コンプレッサツリー生成のための一般化並列カウンタの網羅的探索

野田 麦[†1,a)]　石浦 菜岐佐[†2,b)]

概要：本稿では多入力加算器の効率的な FPGA 実装を目的に, まだ発見されていない一般化並列カウンタ (Generalized Parallel Counter; GPC) を網羅的に探索する手法を提案する. 多入力の加算器は乗算回路や積和演算回路のコアであり, FPGA をターゲットに効率的な多入力加算器を実装する手法として, 全加算器を拡張した GPC の木を構築する手法が提案されている. Xilinx 7 シリーズの 1 スライスで実現可能な GPC はこれまでに 3 種類知られているが, 本稿では 1 スライスで実現可能な GPC を網羅的に列挙することにより, 利用可能な新たな GPC の発見を目指す. 出力が 5 ビット以下の GPC の全ての入出力仕様を列挙し, それぞれについて入力から LUT への接続及び LUT の真理値表が存在すればそれを求める. 提案の探索手法を Rust で実装した結果, 探索は約 100 秒で完了し, これまでに知られていなかった 5 種類の GPC (1,2,6;4), (4,2,5;5), (1,2,4,4;5), (1,3,1,6;5), (1,3,3,4;5) を発見した. また, これらの GPC を用いて 8 ～ 32 ビットの乗算回路と多入力加算回路を構成した結果, 2 つの回路でスライス数を, 2 つの回路で段数を削減することができた.

Enumeration of Generalized Parallel Counters for Compressor Tree Synthesis

Mugi Noda[†1,a)]　Nagisa Ishiura[†2,b)]

Abstract: This paper proposes a method to exhaustively search for undiscovered generalized parallel counters (GPCs), aiming for the efficient implementation of multi-input adders on FPGAs. Multi-input adders, which are core components of multiplication and multiply-accumulate circuits, can be efficiently implemented on FPGAs by constructing trees of expanded full adders, known as GPCs. Although three GPC designs implementable with a single slice of the Xilinx 7 series are known, this paper aims to discover new GPCs by exhaustively enumerating those that can be realized within one slice. We enumerate all possible input-output specifications for GPCs with up to five output bits and search for the connections from the inputs to LUTs and their truth tables if they exist. A search program implemented in Rust completed enumeration in about 100 seconds and discovered five previously unknown GPCs: (1,2,6;4), (4,2,5;5), (1,2,4,4;5), (1,3,1,6;5), and (1,3,3,4;5). Using these GPCs to construct 8 to 32-bit multiplication circuits and multi-input adders resulted in a reduction of the number of slices in two circuits and the number of stages in two others.

1. はじめに

多入力の加算回路は, 古くから乗算や積和演算等の様々な算術演算回路の構成に用いられている [1] [2]. 特に近年ではニューラルネットワークのハードウェアアクセラレーションのコアとして重要度が高まっている.

多入力加算の効率的な実装法としては, 3 入力 2 出力の全加算器を基本素子として桁上げ保存加算器の木を構成する古典的な手法 [1] [2] や, 冗長 2 進加算器の木を構成する方法 [3] が知られており, n 個の n ビット 2 進数の加算が $O(\log n)$ の段数 (遅延時間) で実現できる.

これらの方法は, 回路の構成要素に論理ゲートもしくは全加算器を想定している. LUT (Look-up Table) 型の FPGA

†1 現在，関西学院大学大学院理工学研究科
†2 現在，関西学院大学工学部
a) mugi-noda@kwansei.ac.jp
b) ishiura@kwansei.ac.jp

ⓒ2024 Information Processing Society of Japan

での実装を考えた場合, 3 入力 2 出力の全加算器に基づく回路は必ずしも 5 ～ 7 入力の LUT を効率的に活用できるとは限らない.

このため, 多入力加算器の FPGA 実装には, 全加算器を 6 入力 3 出力に拡張したり, さらに入力に重み 1 だけでなく 2 の冪乗の重みを許した多入力の拡張加算器が用いられる. このような加算器を一般化並列カウンタ (Generalized Parallel Counter; GPC) と呼び, 多入力加算を行う GPC の木はコンプレッサツリーと呼ばれる [4] [6] [7] [9] [10]. コンプレッサツリーによる多入力加算も段数は $O(\log n)$ だが, FPGA に実装した場合には一般的に全加算器に基づくものよりも遅延や回路規模が小さくなる.

GPC には FPGA の内部構造に適合したものが種々考案されている. 特に加減算を効率化するための桁上げ先見回路 (キャリーロジック) を埋め込みで持つ FPGA では, これを活用することが重要になる. また, 数個の LUT とキャリーロジックをまとめた "スライス" や "ロジックブロック" を物理的な構成単位とする FPGA では, GPC をその単位で構成することが FPGA 全体の利用効率を高める.

Xilinx の 7 シリーズの FPGA に関しては, 1 スライスに実装可能な GPC として 3 種類が知られており [7] [10], そこから入出力を省略したバリエーションの 21 種類がコンプレッサツリーの基本素子として用いられている. しかし, これらはいずれも発見的に設計されたものであり, 1 スライスで実装可能な GPC がこれ以外に存在するかどうかは知られていなかった.

