정의한다. CSA 트리는 임의의 주어진 덧셈 연산을 변환하여 두개의 피연산자를 생성하며 최종적으로 이는 보통의 일반적인 가산기를 이용하여 더해진다. 이때, CSA 변환은 가산에만 제한되지 않는다[2, 4].

우리의 알고리즘은 일반적으로 보여지는 멀티플렉서를 포함한 연산 트리, 회로 경계를 포함한 연산트리, 그리고 곱셈을 포함한 연산트리에 대해 적용이 불가능했던 기존의 일반적인 CSA 변환[2]의 단점을 극복한다.

2 CSA 최적화

2.1 멀티플렉서를 포함한 최적화 (OA_{mux})

회로 설계 내의 조건문이나, 자원 공유는 회로내에 멀티플렉서를 생성한다. 그림 1(a)와 (b)는 각각 조건문에 대한 VHDL 회로 설계와 그에 대응하는 변환된 회로 그래프를 각각 나타낸다. 그림 1(b)의 점선은 회로의 주요 경로(critical path)를 나타낸다. 우리는 이 경우 세 단계의 변환을 수행한다:

1. *Move-up:* 멀티플렉서에서 생성된 입력으로부터의 주요 경로 위의 연산기를 그림 1(c)에서 보이는 바와 같이 멀티플렉서 위로 옮겨여, 연산 트리를 생성한다. 이 경우 회로 면적은 증가하지만 회로 지연시간은 변하지 않는다.

2. *Transformation:* 주요 경로 위의 조건 분기 위에 정의된 연산 트리는 그림 1(d)에서 보이는 CSA트리로 변환된다.(트리 1) 그리고, 오른쪽의 조건 분기가 중복된 연산기를 루트로써 포함한 경우 (트리 2), 이또한 CSA 트리로 변환된다.

3. *Move-down:* 조건 분기 위에 변환된 트리의 마지막 가산기를($op1, op2$) 멀티플렉서를 경유하여 아래로 내리고, 그림 1(e)에서 보이는것처럼 통합한다. 이 과정에서 멀티플렉서는 중복될 수 있지만, 지연시간은 증가하지 않는다.

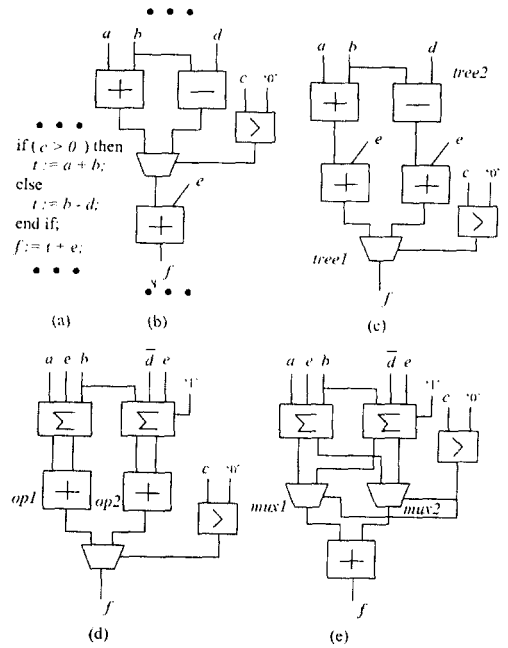

그림 1: 멀티플렉서를 이용한 CSA 변환의 예

2.2 회로 경계를 포함한 최적화 (OA_{bound})

그림 2(a)는 두개의 회로 A,B를 가지고 있는 연산 회로를 보여준다. 기존의 CSA변환을 두개의 회로에 차례로 적용한 결과는 그림 2(b)에서 보여진다. 서로 다른 회로에 속한 두개의 연산기를 합하기 위해, 우리는 [3]에서 제시된 새로운 포트를 할당하는 개념을 사용한다. 그러나, 하나의 부가적인 포트만 사용하더라도 속도가 그다지 감소하지 않고 또한 간단하기 때문에 우리는 하나의 부가적인 포트만을 사용한다. 그림 2(c)는 주어진 포트 r외에 또다른 포트 s를 할당한 예이다. 초기에는 포트 s의 값이 0으로 주어져 있다. 이 경우 회로 A의 연산 트리의 두개의 출력을 두개의 포트를 이용해 다른 회로 B로 연결함으로써, *최종 가산기 없이* CSA 트리를 생성할 수 있다. 그림 2(d)는 그림 2(c)의 연산식으로부터 변형된 최종적인 CSA 트리를 보여준다.

