연산 회로에서의 모듈 배치를 통한 지연시간 최적화 알고리즘

김동현
과학영재학교
김태환[0]
서울대학교, 전기.컴퓨터공학부
{dhkim, tkim}@ssl.snu.ac.kr

Algorithm for Timing Optimization Using Module Placement in Arithmetic Circuits

Donghyun Kim
Busan Science Academy
Taewhan Kim[0]
School of EECS, Seoul National University

요 약

본 연구는 컴퓨터 연산을 위한 하드웨어 설계에서 고성능 연산에 사용되는 캐리-세이브 가산기 (Carry-save adder) 합성에 관한 연구이다. 기존의 연구에서는, 연산 합성 문제와 합성된 연산의 배치 문제를 두개의 연속된 독립된 두개의 문제로 간주하고 풀었지만, 본 연구에서는 연산 합성 과정에서 연산 배치를 고려한 통합된 방법을 제시하여 전체적인 최적화된 결과를 얻었다. 연결선 상에서의 전력 소모나 지연시간이 점점 더 중요해지는 시스템-온-칩 (system-on-chip) 설계에서 본 연구의 통합적인 설계 방법은 매우 긴요하며 앞으로 효과적으로 이용될 수 있을 것이다.

1. 서 론

하드웨어설계에서의 연산 최적화는 점점 더 소형화 되는 하드웨어와 더불어 그 중요성이 점점 중요해 지고 있다. 특히 그것에 대해 시스템 온 칩(System on Chip) 형태의 이런 초소형 디자인에서는 이런 회로의 설계가 매우 중요하다.

이런 회로의 설계에서는 더하기, 빼기 등의 다양한 오퍼레이션이 있다. 이들 중에서 CSA(Carry Save - Adder)는 특히 시간, 공간적 측면에서 상당한 이점을 가지고 있다. CSA는 3개의 n-bit input-bit을 가지고 있으며 2개의 n-bit output-bit을 가지고 있다. Output-bit 는 n-bit Sum Vector 와 n-bit Carry Vector 로 이루어져 있다. CSA의 구조적 특성상 n이 커질수록 CSA는 더 큰 효과를 낼 수 있으며 [1]에 의하면 일반적인 adder cell보다 더 뛰어난 성능을 보여주었다. 연산회로들은 [1]에 의하여 CSA로 변환할 수 있다.

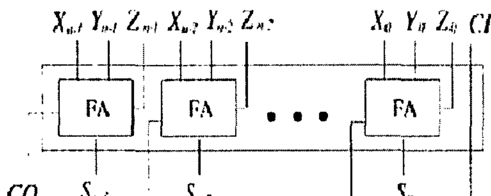

그림 1. CSA 구조

그림 2는 연산 수식으로부터 CSA tree 를 생성한 예를 보여준다. 이 CSA 모형에서 각각의 변수 A, B, C, D, E는 각각 delay를 가지고 있으며 최종적으로 나오는 F의 delay는 A + B + C + D + E = F 를 구했을 때의 시간이다.

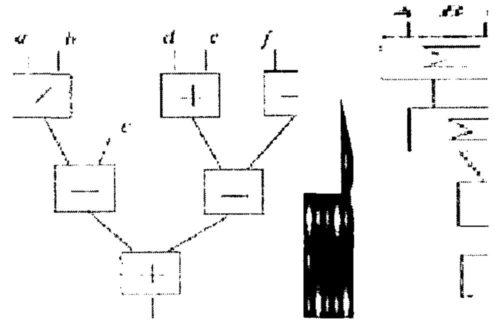
그림 2. CSA tree 생성 예

여기서 만들어지는 CSA tree는 그 회로의 배치에 따라서 매우 효율적인 설계가 될 수도 있고 부적합한 설계가 될 수도 있다. 예를 들어서 그림3 과 같이 약간 변형시킨 CSA tree 배치가 나올 수도 있는데 이런 경우 마지막의 D(F) 의 값이 조금 달라질 수도 있다 (D(x) = x의 delay).

이런 비효율성을 피하기 위해서 CSA tree를 만들 때 가장 빠른 delay를 가진 3개의 변수끼리 묶어서 CSA를 만드는 Greedy Algorithm 을 사용하는데 그렇게 만들면 마구잡이로 만드는 것 보다 상당한 시간이 단축됨을 알 수 있다.

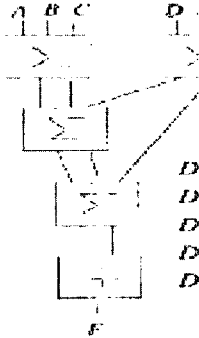

그림 3. 그림 2에서의 생성된 CSA tree 와 구조가 조금 달라진 CSA tree

그러나 [1]의 알고리즘에서 문제점이 발견되는데 CSA 와 CSA 사이의 회로에서 걸리는 지연시간을 고려하지 않았다는 것이다. 그림 4. 에 따르면 1번 CSA에서 3번 CSA 로 갈 때의 CSA 의 delay가 조금 달라져 있음을 알 수 있다. 이렇게 약간씩의 변화에 의해 기존의 알고리즘에서 문제점이 발견될 수 있으며 이런 문제점을 해결하기 위해서는 CSA tree를 만드는 것뿐만이 아닌 배치와 연결선의 지연시간(wire delay)도 고려해주는 것이 필요하다.

