社団法人　情報処理学会　研究報告
IPSJ SIG Technical Reports

2007－ＳＬＤＭ－132（28）
2007／11／22

Lingのキャリー計算に基づく parallel prefix adder 合成について

松永多苗子†　木村　晋二†　松永　裕介††

† 早稲田大学大学院情報生産システム研究科　〒808-0135 北九州市若松区ひびきの 2-7
†† 九州大学大学院システム情報科学研究院　〒819-0395 福岡市西区元岡 744
E-mail: †t_matsunaga@akane.waseda.jp

あらまし　Ling adder は隣り合う２ビットずつをまとめたキャリー計算に基づく加算器であり、このキャリー計算は prefix 計算として通常の parallel prefix adder と同様の定式化が可能である。本論文では、通常のキャリー生成、伝搬関数に基づいた parallel prefix adder 合成手法を Ling キャリーに対して適用することによって Ling adder を生成し、両者を比較することによって、方式の違いによる差異、および、それを採り入れた合成手法の評価を行う。
キーワード　parallel prefix adder, Ling adder, 演算器合成

Synthesis of parallel prefix adders based on Ling's carry computation

Taeko MATSUNAGA†, Shinji KIMURA†, and Yusuke MATSUNAGA††

† Graduate School of Information, Production and Systems Waseda University　2-7 Hibikino, Wakamatsu-ku, Kitakyushu-shi, Fukuoka 808-0135, JAPAN
†† Faculty of Information Science and Electorical Engineering, Graduate School of Kyushu University 744 Motooka, Nishi-ku, Fukuoka 819-0395, JAPAN
E-mail: †t_matsunaga@akane.waseda.jp

Abstract　Ling adders calculate carry propagation based on adjacent bit pairs, and can be formulated as parallel prefix adders. In this paper, our synthesis framework for usual parallel prefix adders based on carry-generate and propagate functions is extended to treat Ling' carry. Some experimental results are shown to discuss its effectiveness to integrate into our framework.
Key words　parallel prefix adder, Ling adder, arithmetic synthesis

1. はじめに

加算、乗算等の算術演算回路は、回路全体の品質に影響を与え得る重要な要素であり、その構成については多くの研究がなされている [3]。その中で、parallel prefix adder(PPA) は、桁上げ先見の概念を一般化した手法を用いて桁上げ伝搬を高速化するタイプの加算器で、様々な特徴をもった構造が提案されている [1], [2], [5]。また、構造をあらかじめ固定するのではなく自動生成するアルゴリズムも提案されている [4], [6], [8]。自動合成アルゴリズムが優位な点は、ある加算器が使用される個別の状況に合わせて回路構造を柔軟に構築可能であることで、特に、ビット毎のタイミング制約が異なるような状況で、それを利用して回路をより小さく、あるいは速くすることが可能である。

PPA におけるキャリー計算は、キャリー生成、伝搬関数を基本演算として実現され、その演算順序を変えることで演算量や実行時間が変わる。演算順序は、各演算をノードに対応させた prefix graph によって表すことができ、グラフの各ノードを、その演算を行う回路要素に対応づければ、そのまま回路構造を表していると考えられる。筆者らは、このグラフ構造を加算器のテクノロジ非依存レベルの概略構造として、概略構造レベルの最適化手法を提案してきた [8]。

一方、キャリー計算の高速化手法は他にも存在する。Ling は、隣りあうビットのキャリーをひとまとめにして扱うことで、キャリーの計算を簡単にする方式を提案している [11]。この計算も prefix 計算として、通常の PPA と同様の定式化をすることができることが知られている [12]。キャリーの計算が通常の方式よりも簡単になることで高速になると言われているが、実際には回路の実現方法等に影響される。