本稿では, 既知の GPC よりもさらに効率的な GPC の発見を目的に, 1 スライスで実装可能な GPC の網羅的探索を行う. 出力が 5 ビット以下の GPC の入出力仕様全てについて, 入力と LUT の接続と LUT の真理値表が存在するかどうかを判定し, 存在すればそれを求める.

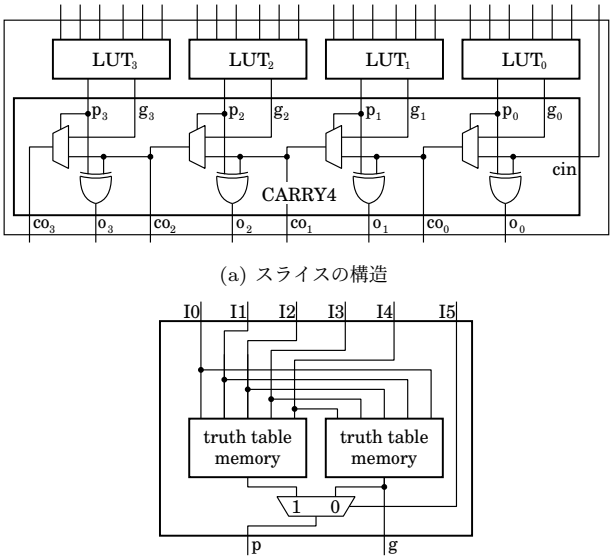
提案手法に基づき探索を行った結果, これまでに知られていなかった 1 スライスで実装可能な GPC を 5 種類発見することに成功した. これらの GPC を用いて 8 ～ 32 ビットの乗算回路と多入力加算回路のコンプレッサツリーを構成した結果, 発見した GPC が回路規模と遅延の削減に貢献することが確認できた.

2. 一般化並列カウンタの FPGA 実装

2.1 LUT 型 FPGA

LUT (Look-up Table) 型 FPGA (Field-Programmable Gate Array) は LUT とプログラム可能な配線によって論理回路を構成する. LUT はメモリに真理値表を記憶することにより, 任意の論理関数を実現する論理ゲートとして利用することができる. また, 加減算を効率よく実装するために専用のキャリーロジックが組み込まれているものもある.

本稿では, **図 1** のような FPGA のモデルを想定する. こ



(a) スライスの構造

(b) LUT の構造

図 1: 本稿で想定する FPGA のモデル

のような構成の FPGA としては Xilinx 7 シリーズが挙げられる [11]. 4 つの 6 入力 2 出力 LUT と 4 ビットのキャリーロジック (CARRY4) が "スライス" と呼ばれる 1 つのブロックを構成する. LUT の各出力は CARRY4 のキャリー生成信号 g_i と伝播信号 p_i に入力されている.

各 LUT は **図 1(b)** のように 2 つの真理値表を記憶したメモリとマルチプレクサにより構成される. それぞれの真理値表メモリは任意の 5 入力論理関数を実現する. ポート I5 に 1 を入力したときこの LUT は任意の 5 入力 2 出力論理関数を計算する. I5 を変数と見たときは, p 出力に任意の 6 入力関数を実現する. (g 出力には p の論理関数において I5 = 0 に固定したものが実現される.)

2.2 一般化並列カウンタ

多入力加算の効率的な回路構成法としては, 3 入力 2 出力の全加算器を基本素子とした Wallace Tree [1] や Dadda Tree [2] が知られている. しかし, FPGA への実装を考えた場合には, 全加算器の木は 5 ～ 6 入力の LUT にフィットせず, またキャリーロジックを有効活用できないため, これらは必ずしも効率的ではない.

そこで, 全加算器を 6 入力 3 出力に拡張した加算器や, 各入力に重み 1 だけでなく 2 の冪乗の重みを許す加算器を基本素子とすることが提案されている. このような拡張加算器は一般化並列カウンタ (Generalized Parallel Counter; GPC) と呼ばれる. また, GPC を用いた加算木はコンプレッサツリーと呼ばれる.

GPC の入出力仕様は $(p_{q-2}, p_{q-3}, ..., p_0; q)$ のように表される. この回路は $p_{q-2} + p_{q-3} + ... + p_0$ 個の入力と q 個の出力を持ち, 2^k の重みを持つ p_k 個のビットの重み和を q ビットで出力する. 例えば, 全加算器 (3;2) は重み 1 の入力を 3 つ持ち, その和を 2 ビットで出力する. また,

ⓒ2024 Information Processing Society of Japan

表 1: これまでに知られている 1 スライス実装可能な GPC

GPC 仕様	[ref]
(1,1,7;4)	[10]
(1,3,5;4)	[10]
(1,3,2,5;5)	[7]

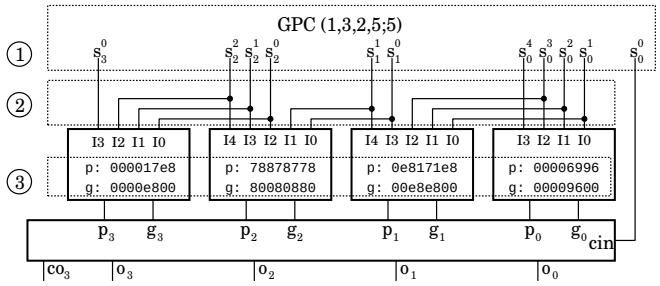


図 2: GPC (1,3,2,5;5) [10]

(1,3,5;4) は重み 1, 2, 4 の入力を, それぞれ 5, 3, 1 個持ち, その和を 4 ビットで出力する.

