英語論文輪講

岩淵 勇樹

2006 年 11 月 9 日

Yuke Wang and Parhi, K.K.,"**A unified adder design**," IEEE Conference Record of the Thirty-Fifth Asilomar Conference on, Pacific Grove, CA, USA, pp. 177-182 Vol.1, November 2001

Abstract

- 桁上げ伝播加算器 (RCA)
- 桁上げ先見 (CLA) 加算器
- prefix 加算器
- canonic 加算器
- block-based(多段型) CLA 加算器
- 桁上げ飛び越し加算器 (CSkA)
- conditional-sum 加算器 (CSuA)
- 桁上げ選択加算器 (CSlA)
- 複合型加算器

について、比較・改良のためにその仕組みを一様化して捉える。それらすべての加算器は桁上がり (carry) と和 (sum) の 2 パートに別けられ、その中でも桁上がりは

- prefix 演算器
- fco 演算器
- マルチプレクサ (MUX)
- AND-OR ゲート
- パストランジスタ[1]

といった機構を用いて実現される均一なキャリー演算によって生成される。キャリーの生成に最も効率的な方法は、グループによる生成の代わりに、連続的でなく非等長なものの集まりであるレイヤーによる生成を用いることである。我々は、 CSuA と CSlA が冗長な sum 部と不充分な carry 部を持っており、それらが取り除かれるべきであることを実証する。さらに我々は、キャリー生成のための Brent-Kung 加算器のような prefix 加算を改善する。ここで取り上げるアイデアは回路実装技術を独立させるものであり、それゆえすべての回路実装・技術に有用である。

[1]信号の伝達を開閉する FET を用いたスイッチで、FET のソースドレインが伝達方向になるように挿入する。CMOS の場合は p-ch MOSFET と n-ch MOSFET を並列に入れ、p-ch MOS ゲートには制御信号を反転したものを入れる。

1 Introduction

数は 2 の補数表現または符号付き冗長 2 進数[2]として加算される。桁上げは、普通のキャリーのほかに、Ling による多様なキャリーがある。基本的な論理演算は

- prefix 演算器
- fco 演算器
- マルチプレクサ (MUX)
- AND-OR ゲート
- 複合ゲート
- パストランジスタ

によって実現される。加算器は、高速、低消費電力、小回路面積などの目的に応じて設計され、バイポーラ、ダイナミックロジック、ドミノロジック、CMOS、BiCMOS などの技術を用いて実装される。

[2]通常の 2 進数の基数 $\{0, 1\}$ に $\{-1\}$ を加えた数表現。これによってキャリーの伝播を防ぐことができる。

この論文の *first contribution* は、加算器が sum 部と carry 部に別けられるということを示すことである。CSlA、CSuA、およびそれらに基づいた複合型加算器は sum 部が冗長であり、簡単化されるべきである。

この論文の *second contribution* は、キャリー生成のための prefix 演算器、fco 演算器、MUX、AND-OR ゲート、パストランジスタなどの異なる演算器における等価性を示すことである。

加算器のキャリー部を一様化することによって、加算器同士の比較が容易になり、実装技術も簡単化される。実装技術は基本的なキャリー演算器がどう実装されるかということと、その最大のファンイン・ファンアウトを決定する。キャリー部の最適な設計はほとんどがキャリー部の影響を受け、異なるファンイン・ファンアウト・遅延となる。実装技術におけるファンイン・ファンアウトの限界と遅延に関する最適化問題に対しての研究がほとんどないため、クリティカルパスが増加するような加算器が多い。

我々は最終的に、MSB のキャリーを求めることが効果的なキャリー部の設計につながることを示す。MSB のキャリーが生成されるとき、他の幾つかのキャリーも副産物的に求められている。このアプローチはレイヤーキャリー生成と呼ばれ、他のビットとのキャリー生成の共有を最大限に実現できる。元来の CLA 加算器はゲートを共有することなくすべてのビットでのキャリーを並列に求めている。旧来の block-based CLA 加算器はキャリーの共有が充分でない。canonic 加算器は実装が AND-OR に限定されており、AND の共有も限定されている。我々のレイヤーキャリー生成では、わざわざ単体のゲートを作らなくても多くのビットのキャリーが求められる。このデザインは CSuA や CSlA よりも全面的に優れているため、CSuA と CSlA はもう使う必要はないだろう。