本稿では、概略構造レベルでの prefix graph 合成手法を Ling のキャリー計算方式を扱えるように拡張し、通常のキャリー計算方式を用いた場合と比較することによって、 扱うキャリー方式による差異や、それを採り入れた合成手法の評価を行い、それらを通じて、より適切な合成手法の検討を行うものである。以下、まず 2 章において、PPA に関する諸定義、および、

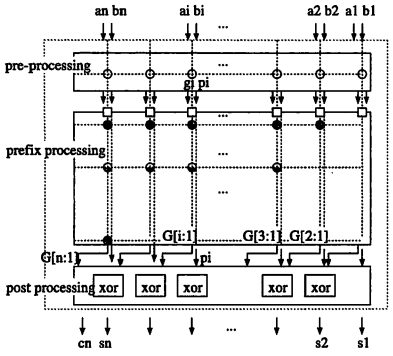

図 1 PPA
Fig. 1 PPA

PPA 合成手法を示す。3章では、Ling のキャリー計算方式を説明し、その方式に対応した PPA 合成手法を示す。実際に2つのキャリー計算方式で加算器を合成し比較した結果を4章で示し、結果の評価および考察を行う。

2. Parallel prefix adder 合成

2.1 Parallel prefix adder

n ビットの2進加算の入力を $A=a_n,\cdots,a_1, B=b_n,\cdots,b_1$, 出力を和 $S=s_n,\cdots,s_1$, 及び、キャリー出力 c_n とすると、各ビットの和 s_i とキャリー出力 c_i は、$s_i=a_i\oplus b_i\oplus c_{i-1}$, $c_i=a_ib_i+a_ic_{i-1}+b_ic_{i-1}$ で計算される。この n ビット加算は、個々のビットに対する桁上げ生成/伝搬関数 (g,p) と、その概念をビットグループに対して拡張した、グループ桁上げ生成/伝搬関数 (G,P) を用いると以下のように計算できる。

- g,p の生成: $G_{[i:i]}=g_i=a_i\cdot b_i$, $P_{[i:i]}=p_i=a_i\oplus b_i$
- prefix 処理: G, P を用いて $c_i=G_{[i:1]}$ を計算

$$G_{[i:j]}=G_{[i:k]}+P_{[i:k]}\cdot G_{[k-1:j]}$$

$$P_{[i:j]}=P_{[i:k]}\cdot P_{[k-1:j]}$$

- s_i の生成: $c_i=G_{[i:1]}$, $s_i=p_i\oplus c_{i-1}$

Parallel prefix adder は上記の3つの処理を行う要素から構成される加算器である (図1)。ビットごとの g_i, p_i の生成、及び、s_i の計算部分の概略構成には自由度はないため、$G_{[i:1]}$ の処理部分が加算器全体の性能に影響することになる。

2.2 Prefix computation と prefix graph

Prefix computation とは、N 個の入力 $x_N, x_{N-1},\cdots,x_1$ と結合則を満たす任意の演算 $\circ$ が与えられたとき、N 個の出力 y_i $(1\leqq i\leqq N)$ を、$y_i=x_i\circ x_{i-1}\circ\cdots\circ x_1$ によって計算するものである。これは、i 番目の出力 y_i は、$j\leqq i$ となる入力 x_j のみに依存することを意味する。

N 入力 prefix graph は、N 個の入力に対する prefix computation における各演算 $\circ$ をノードとした非巡回有向グラフ (directed ascyclic graph: DAG) であり、各出力の値を計算するための演算の実行順序を表現したものである。ノードのファンインは2項演算 $\circ$ の2つのオペランドに対応し、ノード v_1 が v_2 のオペランドであるとき、v_1 から v_2 へのエッジが存在する。v_1 を v_2 のファンインノード、v_2 を v_1 のファンアウトノードと呼ぶ。Prefix graph の入力数 (および出力数)N を、prefix graph の幅と呼ぶ。

