高速化指向の関数分解と新しい自由度を用いた論理回路の簡単化

山下 茂, 澤田 宏，名古屋 彰
NTT コミュニケーション科学研究所
〒 619-02 京都府相楽郡精華町光台 2 丁目
Tel: 0774-95-1867
Fax: 0774-95-1876
E-mail: {ger, sawada, nagoya}@cslab.kecl.ntt.co.jp

あらまし

本稿では，テーブル参照 (LUT: Look-Up Table) 型 FPGA (Field Programmable Gate Array) をターゲットとする論理合成手法のための 2 つの要素技術について紹介する．また，その要素技術をどのように利用するかについても言及する．

1 つ目は，特に最終的な段数を減らすことを目的とした関数分解手法である．この手法は，与えられた関数 f の $f = \alpha(g_1(X^1), g_2(X^2))$ という分解の中で，変数集合 X^1 と X^2 に含まれる変数の数が最小である分解を効率的に見つけるものである．この手法を再帰的にまたは他の分解手法と組み合わせて使うことにより，LUT 型 FPGA 向けの回路が合成可能である．

2 つ目は論理関数の自由度の新しい表現方法およびそれを用いた論理回路の簡単化手法である．この手法は，**SPFD (Sets of Pairs of Functions to be Distinguished)** と呼ばれる「関数の対の集合」を用いて回路内の中間点での論理関数の自由度を表現する．この手法を用いて，冗長度を含んだ LUT の初期回路を簡単化することが可能である．

Peformance-Driven Functional Decomposition and Circuit Minimization Using New Functional Permissibilities

Shigeru Yamashita, Hiroshi Sawada and Akira Nagoya
NTT Communication Science Laboratories
2 Hikaridai, Seika-cho, Soraku-gun, Kyoto 619-02, JAPAN
Tel: 0774-95-1867
Fax: 0774-95-1876
E-mail: {ger, sawada, nagoya}@cslab.kecl.ntt.co.jp

Abstract

This paper presents two basic techniques for LUT (Look-Up Table) based FPGA (Field Programmable Gate Array) logic synthesis. How to utilize them is also presented.

One of them is a functional decomposition method which aims to produce logic circuits with few levels. The method efficiently finds the decomposition form: $f = \alpha(g_1(X^1), g_2(X^2))$ where the total number of variables in X^1 and X^2 is the smallest for given f. Logic circuits for LUT based FPGAs can be generated by using this method recursively or using this method together with other decomposition methods.

The other technique is a new method to expresses functional permissibilities and its application to circuit minimization. This method express permissibilities of internal functions in a circuit by using sets of pairs of functions called **SPFD (Sets of Pairs of Functions to be Distinguished)**. LUT circuits with some logical redundancies can be minimized by the method.

1 はじめに

近年 FPGA (Field Programmable Gate Array) は，従来の ASIC では考えられなかった柔軟性をもつハードウエアとして注目を集めるようになっている．種々ある FPGA の一つに，テーブル参照 (LUT: Look-Up Table) 型と呼ばれる FPGA がある．それは，プログラム可能な論理ブロックの配列と，それら論理ブロックをつなぐプログラム可能な配線領域から成る．論理ブロックには，LUT とフリップフロップが含まれている．LUT は k (k は 4 か 5 であることが多い) 入力のブール関数を実現できる．我々は LUT 型 FPGA に特化した論理設計手法を開発し，既に発表している論理合成ツール PARTHENON[1] に組み入れることを検討している．

LUT はある一定数 k 入力以下の任意の関数が実現できる万能セルであるので，LUT 型 FPGA 向けの論理設計では，まず与えられた論理を全て k 入力以下の関数の多段接続 (このような回路を以下では LUT 回路と呼ぶ) に変換しなければならない．この作業をテクノロジーマッピングと呼ぶ．我々は既に，関数分解による LUT 向けのテクノロジーマッピングの手法を開発している [2]．この手法で用いられている関数分解手法は，与えられた関数 f を
$f(X) = \alpha(g_1(X^B), \cdots, g_t(X^B), X^F)$ と分解する手法を基にしている．本稿では，この分解手法と違い，与えられた関数 f を $f(X) = \alpha(g_1(X^1), g_2(X^2))$ と分解する手法について述べる．この分解の形は bi-decomposition と呼ばれているが，特に X^1 と X^2 の変数集合に重なりがない分解 (disjoint bi-decomposition と呼ばれる) に関して，近年盛んに研究が行われている [3, 4]．本稿では，X^1 と X^2 の変数集合に重なりがあるような場合も含めて，X^1 と X^2 に含まれる変数の数が最小となるような分解 (以下最適な分解と呼ぶ) を見つける手法について述べる．この手法は，$g_1(X^1)$ と $g_2(X^2)$ の規模を両方ともできるだけ小さくする事によって与えられた関数を 2 入力関数のツリー状に分解し，最終的に段数の少ない LUT 回路を生成することを目標としている．そのため，高速化指向の関数分解と呼んでいる．