以下、第 2 章以降の論文の構成。

第 2 章 加算器の要素についての背景
第 3 章 carry/sum 部の分離とキャリー演算の等価性
第 4 章 レイヤーキャリー生成アルゴリズムの有用性
第 5 章 まとめ

2 Background

この論文では以下に示すような性質や定義を用いる。加算に用いられる引数は $A = \{a_{n-1}, \ldots, a_1, a_0\}$ と $B = \{b_{n-1}, \ldots, b_1, b_0\}$ であり、加算結果は $S = \{S_{n-1}, \ldots, S_1, S_0\}$ である。キャリーの伝播・生成信号は $p_i = a_i + b_i$ および $g_i = a_i b_i$ として定義される。なお、ここで、$p_i g_i = g_i$ 、$p_i + g_i = p_i$ という等式が成立する。他の文献では $k_i = \overline{a_i + b_i}$ や $p_i = a_i \oplus b_i$ という表現もある。

表 1: 伝播・生成の真理値表

a_i	b_i	p_i	g_i	k_i	p_i
0	0	0	0	1	0
0	1	1	0	0	1
1	0	1	0	0	1
1	1	1	1	0	0

キャリー入力信号は $c_{in} = c_{-1}$ によって表され、桁上げ出力信号 (c_{out}) は MSB で生成されるキャリーである。i ビット目で生成されるキャリーは一般的に $c_i = g_i + p_i c_{i-1} = g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + \ldots + p_i p_{i-1} \ldots p_i p_0 c_{-1}$ であり、$S_i = a_i \oplus b_i \oplus c_{i-1}$ である。他に、 Ling によるキャリーや、冗長 2 進数 (?) に基づいたアルゴリズムのためのキャリーなどのバリエーションがある。

加算器は以下のように分類される。

- 桁上げ伝播加算器 (RCA)
- 桁上げ先見 (CLA) 加算器
- prefix 加算器
- canonic 加算器
- block-based(多段型) CLA 加算器
- 桁上げ飛び越し加算器 (CSkA)
- conditional-sum 加算器 (CSuA)
- 桁上げ選択加算器 (CSlA)
- 複合型加算器

このように、全部で 9 種類の異なる加算器が挙げられる。

RCA は、単位時間当たり 1 ビットのキャリーを生成する。キャリー c_i は c_0〜c_{i-1} のキャリーがすべて

生成されてからしか生成されない。一方、 **CLA 加算器** はすべてのキャリー c_i を並列に生成する。ただし、これは理論上可能ではあるが、入力ゲート数が増えて実用的ではない。CLA 加算器の実現方法としては、異なるキャリーを共有することが可能な prefix 操作を取り入れることである。**prefix 加算器**は主に Ladner ·Fischer 加算器、Koggle-Stone 加算器、Brent-Kung 加算器といったものである。**canonic 加算器**は $p_ip_{i-1}\dots g_j$ を出力するための AND ゲートや、 $c_i = g_i + p_ig_{i-1} + p_ip_{i-1}g_{i-2} + \dots + p_ip_{i-1}\dots p_ip_0c_{-1}$ を出力するための OR ゲートなど、多入力の AND・OR ゲートを用いて並列キャリー生成を実装したものである。canonic 加算器は $p_ip_{i-1}\dots g_j$ といった論理積を異なる c_i と共有することがキーとなるアイデアである。prefix 加算器と canonic 加算器は主として遅延を抑えるために設計されており、例えば canonic 加算器は $2\lceil \log_r (n-1) \rceil + 2 - \delta$ の遅延であり、また大きな回路規模が要求される。

ハード面でのトレードオフにより、遅延は増大する。

block-based(多段型) CLA 加算器は n ビットの入力を通常 4 入力ずつのグループに分割して、各グループを CLA の論理で構成したものであり、グループ同士は RCA と同じように接続される。

桁上げ飛び越し加算器 (CSkA) もグループ分割によ

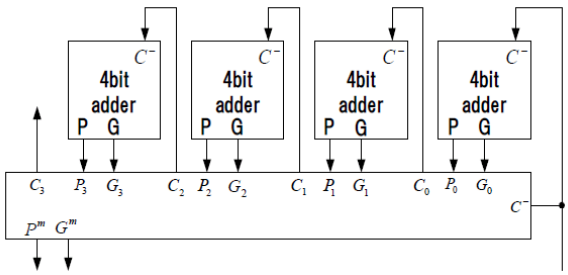

図 1: 多段型 Carry look-ahead adder

るものである。

各グループでは、すべてのステージで $p_j = 1$ になったときのみグループ桁上げ伝播信号が生成される。

conditional-sum 加算器 (CSuA) は、キャリーの選択を再帰的に行うアルゴリズムである。遅延時間は $O(\log n)$ となるが。その分多数のマルチプレクサを必要とする。**桁上げ選択加算器 (CSlA)** は、加算器をい