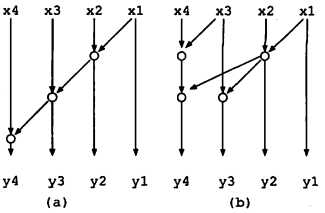

図 2 幅 4 の prefix graph の例
Fig. 2 Examples of prefix graphs: width = 4

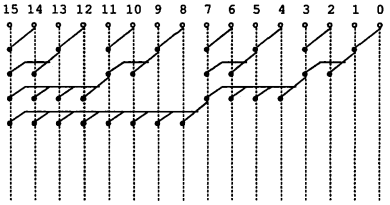

図 3 Sklansky
Fig. 3 Sklansky

図2 は幅4の prefix graph の例を示す。(a) の y_4 は、$y_4=x_4\circ(x_3\circ(x_2\circ x_1))$ という演算順で計算され、(b) では、$y_4=(x_4\circ x_3)\circ(x_2\circ x_1)$ で計算される。Prefix computation は結合則が成り立つため、どちらの順序で計算しても結果は変わらない。

上述の PPA のキャリー計算において、prefix 処理の部分は、演算 $\circ$ を以下のように定義することによって、prefix computation とみなせ、その演算順序は prefix graph で表現することができる。

$$\begin{aligned}(G,P)_{[i:j]} &= (G,P)_{[i:k]}\circ(G,P)_{[k-1:j]}\\ &= (G_{[i:k]}+P_{[i:k]}\cdot G_{[k-1:j]},\\ &\quad P_{[i:k]}\cdot P_{[k-1:j]})\end{aligned}$$

図3 は Sklansky 型 PPA の概略構造を表した prefix graph である。

2.3 PPA 合成アルゴリズム

Parallel prefix adder の概略構造は prefix graph として表せ、様々な特徴をもつ prefix graph を探索することによって、加算器の構造を抽象度の高いレベルで探索することが可能になる。このテクノロジに独立なレベルでの探索結果は、最終的にはテクノロジマッピングの処理を経て、対象とするテクノロジライブラリのセルにマップされる必要がある。

著者らは PPA 合成を、prefix graph を対象とした概略構造の探索と、その結果のセルへのマッピングの2段階に分けて、まずは prefix graph を対象とした合成問題に取り組んでいる。Prefix graph に対する面積、遅延の指標としては、現在のとこ

ろグラフのノード数と段数という、単純化したモデルを用いている。

Prefix graph は DAG であり、実用的なビット幅の prefix graph に対して遅延制約下での面積最小化の厳密解を効率よく得るのは困難である。そのため、既存の手法では、まず各ノードの到達レベルが最小になるような構成を求めてから、タイミング制約に余裕がある範囲で、面積を削減するための局所的な構造変換を行う、という手法が用いられている [4], [6]。これに対して本手法では、面積最小化に対してより大局的な視点で取り組むことを目指して、2つのプロセスからなるアルゴリズムを提案する。まず、最初のプロセス、動的計画法に基づく面積最小化 (Dynamic Programming based Area Minimization: DPAM) では、prefix graph のある部分集合に集中して最小解の探索をする。探索する部分集合を適切に定義することにより、その部分集合に対して効率よく動的計画法を適用することが可能になり、厳密最小解が得られる。ただしこの厳密性は部分集合に対するもので、prefix graph 全空間を対象とした場合には、まだ面積削減の余地が残っている。そこで、次のプロセス、再構築による面積削減 (Area Reduction with Re-Structuring:ARRS) において、構造に対して与えた制限を取り外してより小さな prefix graph となるよう再構築する。部分集合を適切に定義し、その性質を利用することで、実用的なビット幅の問題に対して、現実的な処理時間で結果が得られている。

3. Parallel prefix Ling adder の合成

以下では、Ling のキャリー計算、および、PPA としての定式化に関する概念について、例を通して概要を示す。

3.1 Ling のキャリー方式

Ling のキャリー H_i は、以下の式で計算される：

$$H_i = c_i + c_{i-1}$$

これは、隣り合うビット対 $(a_i, b_i), (a_{i-1}, b_{i-1})$ のキャリーをひとまとめにすることで、キャリーの計算式を簡単化し、それにより計算速度を速くする、という概念に基づいている [11]。