図 1 のモデルの FPGA において, キャリーチェーンを活用して 1 スライスで実装可能な GPC が求められており, これまでに **表 1** に示すものが知られている. これらの入出力の削減, またそれらの組み合わせによって得られる 21 種類の GPC がコンプレッサツリーの構成に用いられている [5] [6] [7] [8] [9] [10].

このような GPC を基本素子として多入力加算器を実現するコンプレッサツリーの構造は全加算器の木に比べて複雑になる. より小さい段数と, 小さい回路規模のコンプレッサツリーを構成するために整数線型計画法への定式化やヒューリスティックアルゴリズムが種々提案されている [4] [7] [9] [10].

3. GPC の列挙

3.1 GPC 列挙問題とアルゴリズムの概要

表 1 の GPC は発見的に知られているものであり, これが全てとは限らない. 本稿では, より効率的な GPC の発見を目的に, 前章の FPGA モデルにおいて 1 スライスで実装可能な全ての GPC を求める問題を扱う.

まず, 出力が 5 ビット以下の全ての GPC 仕様を生成する (**図 2** の①). 次に, そのそれぞれについて入力から LUT への可能な全ての接続を生成し (②), そのそれぞれについて所望の出力を与える LUT の真理値表が存在するかどうかを調べる (③). この際, GPC は分割が可能な場合があるため, その場合には分割して得られる全ての GPC の実現可能性を調べる.

3.2 GPC 仕様の列挙

本節では, GPC 仕様 $(p_{q-2}, p_{q-3}, ..., p_0; q)$ のうち, $q \leq 5$ であり, $2^{q-1} \leq \sum_{k=0}^{3} 2^k p_k < 2^q$ となるものを全て生成する.

k 桁目の入力の値を k 桁目の出力に反映させるためには, 図 2 のように k 桁目の入力は全て k 番目の LUT に入力する必要がある. 以下では右から k 番目の LUT を k 桁目の LUT と呼び, LUT_k と表記する. 本稿で対象とする LUT の入力数は最大 6 なので GPC の各桁の入力数 p_k は基本的に 6 以下を考えれば良い. しかし, 図 2 の s_0^0 入力のように, キャリーチェーンの cin 入力を用いれば最下位桁の入力は 1 つ増やすことができる. また, 次節で述べる GPC の分割を行うことにより最下位桁以外でも 7 入力まで扱えることがある. このため, 本稿では各桁 k について $0 \leq p_k \leq 7$ の範囲で GPC 仕様の生成を行う. 仕様の生成は $(0,0,0,0;0)$, $(0,0,0,1;1)$,...,$(0,0,0,7;3)$, $(0,0,1,0;2)$,...,$(3,1,1,1;5)$ のように辞書順で小さい順に行う.

この際, GPC 仕様の包含関係を利用して判定を省略し, 計算量の削減を図る. 例えば, $(6,0,7;5)$ が実現可能であれば $(5,0,6;5)$ は明らかに実現可能なので判定は行わない. また, $(2,7;4)$ が実現不可能ならば $(3,7;4)$ は明らかに実現不可能なので判定を行わない.

3.3 GPC の分割

GPC は分割可能な場合があり, これを利用すると以降の計算量を大幅に削減できる. また, 分割によって最下位桁以外でも cin を用いることができるようになり, p_k を 7 とすることが可能となる.

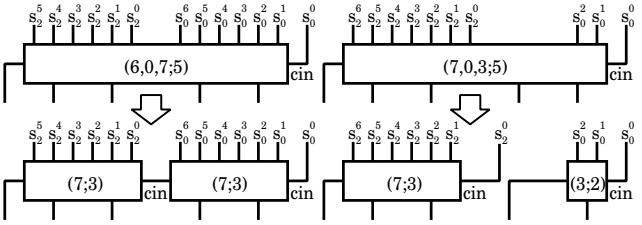
k 桁目から $k+1$ 桁目への桁上がりが最大で 1 のときには, その桁上がりはキャリーチェーンのみを使って伝播することができる. このような GPC は k 桁目以下と $k+1$ 以上の 2 つに分割することができる. 例えば, GPC $(6,0,7;5)$ は 1 桁目から 2 桁目への桁上がりの最大値が 1 なので, **図 3(a)** のように 2 つの (7;3) に分割できる. GPC $(6,0,7;5)$ は **図 3(c)** のように, キャリーチェーンを使って桁上がりを伝播することにより 1 スライスで実装できる.