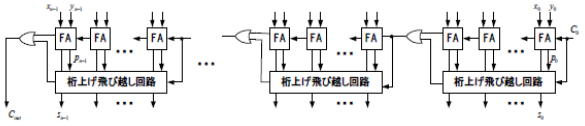

図 2: Carry skip adder

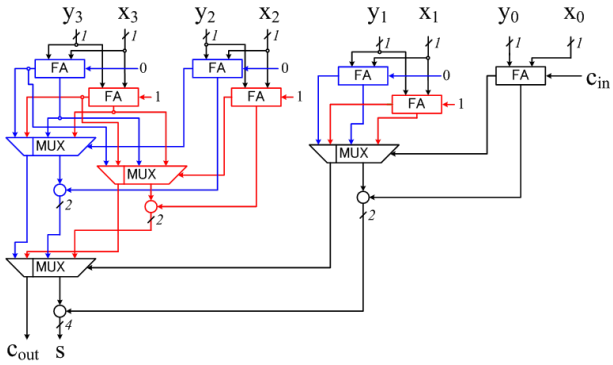

図 3: 4-bit conditional sum adder

くつかのブロックにわけ，ブロックごとにキャリーを選択するアルゴリズムである。これらは、下位桁からの

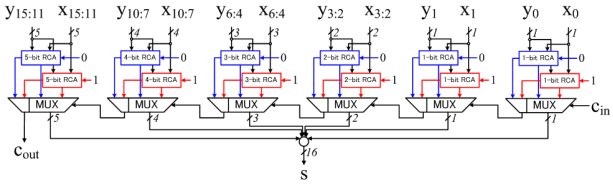

図 4: 16-bit carry select adder

キャリーが 0 の場合と 1 の場合の加算結果をあらかじめ両方計算しておき、キャリーが確定した時点で正しい加算結果を選択していくことで高速化を図った算術アルゴリズムである。**複合型加算器 (hybrid adder)** は異なる 2 つのアルゴリズムを組み合わせたものであり、たとえば CLA+CSuA や、CLA+CSlA などの組み合わせである。

これらの 9 つの異なる加算器は遅延に応じて設計されている。その上、幾つかの加算器のアルゴリズムでは $S_i = a_i \oplus b_i \oplus c_{i-1}$ という標準的な式によって求められる。現在の加算アルゴリズムの分類は論理的なものというより歴史的なものであるが、それらの異なる

加算器の間には論理的な関係性が包括的に示されたことがない。第 3 章では、それぞれの加算器が sum 部と carry 部に別けられることを示す。

加算器におけるキャリー部は、prefix 演算器、fco 演算器、MUX、AND-OR ゲート、複合ゲート、パストランジスタなどの演算器によって実装できる。

Ladner と Fisher が提案した **prefix 演算子** $\bullet$ は $\left(\frac{g}{k}\right)_i \bullet \left(\frac{g}{k}\right)_j = \left(\frac{g_i + \bar{k}_i g_j}{k_i k_j}\right)$ または $(g_i, p_i) \bullet (g_j, p_j) = (g_i + p_i g_j, p_i p_j)$ として定義される。キャリー c_i は $c_i = \left(\frac{g}{k}\right)_i \bullet \left(\frac{g}{k}\right)_{i-1} \bullet \ldots \bullet \left(\frac{g}{k}\right)_0 \bullet \left(\frac{c_{-1}}{1}\right)$ として計算される。Brent と Kung による **fco 演算子**は $(p_{i:j}, g_{i:j}) = (p_{i:k+1}, g_{i:k+1}) fco(p_{k:j}, g_{k:j})$ によって定義される。($p_{i:j} = p_{i:k+1} p_{k:j}$ 、 $g_{i:j} = g_{i:k+1} + p_{i:k+1} g_{k:j}$ である。) キャリー c_i は $(p_{i:0}, c_i) = (p_{i:i-1}, g_{i:i-1}) fco \ldots fco(1, c_{-1})$ として計算される。