以下では、キャリー伝搬関数として $p_i = a_i + b_i$ を用いる。例として c_3と c_2 を考えるとする。

$$\begin{aligned} c_3 &= g_3 + g_2 \cdot p_3 + g_1 \cdot p_2 \cdot p_3 + g_0 \cdot p_1 \cdot p_2 \cdot p_3 \\ c_2 &= g_2 + g_1 \cdot p_2 + g_0 \cdot p_1 \cdot p_2 \\ H_3 &= g_3 + (g_2 + g_2 \cdot p_3) + (g_1 \cdot p_2 + g_1 \cdot p_2 \cdot p_3) \\ &\quad + (g_0 \cdot p_1 \cdot p_2 + g_0 \cdot p_1 \cdot p_2 \cdot p_3) \\ &= g_3 + g_2 + g_1 \cdot p_2 + g_0 \cdot p_1 \cdot p_2 \end{aligned}$$

c_3 と H_3 を比較すると、積項数は同一であるが、リテラル数は減っており、論理が簡単になっているため、通常より簡単な高速な回路として実現されると考えられる。ただし、最終的な和の計算が通常の方式より複雑になる。$c_i = p_i \cdot H_i$ であるので、

$$\begin{aligned} s_i &= (a_i \oplus b_i) \oplus c_{i-1} && (1) \\ &= (a_i \oplus b_i) \oplus (p_{i-1} \cdot H_{i-1}) && (2) \end{aligned}$$

これは以下の式に変形できる：

$$s_i = \bar{H}_{i-1} \cdot d_i + H_{i-1} \cdot ((a_i \oplus b_i) \oplus p_{i-1})$$

したがって、和の計算に対して、マルチプレクサが余分に必要になる。ただし、最大遅延時間に関しては、通常 は H_i の方が他の信号よりも到達時刻が遅いので、影響を与えることはないと考えられる。

3.2 Parallel prefix adder としての定式化

Giorgos らは、隣りあうキャリー伝搬、生成関数を以下のように組み合わせることによって、Ling のキャリー計算が parallel prefix adder の枠組で定式化できることを示している [12]:

$$G_i^* = g_i + g_{i-1} \quad (3)$$

$$P_i^* = p_i \cdot p_{i-1} \quad (4)$$

ここで、i が負の範囲にある場合は、$g_i = p_i = 0, G_i^* = P_i^* = 0$ とする。

前述の H_3 に対して、$g_i \cdot p_i = g_i$ を用いて書換えると

$$\begin{aligned} H_3 &= (g_3 + g_2) + (g_1 + g_0)p_2p_1 && (5) \\ &= G_3^* + G_1^* \cdot P_2^* && (6) \end{aligned}$$

となり、結果的に (G_i^*, P_{i-1}^*) という対を用いれば、Ling のキャリー計算は通常の prefix adder と同じ基本演算を用いて計算でき、かつ、偶数ビット群と奇数ビット群に分割して処理することが可能となる。

$$H_i = (G_i^*, P_{i-1}^*) \circ (G_{i-2}^*, P_{i-3}^*) \circ \cdots \circ (G_0^*, P_{-1}^*) \quad (7)$$

$$H_{i+1} = (G_{i_1}^*, P_i^*) \circ (G_{i-1}^*, P_{i-2}^*) \circ \cdots \circ (G_1^*, P_0^*) \quad (8)$$

すなわち、偶数ビットと奇数ビットで分割して、それぞれに対して計算を行えばよいことになる。

3.3 Parallel prefix Ling adder 合成

図 4 に Ling のキャリー計算方式に基づく parallel prefix adder の構成を示す。通常の prefix adder (図 1) との違いは、