また, k 桁目から $k+1$ 桁目への桁上がりが生じない場合には, 使われない cin 入力を活用した分割を行うことができる. 例えば, **図 3(b)** GPC $(7,0,3;5)$ は 1 桁目から 2 桁目への桁上がりが生じないので, 1 桁目と 2 桁目で分割を行うと, 2 桁目の 1 入力を使用されない cin から与えることができる. これにより 2 桁目の入力を最大 7 に増やすことができる. このように分割した GPC は, **図 3(d)** のように入力の 1 つを下位桁の LUT を経由して cin に入力することにより 1 スライスで実現できる.

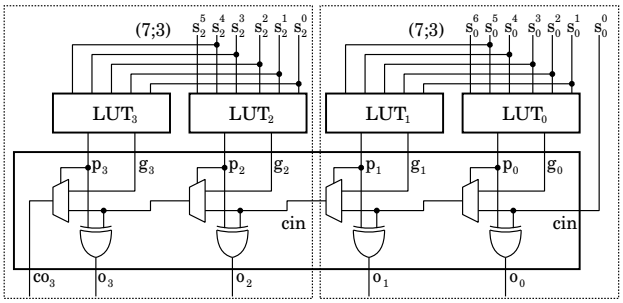
本手法では, 前節で生成した GPC 仕様が分割可能かどうかを判定し, 分割可能な場合には分割後の小さな GPC 仕様について次節以降の実装可能性判定を行う.

3.4 入力と LUT の接続の列挙

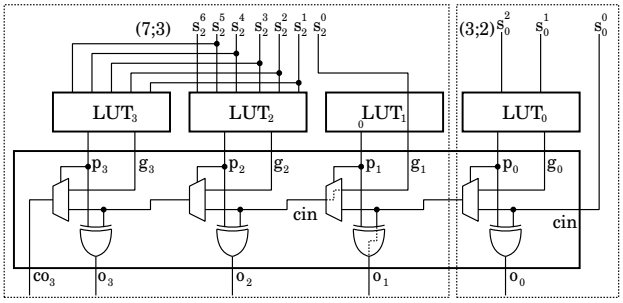
前節で得られた GPC 仕様に対し, 入力から LUT への



(a) (6,0,7;5) の分割　(b) (7,0,3;5) の分割



(c) (6,0,7;5) の実装



(d) (7,0,3;5) の実装

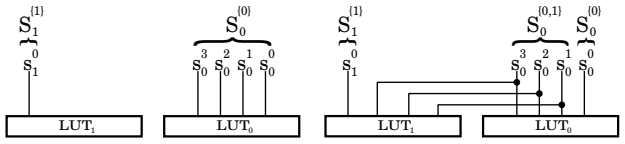
図 3: GPC の分割

可能な接続を全て生成する (図 2 の ②). この際, 本節ではまず LUT の全ての入力ポートが対称であるものとして扱い, 非対称なポートの扱いについては次節で述べる.

図 2 のようにキャリーチェーンの cin 入力を用いれば最下位桁の入力は必ず 1 つ増やせるので, GPC の実現可能性は最下位を 1 減らした GPC 仕様に対して行う. すると, 分割後の GPC 仕様 $(p_3, p_2, p_1, p_0; q)$ においては, 全ての桁 k で $p_k \leq 6$ であるものだけが実現可能であるため, そのような GPC 仕様に対してのみ処理を行う.

GPC $(p_3, p_2, p_1, p_0; q)$ の k 桁目の入力の集合を S_k とする. 図 2 の例では, $S_0 = \{s_0^4, s_0^3, s_0^2, s_0^1, s_0^0\}$, $S_1 = \{s_1^1, s_1^0\}$, $S_2 = \{s_2^2, s_2^1, s_2^0\}$, $S_3 = \{s_3^0\}$ である. k 桁目の LUT (LUT_k) に割り当てる入力の集合を L_k とする. 図 2 では $L_0 = \{s_0^4, s_0^3, s_0^2, s_0^1\}$, $L_1 = \{s_1^1, s_1^0, s_0^3, s_0^2, s_0^1\}$, $L_2 = \{s_2^2, s_2^1, s_2^0, s_1^1, s_1^0\}$, $L_3 = \{s_3^0, s_2^2, s_2^1, s_2^0\}$ である. 入力から LUT への接続の列挙は, 以下を満たす可能な組 (L_3, L_2, L_1, L_0) を列挙することである.

- $|L_k| \leq 6$. すなわち, LUT の入力数は最大 6 である.
- $S_k \subseteq L_k$. すなわち, GPC の k 桁目の入力は全て k 桁目の LUT に入力する.
- $L_k \subseteq S_k \cup S_{k-1} \cup ... \cup S_0$. すなわち, k 桁目の LUT には, GPC の k 桁目以下の入力からのみ入力される.



(a) 初期状態　(b) 割り当て後

図 4: 入力の割り当てによるグループの分割

組 (L_3, L_2, L_1, L_0) の生成は, L_k を $k = 0, 1, 2, ...$ の順に定めていくことにより行う.