(*function graph* については省略。)

3 Unified Adder Design

この章では、RCA、CSuA、CSlA が sum と carry のパートに分けることができることを示す。CSlA/CSuA を用いた複合型加算器も同様に分けることができる。他のアルゴリズムは総てキャリーを生成するのみである。それゆえ我々はすべてのキャリーを先に求めるように一様化する。

3.1 Carry Logic and Sum Logic

Figure 2:
(b) 4 ビット RCA
(c) sum 部
※灰色の円:XOR
(d) carry 部

Figure 3:
(a) 8 ビット CSuA
※灰色の円:sum 部 (XOR)、矩形:carry 部 (MUX)
(b) XOR を移動することによって灰色で示された MUX を取り除くことができた
XOR による遅延は増えたが、MUX の数は 28 → 17 と減少した。規模の小さい部分ではクリティカルパスとはならないので、上位 3 ビット分の回路構造を元の形に戻せば、遅延は据え置きのまま MUX を 5 つ取り除ける。

Figure 4:
(a) 4 ビット CSlA
※灰色の円:sum 部 (XOR)、矩形:carry 部 (AND-OR/MUX)
(b) 余分な XOR を取り除いた回路

3.2 Uniformed Logic Operator for Carry Logic

加算器の重要な性質のひとつは、 $g_i < p_i$ という関係性であり、これは $g_i + p_i = p_i$ 、$g_i p_i = g_i$ ということを暗に示している。そしてキャリー生成には $c_i = g_i + p_i c_{i-1} = \overline{c_{i-1}} g_i + c_{i-1} p_i$ という関数が重要である。我々はこの基本的な論理演算を**キャリー演算**と呼ぶ。

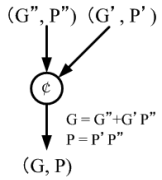

図 5: Carry operator

$c_0 = g_0 + p_0 c_{-1} = \overline{c_{-1}} g_0 + c_{-1} p_0$ より、1 ビット加算器は MUX または AND-OR ゲートを用いて、Figure 5 のように作ることができる。同じくして、Figure 4 の CSlA においても AND-OR ゲートを MUX に置き換えることができる。

fco/prefix 演算も、 $g_{i:k} < p_{i:k+1}$ より $g_{i:j} = g_{i:k+1} \overline{g_{k:j}} + p_{i:k+1} g_{k:j}$ となり、MUX に置き換えることができる。Figure 6 は Brent-Kung による fco 演算を用いた 4 ビットのキャリー生成を、MUX をベースに実装したものである。MUX を用いた新しい実装方法によって、論理構造を保ったまま多くの無駄を省けることが容易に見てとれる。

Figure 7 は桁上げ伝播の連鎖をさまざまな演算器を用いて実装した図である。
(a) 桁上げ伝播のブロック図 (b) MUX (c) AND-OR
(d) MUX-AND を用いた一般的な prefix 演算

(e) 一般的な prefix 演算 (f) パストランジスタ
その他、マンチェスタ・キャリー連鎖 (MCC) や複合ゲートを用いても簡単に実装できる。

3.3 Unified Adders with Uniformed Carry Operator

これまで、sum 部と carry 部の 2 パートの分離について示した。その上、carry 部がさまざまなアルゴリズムの中で同じキャリー演算が用いられていることも示された。このような一様化によって多様なアルゴリズムの比較・改善を行うことができる。

figure 8 は 4 ビットの RCA(a:左上)、CSlA(b:左下)、CSuA(c:右) をそれぞれ示したものである。色付きのブロックが sum 部の XOR に等しく、その他はキャリー演算である。4 ビット加算器は大規模な加算器の基礎となるため、4 ビット加算器の最適化は非常に重要である。これらの改善は第 4 章で行う。

4 Layered Carry Generation

一様化された sum 部と carry 部がキャリー演算を用いて実現される一方、先ほど紹介したほぼすべての改良アルゴリズムで、
(1) グループ内のキャリーは MSB から生成される
(2) 例えば $O(\log n)$ の遅延に対して CSuA、$O(n)$ の遅延に対して CSlA というように、遅延の大きさに対してアルゴリズムの分類ができる
といった 2 つの性質を持っている。