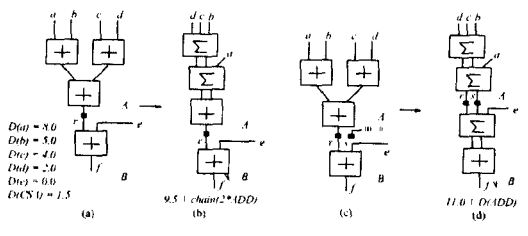

그림 2: 다른 회로에 속하는 연산들에 대한 변환의 예

2.3 곱셈기를 포함한 최적화 (OA_{mult})

CSA들로 변형되기 위해 선택된 연산트리는 곱셈기를 포함할 수 있다. 이 경우, 곱셈기는 트리의 마지막 노드에 위치해야 한다[2]. 곱셈기를 포함한 이러한 CSA 변환의 영역을 확대하기 위해, 곱셈기가 회로의 주요 경로 위에 있을 경우 우리는 연산식에 대해 distribution rule을 적용한다.

3 알고리즘

3.1 변환의 종류

최적화의 효율을 극대화 하기 위해, 단락 2에서의 CSA 변환방법을 같이 혼합하여 적용하는 것이 필요하다. 우리의 알고리즘은 CSA 변환방법을 6가지를 분류한다.

type1: *일반적 최적화*[2] - 변환될 CSA 트리는, 전체적으로 덧셈, 뺀셈, 곱셈기 (곱셈기가 트리의 가장 마지막 노드에 위치할때) 로 구성되어야 한다.

type: OA_{mux}; **type3**: OA_{bound}; **type4** : OA_{mult}

type5: OA_{mult}+OA_{mux}- OA_{mult}가 수행된 후, OA_{mux}가 다시 연산 트리에 수행된다.

type6: OA_{mult}+OA_{bound}- OA_{mult}가 수행된 후, OA_{bound}가 연산 트리에 다시 적용된다.

3.2 변환 방법

회로의 주요 경로위에 주어진 연산기들에 대해, 우리는 각 변환 방법 i,(i=1,2,$\cdots$,6)이 적용될 모든 연산 트리를 추출한다. 각 변환 방법 i에 대해, 모두 m_i개의 후보가 있다고 가정하자. 우리는 m_i개의 후보중에 가장 좋은 트리를 선택하기 위한 cost 함수를 정의한다. 각 m_i개의 후보들에 대해, 우리는 그에 대응하는 변환을 적용하고 다음의 비율을 계산한다.

$$\Delta T/(A - \Delta A) \qquad (1)$$

이때, ΔT와 ΔA는 각각 결과 회로의 지연시간과 회로 면적 감소의 양을 나타낸다. 그리고, A는 변환 전의 회로의 면적을 나타낸다.

이러한 비율은 회로 면적의 증가에 대한 주요 경로 위의 지연시간 감소의 효율성을 측정할 수 있다. m_i개의 후보중에, 우리는 가장 큰 값의 비율을 갖는 후보자를 선택한다. 다시 이러한 선택된 6개의 후보 중에서, 우리는 가장 큰 비율을 갖는 하나를 선택하고, 선택된 트리를 그에 대응하는 변환 방법에 의해 변환한다. 이러한 과정은 지연시간을 더 이상 감소시킬 수 없을때까지 반복된다.

```
알고리즘
/* A, T: 현 회로의 면적, 지연시간 */
/* ΔA, ΔT: 변환으로 인한 면적과 지연시간의 증가 */
• 현 회로의 지연시간/면적 계산 (모듈 수행, logic 최적화 이용);
while (T > cycle_time) {
  • (op_tree, trans_type)의 모든 쌍을 골라낸다;
  • 각 op_tree에 trans_type을 적용하고 ΔA, ΔT를 계산한다
  (연산기 선택만 이용); (statement_a)
  • 가장 큰 ΔT/(A-ΔA)값을 갖는것을 골라내고 변환한다;
  • 회로의 지연시간/면적을 갱신한다
  (모듈 수행, logic 최적화 이용); (statement_b)
}
```

그림 3: 알고리즘

그림 3는 전체적인 알고리즘을 요약하였다. 모듈 선택과 logic최적화 수행을 빠르게 하기 위해, 우리는 부분측정

functions	bit-width	impl. selection + logic opt.		
		area-eff. time, area	moderate time, area	time-eff. time, area
$A+B$	16×16	3.47, 339	2.38, 686	1.37, 1172
$A-B$	16×16	3.55, 387	2.42, 735	1.47, 1387
$A+B$	8×8	1.71, 163	1.59, 274	0.94, 577
$A-B$	8×8	1.79, 187	1.60, 284	0.90, 697
$A \times B$	8×8	5.92, 1610	4.49, 2234	3.59, 2820
$A \times_p B$	8×8	-	2.22, 1393	-
$Mux(A,B)$	$n \times n$	-	$0.32, 24 \cdot n$	-
$Inverter(A)$	n	-	$0.06, 3 \cdot n$	-
$CSA(A,B,C)$	$n \times n \times n$	-	$0.36, 21 \cdot n$	-

표 1: 모듈에 대해 지역적인 변환에 의한 회로 면적과 지연시간의 변환을 측정하기 위한 라이브러리

방법 (statement_a)를 이용한다. 즉, 우리는 CSA트리의 모듈의 현재 반복만을 고려한다. CSA 수행은 규칙적인 구조를 가지기 때문에, 변형된 트리에서의 지연시간과 회로 면적의 변화를 빠르게 계산하는것이 비교적 쉽다. 그리고, 우리는 각 반복 후의 전체적인 모듈 선택이나 logic 최적화로 인한 부분적인 회로 지연시간/면적을 계산하는 과정을 좀 더 간단한 방법으로 대체하고자 한다. (statement_b).