- 前処理部分と後処理部分の構成
- prefix computation 部分への入力タイミング
- prefix computation 部分が奇数ビット用、偶数ビット用に 2 分割されること

である。prefix graph 合成手法自体はそのまま適用できるが、適用の対象として偶数ビットあるいは奇数ビットだけを抽出したグラフを対象とすることになる。その際、ビットごとに与えられた入力制約を、前処理部分の構造にしたがって、適切に再計算する必要がある。個々の prefix graph に対して、それぞれの制約下での面積最小化を行った後、前処理、後処理部分と合わせて、加算器全体の構造を verilog 記述として出力する。

4. Parallel prefix adder 合成におけるキャリー方式の比較実験

キャリー計算方式の違いによって、本 prefix graph 合成手法の結果がどのように影響を受けるかを評価するための実験を行った。

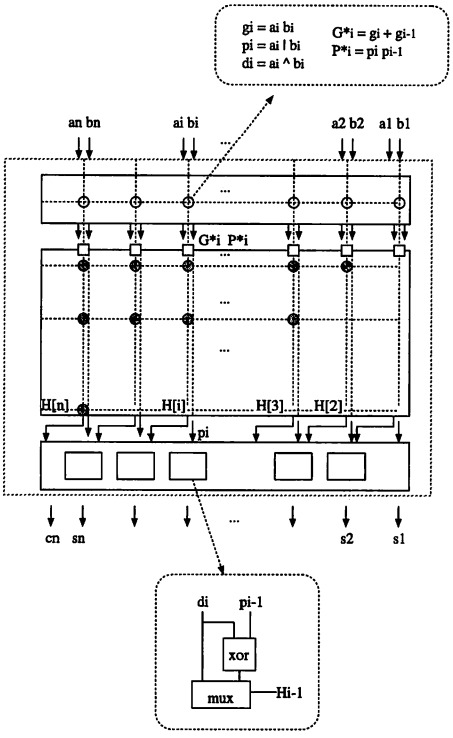

図 4 Ling のキャリー計算に基づく parallel prefix adder
Fig. 4 Parallel prefix Ling adder

- 入力タイミング制約下で、最小段数で実現するという制約の下で、ノード数を最小化する。
- ただし、Ling の場合 prefix graph 部分は 1 段小さくなるので、通常版と同じ段数を出力とした場合も実行。
- 入力制約は、ビットごとに到達レベルがばらつく場合と均一な場合で実験
  - 均一な場合：32 ビット、64 ビット
  - ばらつく場合（凸型）：32 ビット, 64 ビット、24 ビット、55 ビット、117 ビット（後半 4 つは、それぞれ、16bit, 32bit, 64bit Wallace-tree 乗算器の最終段加算器）

実験結果は、まず prefix graph のレベルで評価し、一部データについては、その結果に対して論理合成を実行し、ゲートレベルでの比較を行った。

4.1 実験結果：概略構造レベル

表 1 に、入力タイミングが一様な制約条件下での実験結果を示す。表の中で、32 および 64 はビット幅、L/N はそれぞれ Ling、通常のキャリー計算方式を示す。SW は生成されたグラフの各ノードのスイッチング確率の総和を計算した値である。比率は、通常版とのノード数の比を表している。この実験では、入力到達レベルは一様で 0 とし、出力要求レベルは可能な最小段数とした上で、ノード数最小化を対象とした合成を行った。Ling 方式の場合、prefix graph 部分の段数は 1 段小さくなるため、通常版と同じ要求時刻を設定した場合に付いても合成を行った。最小段数での結果を比較すると、Ling 方式 (32L0, 64L0) の方が、通常版 (32N, 64N) よりも prefix graph 部分の段数は 1 段少なく、ノード数も 12-16%少なくなった。要求レベルを通常版と同じにした場合、ノード数は 30%以上減少した。prefix graph の合成だけをみるならば、Ling 方式を用いることにより、より段数の小さい、あるいは、制約の中でよりノード数の小さい結果が得られている。ただし、prefix graph 以外の部分の面積が Ling 方式では増えているので、全体としては面積も SW も増加している可能性が高い。