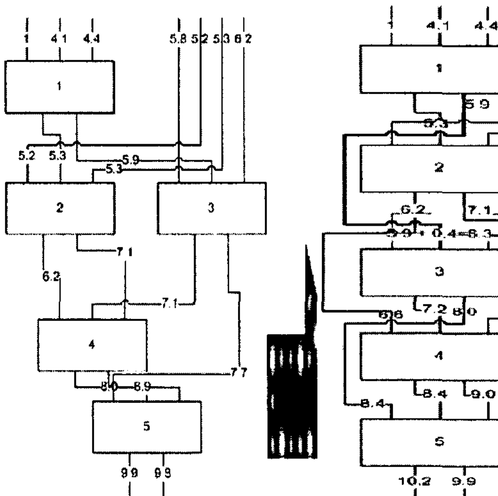

그림 4. 회로에서 걸리는 지연시간을 고려한 것과 고려하지 않은 것의 차이

본 연구는 효율적으로 delay를 줄일 수 있는 배치방법에 대한 알고리즘을 제안한다.

2. 지연시간 최소를 고려한 모듈 배치 알고리즘

앞에서 살펴본 대로 기존의 CSA tree를 만드는 방법에서는 그 구조만 생각했었다. 우리는 CSA의 구조를 만들 때 그 다음에 올 input-bit의 연결선 지연시간(wire delay)까지 고려하여 가장 낮은 delay를 가지는 input-bit를 선택하였다.

연결선의 지연시간은 선의 길이에 비례한다. CSA 와 CSA 사이의 간격에 따라서 지연시간은 달라질 수 있다. 그래서 연결선의 지연시간을 구하기 위해서 input-bit 이 들어가는 위치와 output-bit 이 나가는 위치를 고려하여 그 값을 적당히 임의로 주었다. 바로 위의 CSA 에서 값을 받을 때는 지연시간을 치지 않았으며 간격이 하나씩 떨어질 때마다 α 만큼의 delay를 주었다.

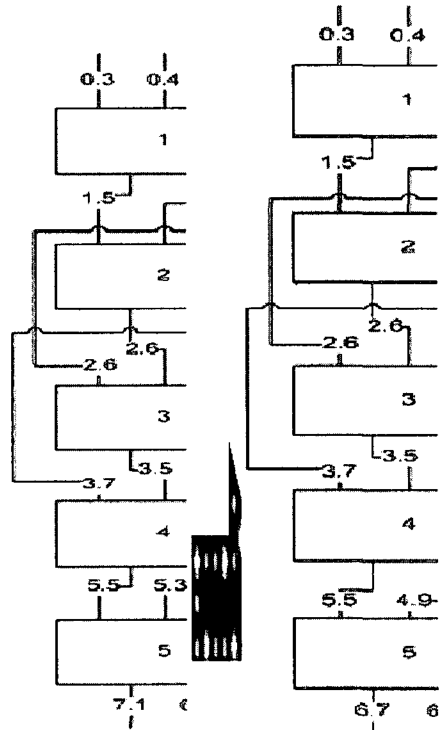

그림5. 1단 (stack) 배치 형태에서의 모듈 배치가 지연시간에 미치는 예

그림 5 은 기존의 알고리즘과 개선한 알고리즘을 비교한 것이다. 이 그림에 따르면 마지막에 나온 delay가 왼쪽의 기존알고리즘은 7.1, 오른쪽의 개선알고리즘은 6.7 로 다른데 잘 살펴보면 3번 CSA 와 4번 CSA 사이에서 약간의 변화가 있음을 알 수 있다. 이것은 기존의 알고리즘에서 최적화된 구조가 나왔다고 하더라도 회선의 delay를 고려하지 않았기 때문에 최적화되지 않는다는 것을 보여준다.

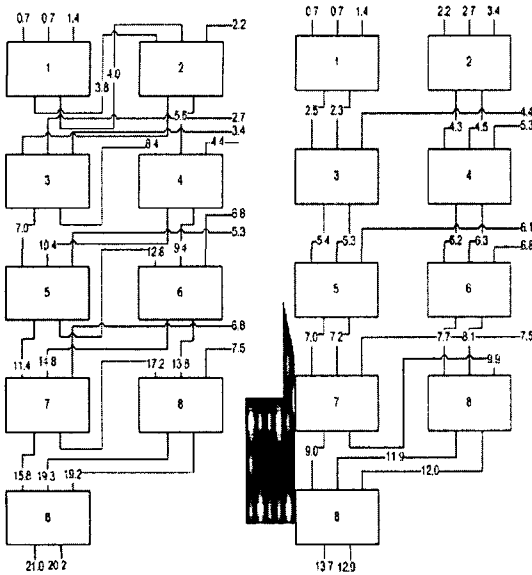

그림 6. 2단 배치 형태에서의 모듈 배치가 지연시간에 미치는 예

그림 6 은 2단 배치를 했을 때를 비교한 것이다. 여기서도 약간의 지연 시간개선이 있음을 알 수 있다.