本章では、可変の遅延およびファンアウトに応じたレイヤーキャリー生成の構想を提案する。上位ビットでのキャリーと、それを求める際に副次的に生成される下位ビットのキャリーを併せて”レイヤー”と呼ぶ。それゆえ、レイヤー内のキャリーはグループを基本としたアルゴリズムのようにキャリーが立て続けに生成されるわけではない。

g_i, p_i が i 毎に求められるという前提で、以下のアルゴリズムでは全てのキャリー c_i を生成する。このアルゴリズムで (n, d, f) の 3 変数を用いる。なお、n はビット数、d はキャリー演算子の数という意味での遅延、f はテクノロジが与えるファンアウト数の上限値を表す。

Layered-Carry Generation Algorithm - (n, d, f)
(1) 最上位キャリー c_n はキャリー演算を用いて構成される。レイヤー内の他のキャリーも同様に生成される。d が小さいときは、分割統治法などを用いて c_n の遅延を d に近づける。
(2) まだレイヤーに含まれないキャリーの最上位ビットについて (1) を繰り返す。
(3) キャリー演算を複合ゲートや CMOS などで実装するためにキャリー演算を最も効果的な場所に微調整する。

続けて、上記のアルゴリズムでいかにキャリーを構成するか、遅延が最小限の場合とそうでない場合について例を示す。

4.1 Minimal Delay Layered Carry Generation

Figure 9 の (a) はレイヤーキャリー生成アルゴリズムを用いた新しい構成の 4 ビット加算器である。我々は遅延を抑えるために分割統治法を用いて c_3 を生成した。c_3 を生成する際、c_0, c_2 も同時に生成されており、残っているのは c_1 のみである。これは新たにキャリー演算器を設けることによって求められる。よって、4 つのキャリーは (c_3, c_2, c_0) と c_1 の 2 つのレイヤーに分かれる。

Table 1 からもわかるように、レイヤーを用いた加算器は遅延の最小化を実現しながら他の加算器に比べてゲート数もファンアウト数も少ない。

Figure 10 は (a) に CSuA が示されており、(b) はそれを改善したものである。レイヤーは $(c_7, c_6, c_2, c_0), (c_5, c_4, c_2), (c_3)$ である。最後に、Figure 11 はレイヤーを用いた 16 ビット加算器で、最上位のレイヤーは $(15, 14, 6, 2, 0)$ である。

ここでひとつ注意すべきは、c_5 が Figure 10(b) と Figure 3(b) で異なることである。これは、後者がキャリー入力 c_{-1} を用いており、遅延が 1 増えているためである。遅延が 4 と大きいため、c_5 を求めるための別個のキャリー演算が不要で、c_4 を生成した後に c_5 を生成すればいいのである。グループを基本としないキャリー構成アルゴリズムを用いることによって、このように最適な結果を得ることができるのである。

以上の例で、同じ遅延の下においてレイヤーキャリー生成が CSuA と比べてあらゆる面で優れていることが

示された。それゆえ、我々は CSuA と CSlA の使用は避けるべきであると考える。また、それらに基づいた複合型加算器も同様に改善されるべきである。

4.2 Non-Minimal Delay Layered Carry Generation

我々のレイヤーキャリー生成アルゴリズムは、他のアルゴリズムとは極めて異なる方法で、遅延に依存しない加算器にも適用できる。Figure 12(a) は 28 個のキャリー演算器を要する元々の Brent-Kung の構成であり、(b) は 23 個のキャリー演算器を要する、原理的に等しい改良版の構成である。最上位のレイヤーは $(15, 14, 13, 11, 7, 3, 1)$ である。ここで用いている演算器は普通のキャリー演算子であり、必ずしも prefix 演算器でなくてもよいことに注意されたい。例えば、MUX や AND-OR で置き換えてよい。

5 Conclusions

- 加算器が sum 部と carry 部に分割できることを示した。
- いくつかの加算器は冗長な sum 部と最適でない carry 部をもっていることを示した。
- キャリー生成の基本的論理には等価性がある。
- レイヤーを用いた効果的なキャリー生成の手法を示した。
- ここで提案されたアイデアは実用的である。

参考文献

[1] 算術アルゴリズム記述言語 ARITH のページ
http://www.aoki.ecei.tohoku.ac.jp/arith/mg/algorithm.html

[2] 水口 貴之, 「算術演算器の速度評価」 高知工科大学 電子・光システム工学科 卒業研究報告, 2005.2

[3] 味元 伸太郎, 「演算器の消費電力についての形式別の評価と検討」 高知工科大学 電子・光システム工学科 卒業研究報告, 2005.2