表 1 実験結果（入力タイミングが一様な場合）

	段数	ノード数	比率	SW	処理時間
32L0	4	62	0.84	68.58	5.85
32L1	5	50	0.68	60.25	5.85
32N	5	74	1.00	72.66	12.24
64L0	5	148	0.88	150.33	24.43
64L1	6	112	0.67	129.09	24.57
64N	6	168	1.00	144.5	50.27

一方、入力到達レベルが一様でない場合の結果を表 2 に示す。こちらの場合も、入力制約を満たす中で段数最小値を出力制約として面積を最小化を行った。入力制約が一様である場合に比べ、ノード数の削減率は悪くなり、減っても数パーセントであり、逆に多くなる場合も見られた。原因として考えられるのは、Ling キャリーの場合、複数のビットから prefix graph の初段に対する初期信号が作られること、および、奇数ビットだけ、偶数ビットだけを取り出して合成の対象となる prefix graph を作成するため、合成対象となる prefix graph の入力到達レベルの形状がもとの形状と異なってくることである。本手法は凸型のプロファイルに対してよい結果がでることが実験的に示されているが、完全にランダムなバラツキが与えられた場合に対しての堅牢性は必ずしも保証されていない。

通常版とレベル数を同一にして実行すると、2-3 割削減できる。

なお、処理時間に関しては、制約が一様であるかどうかにかかわらず、PPA 合成アルゴリズムが対象とする問題のサイズが半分になるため、処理時間は 2 倍以上高速化されていた。

4.2 実験結果：ゲートレベル

上記で得られた prefix graph から verilog 記述を生成し、Rohm 0.18 ライブラリを対象として Design Compiler により論理合成を行った。表 3 に実験結果を示す。表中、第 2、3 行目は入力到達レベルが一様であった場合、残りの 4 つのケースは、凸型であった場合の結果である。

結果の回路の性質は、論理合成ツールに与えるタイミング制約や、簡単化処理のしかたによって大きく左右され、必ずしも prefix graph の構造自体の評価が純粋にできるとはいえない。今回は、上記で生成した二種類の方式による加算器が、どこまで速くできるかを見ることを目的として実行した。最適化処理としては、生成した概略構造は保持するが、隣り合うノード間の境界は取り除いた状況でマッピングを行っている。表 3 は、遅延を厳しい値から徐々に緩めて与えたなかで、得られた最小遅延とそのときの面積を示す。入力到達時刻が一様な場合、最

大遅延が 11 - 16%小さくなったが、面積は倍近くに増大してしまった。一方、遅延時間の削減率はビット幅が大きくなるほど小さくなる。これは、遅延の削減量が基本的には PG 上でノード 1 段分相当であるため、ビット幅が大きくなり実現に必要な段数が大きくなれば、相対的に減少割合は減ってくるからと考えられる。

凸型の場合も、最大遅延時間は通常版よりも小さくなるが、面積が 2 倍近くになっている。試しに遅延制約を緩めて、通常版と同程度に設定して合成してみたが、それでも面積は大きい方にふれていた。

表 3 実験結果（論理合成後）non-uniform

ビット幅	遅延 (L)	遅延 (N)	面積 (L)	面積 (N)
32	0.65	0.77	16070	8228
64	0.83	0.93	26053	14782
24	2.84	3.00	7141	4132
32	2.34	2.48	10703	5641
55	3.27	3.32	18234	10973
64	3.26	3.35	19983	11386