$k = 0$ のとき, それより下の桁が存在しないので, $L_0 = S_0$ である.

$k > 0$ のとき, $L_k = S_k$ を初期値として, LUT_k の残りのポートに割り当てる入力の選択を全通り試行する. 残りのポート数は $6 - p_k$ なので, $0 \leq r \leq 6 - p_k$ であるような r 個の入力を $S_{k-1}, S_{k-2}, ..., S_0$ の中から選ぶ.

この際, 入力の対称性を利用して割り当ての組み合わせを減らす. S_k の元のうち, LUT の集合 T に入力するものの集合を S_k^T とする. 例えば, $S_0^{\{0,1,3\}}$ は S_0 のうち LUT_0, LUT_1, LUT_3 に入力するものを表す. S_k^T 中の入力は LUT の出力に関して対称である. したがって, S_k の入力を選ぶとき, S_k^T の要素は区別する必要がない.

入力の割り当ての例を **図 4** に示す. 初期状態 (**図 4(a)**) では, $S_0^{\{0\}} = S_0 = \{s_0^3, s_0^2, s_0^1, s_0^0\}$, $S_1^{\{1\}} = S_1 = \{s_1^0\}$ である. LUT_1 に S_0 から 3 本の入力を接続する場合, $S_0^{\{0\}}$ の中の要素は区別する必要はないので, 3 本の選択法は 1 通りである. この接続の結果, $S_0^{\{0\}}$ は $S_0^{\{0\}}$ と $S_0^{\{0,1\}}$ に分割される (**図 4(b)**).

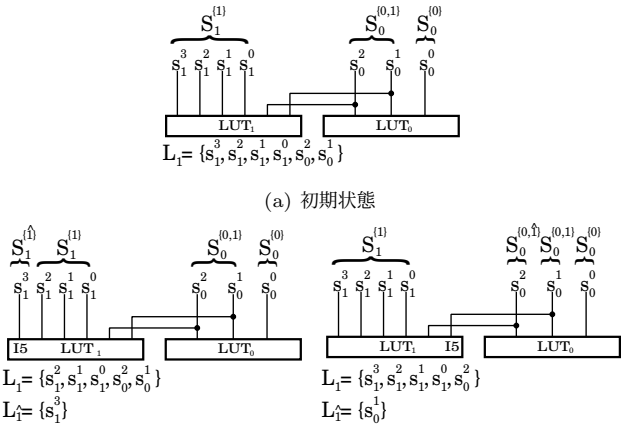
LUT_2 に接続する入力は $S_1^{\{1\}}, S_0^{\{0\}}, S_0^{\{0,1\}}$ の中から選択することになるが, この際に $S_0^{\{0,1\}}$ 内の入力は対称であるため区別する必要がない.

3.5 LUT が非対称な入力を持つ場合

本稿で想定する FPGA モデルでは, LUT は 図 1(b) のように, 5 入力か 6 入力を選ぶことができるが, 6 入力の場合はそのうちの 1 入力 (I5 ポート) が他とは対称ではない. 以下ではそのようなポートを「非対称ポート」と呼び, 他のポート (対称ポート) とは別に扱う. これは, k 桁目の LUT を, 対称ポートを入力とする LUT_k と, 非対称なポートを入力とする $\mathrm{LUT}_{\hat{k}}$ に (仮想的に) 分割することにより, 前節と同じ定式化で扱える.

LUT_k の対称なポートに割り当てる入力を L_k とし, 対称でないポートに割り当てる入力を $L_{\hat{k}}$ とする. また, $\hat{k}$ の記法は, S_k^T の定義にも適用するものとする. 例えば, $S_0^{\{0,1,\hat{2}\}}$ は, S_0 のうち LUT_0 と LUT_1 の対称ポートと LUT_2 の非対称ポートに割り当てられた入力を表す.

L_k と $L_{\hat{k}}$ の選択は, まず k 桁目の LUT の入力が全て対称であるとして L_k の初期値を求めた後, L_k の中から $L_{\hat{k}}$



(a) 初期状態

(b) 割り当て後 1

(c) 割り当て後 2

図 5: 非対称ポート割り当てによる入力グループの分割

に移動させるものを 1 つ選ぶことにより行う.

図 5 に, 非対称ポートを含む LUT_1 に入力を割り当てる例を示す. まず, **図 5(a)** では前節の手法で 6 つの入力が LUT_1 に割り当てられており, 入力のグループは $S_1^{\{1\}} = \{s_1^3, s_1^2, s_1^1, s_1^0\}, S_0^{\{0,1\}} = \{s_0^2, s_0^1\}, S_0^{\{0\}} = \{s_0^0\}$ である. ここで, LUT_1 に割り当てられている入力のグループは, $S_1^{\{1\}}, S_0^{\{0,1\}}$ の 2 つなので, 非対称ポートの割り当て方はこれらのどちらから選ぶかの 2 通りである. $S_1^{\{1\}}$ から選んだ場合の状態を **図 5(b)** に示す. 非対称なポート (I5) に割り当てた入力は他と区別する必要があるので, $S_1^{\{1\}}$ を $S_1^{\{\hat{1}\}} = \{s_1^3\}, S_1^{\{1\}} = \{s_1^2, s_1^1, s_1^0\}$ に分割する. 同様に, $S_0^{\{0,1\}}$ から選んだ場合の状態 (**図 5(c)**) では, $S_0^{\{0,1\}}$ を $S_0^{\{0,\hat{1}\}} = \{s_0^2\}, S_0^{\{0,1\}} = \{s_0^1\}$ に分割する.