제안한 모듈 배치와 CSA tree을 동시에 고려한 알고리즘은 기존의 [1]의 제안한 모듈 배치를 고려하지 않고 CSA tree 를 생성한 알고리즘에 모듈 배치를 추가하여 새로운 CSA 에 입력으로 선택될 operand을 모듈 배치에 따른 연결선에서의 지연시간을 추가한 도착시간에 따라 제일 빠른 도착 시간을 가진 operand 를 세 개 선택하여 CSA 입력으로 적용하는 것이다. 자세한 알고리즘은 다음과 같이 기술된다; 먼저 집합 S의 원소 (처음에는 연산 수식의 operand 임.)를 정렬한 뒤에 위치에 따른 wire delay를 추가한다. 추가된 값에 따라 다시 집합 S의 원소들을 정렬시키고 그 값에 따라 가장 빨리 오는 세 개의 operand를 선택하여 CSA를 만든다. 나오는 output operands 를 set S 에 추가시키고 다시 처음의 과정을 반복하면 operands가 2개가 남게 되는데 그 두 개를 묶어서 adder 로 만든다.

3. 실험 결과

우리는 제안한 알고리즘의 분석을 위하여 c++ 프로그래밍을 통한 실험을 하였다. CSA의 delay를 구하기 Synopsys DC[2]와 0.35u [3]을 이용하였다. 표 1, 표 2에서는 기존의 알고리즘과 새로운 알고리즘의 비교를 한 것을 정리하여 결과로 출력한 것이다. 실험결과는 1단, 2단 두 가지로 나뉘어서 진행하였으며 8bit, 16bit, 32bit 로 나누어 진행하였다. 표에 따르면 1단에서는 평균 약 3%의 이점이, 2단에서는 약 13% 의 성능향상을 보였다. 이는 배치의 형태가 다양할수록 연결선의 지연시간이 상대적으로 늘어나며, 본 제안한 알고리즘은 연결선의 지연시간을 줄이는 방향으로 모듈 배치를 하고 있음을 알 수 있다.

표 1. 1단 배치에서의 몇 가지 산술연산에서의 기존 알고리즘과 개선 알고리즘의 비교

설계	지연 시간		
	기존	개선	감소율(%)
EXP1(16bit)	256	249	2.69
EXP2(32bit)	488	480	1.74
EXP3(8bit)	87	83	5.13
EXP4(16bit)	146	140	3.76
EXP5(32bit)	140	134	4.0
EXP6(16bit)	398	390	2.09
평균			**3.13**

표 2. 2단 배치에서의 산술연산에서의 기존 알고리즘과 개선 알고리즘의 비교

설계	지연시간		
	기존	개선	감소율(%)
EXP1(16bit)	151	132	12.5
EXP2(32bit)	269	247	8.1
EXP3(8bit)	63.1	50.6	19.8
EXP4(16bit)	91.9	79.4	13.6
EXP5(32bit)	91.9	75.1	18.2
EXP6(16bit)	248	228	8.0
평균			**13.36**

EXP1(16bit) : $i_1 - i_2 + i_3 \times i_4 + i_5 + i_6 + i_7 \times i_8$

EXP2(32bit) : $i_1 - i_2 + i_3 \times i_4 + i_5 + i_6$

EXP3(8bit) : $i_1 \times i_2 + i_3 - i_4 + i_5 \times i_6 + i_7 + i_8 + i_9 - i_{10}$

EXP4(16bit) : $i_1 \times i_2 + i_3 - i_4 + i_5 - i_6$

EXP5(32bit) : $i_1 - i_2 + i_3 + i_4 + i_5 + i_6 + i_7 + i_8 + i_9 - i_{10}$

EXP6(16bit) : $i_1 \times i_2 + i_3 \times i_4 + i_5 \times i_6 + i_7 \times i_8$

감사의 글: 본 연구는 KAIST 과학영재 교육원의 R&E 프로그램 지원을 받았음.

참고문헌

[1] T. Kim, W. Jao, and S. Tjiang, "Circuit optimization using carry-save-adder cells", *IEEE TCAD*, pp.974-984, October 1998.
[2] Synopsys Inc., *Design Compiler User Guide*, 2002
[3] LSI Logic Inc., *G10-p Cell-Based ASIC Products Databook*, 2000