全体の結果を通して、parallel prefix Ling adder は、確かにより高速な加算器を実装できる可能性があるが、面積のペナルティが大きく、本手法にとっては、ビットごとのバラツキに対応できる、という柔軟性が、通常の PPA の場合ほど出せない可能性があることがわかった。また、全体の回路に対する prefix graph 部分の割合が小さくなるため、prefix graph 以外の部分の実装もふくめた合成のストラテジ、あるいは、マッピング方法を考慮する必要があると考えられる。

表 2 実験結果（入力タイミングが凸型の場合）

	段数	ノード数	比率	SW	処理時間
24L0	18	46	0.94	51.11	3.21
24L1	19	36	0.73	44.33	3.07
24N	19	49	1.00	50.40	6.75
32L0	15	55	0.98	62.70	6.21
32L1	16	42	0.75	54.42	5.72
32N	16	56	1.00	60.53	12.86
55L0	21	122	1.00	125.63	18.1
55L1	22	96	0.79	110.03	17.53
55N	22	122	1.00	119.52	38.78
64L0	21	146	1.07	145.94	24.49
64L1	22	95	0.69	116.19	24.03
64N	22	137	1.00	135.77	53.39
117L0	26	257	0.98	260.42	83.38
117L1	27	218	0.83	240.13	84.61
117N	27	263	1.00	253.81	177.14

5. おわりに

本稿では、prefix graph 合成手法を Ling のキャリー方式に対して適用し、方式の違いによる回路の品質への影響を比較することによって、本合成手法に Ling のキャリー計算方式を取り込むことへの課題等が確認された。今後の課題としては、ゲートレベルでのより詳細な比較や、回路全体に対するマッピングまで含めた合成手法の検討があげられる。

6. 謝　辞

ツールの使用などに関し、東京大学大規模集積システム設計教育研究センターおよび、シノプシス株式会社に感謝します。またライブラリについては、京都大学小野寺研究室および小林和淑准教授に感謝します。なお、本研究は一部、JSPS Global COE プログラムおよび JST CREST ULP プロジェクトによる。

文　献

[1] R. P. Brent and H. T. Kung. A regular layout for parallel adders. *IEEE Trans. Computers*, 31(3):260–264, March 1982.

[2] P. M. Kogge and H. S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Trans. Computers*, 22(8):786–793, August 1973.

[3] I. Koren. *Computer Arithmetic Algorithms*. A K Peters,

Ltd., 2002.

[4] J. Liu, S. Zhou, H. Zhu, and C.-K. Cheng. An algorithmic approach for generic parallel adders. In *ICCAD*, pages 734–730, November 2003.

[5] J. Sklansky. Conditional sum addition logic. *IRE Trans. Electron. Comput.*, 9(6):226–231, 1960.

[6] R. Zimmermann. Non-heuristic optimization and synthesis of parallel-prefix adders. In *International Workshop on Logic and Architecture Synthesis*, pages 123–132, December 1996.

[7] R. Bryant. "Graph-based algorithms for Boolean function manipulation", In IEEE Trans. Comput. C-35,8(Aug.), pp.677–691, 1986.

[8] Taeko Matsunaga and Yusuke Matsunaga, "Area Minimization Algorithm for Parallel Prefix Adders under Bitwise Delay Constraints", in ACM Great Lakes Symposium on VLSI(GLSVLSI'07), March 2007.

[9] Taeko Matsunaga, Shinji Kimura and Yusuke Matsunaga, "Power-Conscious Synthesis of Parallel Prefix Adders under Bitwise Timing Constraints", in SASIMI 2007, October 2007.

[10] 松永多苗子、松永裕介、"Parallel prefix adder 合成を用いた乗算器の最適化手法について", 第 20 回 回路とシステム軽井沢ワークショップ, 2007 年 4 月

[11] H. Ling, "High-speed binary adder", IBM J. R&D, 25:156-166, May 1981.

[12] Giorgos Dimitrakopoulos and Dimitris Nikolos, "High-Speed Parallel-Prefix VLSI Ling Adders", in IEEE Trans. Computers, vol.54, no.2, pp.225-231, 2005.