3.6 LUT の真理値表の存在判定

与えられた GPC の入出力仕様 $(p_3, p_2, p_1, p_0; q)$ 及び入力の LUT への接続 $(L_3, L_{\hat{3}}, L_2, L_{\hat{2}}, L_1, L_{\hat{1}}, L_0, L_{\hat{0}})$ に対し, そのような GPC を実現する LUT の真理値表が存在するかを判定する (図 2 の ③).

これは, GPC に対する全ての入力値について, 所望の出力値が得られるように各 LUT_k の出力値, すなわち $(p_3, g_3, p_2, g_2, p_1, g_1, p_0, g_0)$ の値を決定していくことにより行う. この際, GPC のある出力値を得るための $(p_3, g_3, p_2, g_2, p_1, g_1, p_0, g_0)$ の値の組み合わせは複数存在すること, および, GPC への入力値に対してある LUT_k の出力に矛盾が生じ得ることに留意する必要がある.

同じ GPC 出力を得るための p_k, g_k の値の組み合わせが複数存在するのは, キャリーロジックの don't care と, キャリーを複数桁にまたがって扱えることに起因する. キャリーロジックへの入力 $(p_k, g_k) = (1, 1)$ に対する出力は don't care だが, 本稿で想定する FPGA のキャリーチェーン (**図 1(a)**) の実装では, **表 2** に示す真理値表のように, $(p_k, g_k) = (1, 1)$ に対する出力は $(p_k, g_k) = (1, 0)$ と同じになっている. したがって, $(p_k, g_k) = (1, 0)$ とすべき場合に $(p_k, g_k) = (1, 1)$ としても同じ出力を得ることができる. また, $cin_k = co_{k-1}$ であることから $k-1$ 桁目からのキャリーを p_{k-1}, g_{k-1} で決定することにより cin_k, ひいては co_k, o_k の値を決定することも可能である.

表 2: キャリーチェーン 1 桁分の真理値表

cin_k	p_k	g_k	co_k	o_k
0	0	0	0	0
0	0	1	1	0
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	1	1
1	1	0	1	0
1	1	1	1	0

例えば, **図 6** において, ① のように $(0, 0, 0, 1, 1)$ を入力したとき, $(0, 1, 0)$ を出力させるためには, (p_1, g_1, p_0, g_0) は $(0, 0, 0, 1), (1, 0, 0, 0), (1, 1, 0, 0)$ の 3 通りが可能である. $(1, 0, 0, 0), (1, 1, 0, 0)$ はキャリーチェーンにおいて, $(p_1, g_1) = (1, 0)$ の代わりに $(1, 1)$ とできることによる. また, $(0, 0, 0, 1)$ は 0 桁目からのキャリー信号 co_0 によって $o_1 = 1$ を実現する.

これらのキャリーチェーンの入力値 (LUT の出力値) の候補は, 与えられた LUT の接続の制約によって実現できない場合があるため, そのようなものを削除する. 図 6 において, ② は入力に $(0, 0, 0, 1, 0)$ を与えた場合を示している. 枠で示した LUT_1 への入力は ① ② いずれの場合も $(0, 0, 0, 1)$ なので, LUT_1 は両入力について同じ値を出力する必要がある. このため, 出力 $(1, 0), (1, 1)$ は候補から削除され, この入力に対する出力は $(0, 0)$ だけが残る.

LUT の非対称な入力を利用している場合には, LUT の真理値表に制約が生じる. **表 3** は 図 1(b) の LUT の真理値表である. 真理値表メモリはそれぞれ $T_0 \sim T_{31}, T_{32} \sim T_{63}$ の 32 ビットずつを記憶し, 入力 I0 ∼ I5 によって対応する値を p, g に出力する. $\mathrm{I5} = 0$ のとき, p 出力は g 出力と同じ真理値表メモリ $(T_0 \sim T_{31})$ から値を出力するため $p = q$ でなければならない. この制約に基づいて候補を削除する.

これらの操作の後, GPC の全ての入力値に対して 1 つ以上の LUT の出力値の候補が存在するならば, 真理値表を定めることができる. 逆に, 1 つ以上の GPC の入力値で, 候補の集合が空になるならば, その LUT の接続で GPC は実現できない.

4. 実験結果

提案手法に基づく GPC 探索プログラムを Rust で実装した [*1]. Ryzen 7 PRO 6850U のシングルスレッドで実行した結果, 約 100 秒で全ての実行が完了した.