*op_tree*에 대한 *trans_type*을 적용했을때의 cost의 값을 계산하는것은 CSA 트리의 부분적인 변환에 기인한 회로 속도와 면적의 변화를 측정하는것이다. 우리의 알고리즘은 각 모듈 종류에 대해 회로 면적을 최적화 할 경우, 지연시간을 최적화 할 경우, 그리고 회로 면적/지연시간을 최적화하는 각 경우의 모듈 선택과 logic 최적화 된 표 1과 같은 라이브러리를 이용한다.

4 실험 결과

우리는 흔히 사용되는 여러가지의 연산식에 알고리즘을 적용해 보았다. 모듈 선택, 그리고 logic 최적화를 위해 Synopsys Inc.의 Design Compiler package를 이용하였고 우리의 알고리즘과 [2]에 의한, 그리고 CSA를 이용하지 않은 회로를 비교하였다. 우리는 곱셈 입력으로는 8비트, 그리고 나머지 입력으로는 16비트를 사용하였으며 모든 입력의 도달시간을 0으로 가정하였다. 표 2의 결과는 우리의 OA_{bound}, OA_{mux} 그리고 OA_{mult}를 같이 사용하는 방법이 기존의 CSA 변환 방법의 제한점을 극복하며 좀더 빠르고 적은 면적을 차지하는 회로를 생산하기 위해 전반적인 회로에 CSA 변환 방법을 확장할 수 있음을 보인다.

5 결론

이 논문은 CSA를 이용하여 연산 회로를 최적화 하는 새로운 알고리즘을 제시하였다. 우리는 (1)*멀티플렉서를 포함한 최적화*, (2)*회로 경계를 포함한 최적화*, (3)*곱셈기를 포함한 최적화*를 이용하여 기존의 CSA 변환 방법의 제한을 극복하였다. 이러한 방법은 전반적인 회로에 대한 CSA의 광범위한 사용을 허락한다. 우리는, 이러한 세가지 방법을 충분히 사용하기 위해 새로운 알고리즘을 제시하였다. 또한, 알고리즘의 효율성을 높이기 위해 빠르고, 또한 비교적 정확한 회로 지연시간/면적 측정 방법을 제시하였다.

감사의 글

Designs	RTL time/ area	[2] time/ area	Ours time/ area	Impr. over RTL	Impr. over [2]
mux_1	3.26	3.26	2.09	36%	36%
	4742	4742	3565	25%	25%
mux_2	4.78	4.78	4.20	12%	12%
	5036	5036	4265	15%	15%
mux_3	2.89	2.81	2.55	12%	9%
	5756	5004	4115	29%	18%
$(a+b) \cdot c - d$	5.14	4.08	3.75	27%	8%
	5066	3813	6052	-19%	-59%
hline $(a+39) \cdot c - d$	4.80	3.93	3.66	24%	7%
	4290	3277	4360	-2%	-32%
$(a+7) \cdot c - d$	4.66	3.87	3.58	23%	7%
	4308	3369	4033	6%	-20%
$(a+1) \cdot c - d$	4.12	3.43	3.35	19%	2%
	4892	3430	3217	34%	6%
hline mix_1	3.41	3.20	2.49	27%	22%
	4456	4399	4051	11%	8%
mix_2	4.49	4.27	3.34	26%	22%
	5296	4676	5129	3%	-10%
mix_3	4.66	4.36	3.33	29%	24%
	4329	4147	3607	17%	13%

표 2: 여러 연산 회로들에 대한 결과 비교

본 논문은 첨단정보기술 연구센터(AITrc)를 통하여 과학재단의 지원을 받았음.

참조 서적

[1] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design - A Systems Perspective*, Addition-Wesley Publishers, 1985.
[2] T. Kim, W. Jao, and S. Tjiang, "Circuit Optimization using Carry-Save-Adder Cells", *IEEE TCAD*, October 1998.
[3] J. Um, T. Kim, C. L. Liu, "Optimal Allocation of Carry-Save-Adders in Arithmetic Optimization", *Proc. ICCAD*, 1999.
[4] Synopsys Inc., *DesignWare Components Databook*, 1996.
[5] LSI Logic Inc., *G10-p Cell-Based ASIC Products Databook*, 1996.

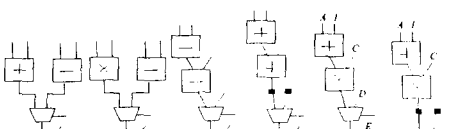

그림 4: (a) 멀티플렉서를 포함하는 회로; (b) 멀티플렉서, 여러개의 회로, $(a+X) \cdot c$ 형식의 곱셈을 포함하는 회로의 예