[*1] 実装: https://github.com/ishiuralab/advgpcgen-rs

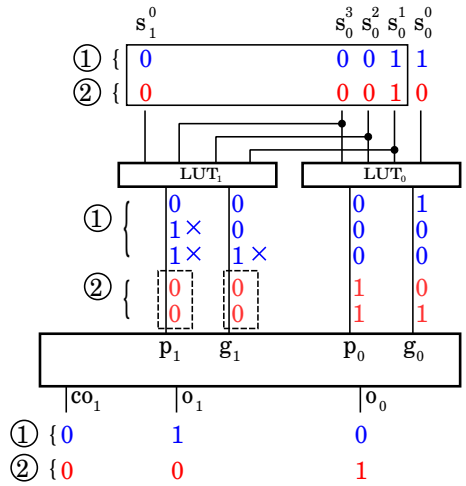


図 6: LUT の出力値の候補とその削除

表 3: 図 1(b) の LUT の真理値表

I5	I4	I3	I2	I1	I0	p	g
0	0	0	0	0	0	T_0	T_0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
0	1	1	1	1	1	T_{31}	T_{31}
1	0	0	0	0	0	T_{32}	T_0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
1	1	1	1	1	1	T_{63}	T_{31}

表 4: 使用した GPC の集合

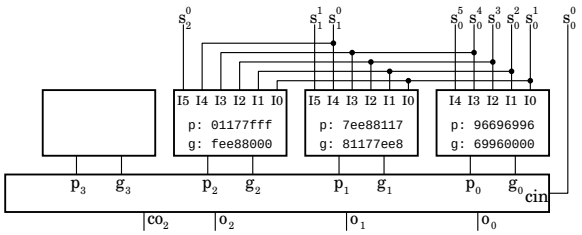
(1;1)	(1,3,5;4)	(1,3,4,3;5)	(2,1,3,5;5)	(1,4,0,6;5)
(1,4,1,5;5)	(1,4,2,3;5)	(1,5;3)	(2,1,5;4)	(1,1,7;4)
(2,1,1,6;5)	(1,1,6,3;5)	(3;2)	(2,3;3)	(2,2,3;4)
(2,2,2,3;5)	(7;3)	(2,0,7;4)	(6,0,6;5)	(6,1,5;5)
(6,2,3;5)	(1,3,2,5;5)			
(7,0,3;5)	(1,2,6;5)	(2,1,2,6;5)	(1,2,5,3;5)	(5,2,4;5)
(4,4;4)	(1,2,4,4;5)	(1,3,1,6;5)	(1,3,3,4;5)	

探索の結果, Xilinx 7 シリーズの 1 スライスで実装可能な GPC として新たに (1,2,6;4), (4,2,5;5), (1,2,4,4;5), (1,3,1,6;5), (1,3,3,4;5) の 5 つを発見した. 発見した GPC の実装を **図 7** に示す.

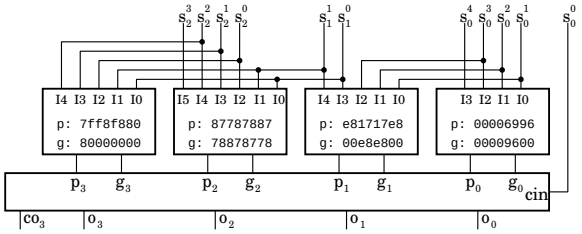
新たに発見した GPC を用いてコンプレッサツリーを構成する実験を行った. 使用した GPC を **表 4** に示す. 上側が既存の GPC で, 下側が本稿で発見した GPC とそれらから入出力を削除して生成されるバリエーションである.

[9] の定式化を用い, 8 ～ 32 ビットの乗算器と正方形を 2 つの値になるまで圧縮するコンプレッサツリーを構成した. 実行には Ryzen 9 3900X 上で, IBM ILOG CPLEX Optimization Studio 22.1.0 を用い, 制限時間は 7200 秒とした. また, 最小段数を求める際には目的関数を設定せず, 最小化の目的関数はスライス数最小化とした.

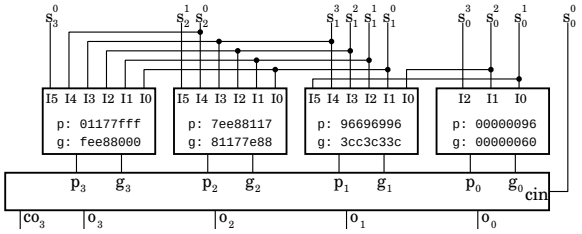
表 5 に結果を示す. 左が乗算器, 右が正方形である. 新たな GPC を用いることにより, 13 ビットの乗算器と 11 ビットの正方形の実装に必要なスライス数を削減できた. これらはいずれも最適解である. また, 26 ビットの乗算器と 23 ビットの乗算器では, 段数を削減することができた.



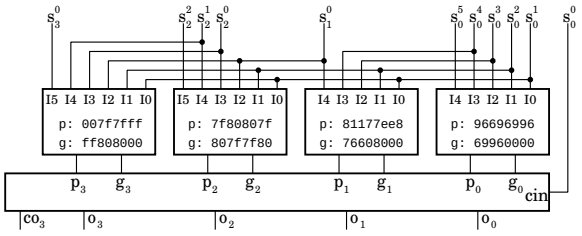
(a) GPC (1,2,6;4)



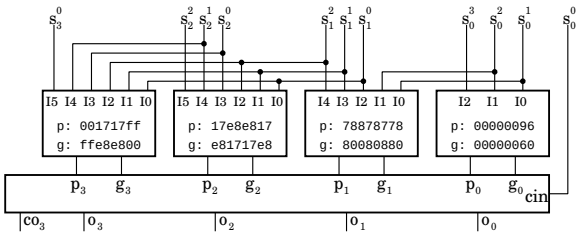
(b) GPC (4,2,5;4)



(c) GPC (1,2,4,4;5)



(d) GPC (1,3,1,6;5)



(e) GPC (1,3,3,4;5)

図 7: 本手法により発見した GPC

表 5: 多入力加算器の合成結果 (”✓” は最適解)

	n ビット乗算器				n ビット n 入力加算器			
	従来法		本手法		従来法		本手法	
n	stage	slice	stage	slice	stage	slice	stage	slice
11	✓2	✓14	✓2	✓14	✓3	✓15	✓3	✓**14**
13	✓3	✓20	✓3	✓**19**	✓3	✓21	✓3	✓21
23	✓3	72	✓3	71	4	71	✓**3**	75
26	4	91	✓**3**	94	4	94	✓4	93

ⓒ2024 Information Processing Society of Japan

5. 結論

本稿では, FPGA における GPC を列挙する方法を提案した. 手法に基づき Xilinx 7 シリーズにおいて 1 スライスに実装できる GPC を探索したところ, 新たに 5 種類の GPC を発見した. 実際に多入力加算回路を構成した結果, これらの GPC は回路規模と遅延の削減に貢献することが確認できた.

今後の課題として, Xilinx 7 シリーズ以外の FPGA モデルに本手法を拡張することが挙げられる.

謝辞 本研究に関して有益な御助言を頂いた京都高度技術研究所の神原弘之氏, 立命館大学の富山宏之教授, 元立命館大学の中谷嵩之氏に感謝いたします. また, 本研究に関してご協力, ご討議頂いた関西学院大学の石浦研究室の諸氏に感謝いたします.

参考文献

[1] S. C. Wallace: “A Suggestion for a Fast Multiplier,” in IEEE Trans. on Electronic Computers, vol. EC-13, no. 1, pp. 14–17 (Feb. 1964).
[2] L. Dadda: “Some Schemes for Parallel Multipliers,” in Alta Frequenza, vol. 34, pp. 349–356 (May 1965).
[3] N. Takagi, H. Yasuura, and S. Yajima: “High-Speed VLSI Multiplication Algorithm with a Redundant Binary Addition Tree,” in IEEE Trans. on Computers, vol. C-34, no. 9, pp. 789–796 (Sept. 1985).
[4] T. Matsunaga, S. Kimura, and Y. Matsunaga: “Multi-Operand Adder Synthesis on FPGAs Using Generalized Parallel Counters,” in Proc. Asia and South Pacific Design Automation Conference (ASP-DAC 2010), pp. 337-342 (Feb. 2010).
[5] B. Khurshid and R. N. Mir: “High Efficiency Generalized Parallel Counters for Xilinx FPGAs,” in Proc. International Conference on High Performance Computing (HiPC 2015), pp. 40–46 (Dec. 2015).
[6] M. Kumm and P. Zipf: “Efficient High Speed Compression Trees on Xilinx FPGAs,” in Proc. Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV 2014), pp. 1–12 (Jan. 2014).
[7] M. Kumm and P. Zipf: “Pipelined Compressor Tree Optimization using Integer Linear Programming,” in Proc. International Conference on Field Programmable Logic and Applications (FPL 2014), pp. 1–8 (Sept. 2014).
[8] T. B. Preußer: “Generic and Universal Parallel Matrix Summation with a Flexible Compression Goal for Xilinx FPGAs,” in Proc. International Conference on Field Programmable Logic and Application (FPL 2017), pp. 1–7 (Sept. 2017).
[9] M. Kumm and J. Kappauf: “Advanced Compressor Tree Synthesis for FPGAs,” in IEEE Trans. on Computers, vol. 67, no. 8, pp. 1078–1091 (Jan. 2018).
[10] Y. Yuan, L. Tu, K. Huang, X. Zhang, T. Zhang, D. Chen, and Z. Wang: “Area Optimized Synthesis of Compressor Trees on Xilinx FPGAs Using Generalized Parallel Counters,” IEEE Access, vol. 7, pp. 134815–134827 (Sept. 2019).
[11] Xilinx, Inc.: 7 Series FPGAs Configurable Logic Block User Guide (UG474) (Sept. 2016), https://docs.amd.com/v/u/en-US/ug474_7Series_CLB (accessed in June 2024).

ⓒ2024 Information Processing Society of Japan