一方，一旦生成された LUT 回路に冗長度が含まれることがあるため，それを何らかの手法で簡単化することも必要である．従来，この簡単化の際には回路内の冗長度を不完全指定論理関数を用いて表現していた [5]．LUT 回路の簡単化にも不完全指定論理関数を利用することは可能であるが，それでは LUT が万能セルであることを利用できない．そこで，我々は LUT が万能セルであることを利用可能な論理関数の自由度の表現方法を考案した [6]．この表現方法は，**SPFD (Sets of Pairs of Functions to be Distinguished)** と呼ばれる「関数の対の集合」を利用することを特徴としている．

本稿は以下のように構成されている．2 章では，高速化指向の関数分解の 1 手法である最適な bi-decomposition を求める手法の概要を述べる．3 章では，SPFD の概要と SPFD を用いた回路簡単化手法について述べる．4 章では，2, 3 章の手法をどのように統合する予定であるかを述べる．最後に 5 章で本稿をしめくくる．

2 高速化指向の関数分解

本章では，段数の少ない LUT 回路生成するための関数分解の 1 手法である，最適な bi-decomposition を見つける手法について述べる．bi-decomposition とは，関数 f に対して $f = \alpha(g_1(X^1), g_2(X^2))$ という分解のことである．この分解は，X^1 と X^2 の変数集合に重なりがない時，disjoint bi-decomposition と呼ばれており，効率的な検出方法も提案されている [3, 4]．disjoint bi-decomposition は比較的簡単に見つけられるため関数分解の前処理などに非常に有効であると考えられている．X^1 と X^2 の変数集合に重なりを許すより一般的な bi-decomposition は，より多くの分解を扱えるが，

disjoint bi-decomposition に比べて検出するのが難しい．しかし，我々は計算時間を多少要してもより多くの分解を扱うために， disjoint ではない分解も考慮に入れて最適な bi-decomposition を見つける手法を考案した [7]．以下ではその手法を簡単に紹介する．

2.1 bi-decomposition による LUT 回路生成

本節では bi-decomposition を用いた LUT 回路生成の簡単な例を示す．今， $f = (x_1 + x_2 + x_3) \cdot (x_2 \oplus x_3 \oplus x_4) \cdot (x_1 \oplus x_3 \oplus x_5)$ という関数を 3 入力 LUT にマッピングすることを考える． f が上記のように表現できることがわかっていれば，図 1のように論理を分解して， 4 つの LUT で実現可能であることがわかる．

通常論理合成の途中では，実現したい関数のこのような分解された形がわかっていないため，まず関数の分解を行わなければならない．この例では， bi-decomposition による分解を用いれば，図 1のような分解が実現できる．具体的には，最初に f に対して $f = g_1 \cdot g_2$, $g_1 = (x_1 + x_2 + x_3) \cdot (x_2 \oplus x_3 \oplus x_4)$, $g_2 = x_1 \oplus x_3 \oplus x_5$ という分解を見つけ，その次に g_1 および g_2 の分解を再帰的に見つければよい．このような分解は， disjoint bi-decomposition 等では見つけることはできないが，本稿で述べる手法では見つけることができる．

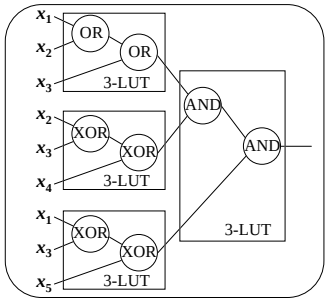

図 1: bi-decomposition による LUT 回路の生成

2.2 最適な bi-decomposition の発見手法

$f(X) = \alpha(g_1(X^1), g_2(X^2))$ という分解では， f, g_1, g_2 の否定を自由に取ることを許せば， α の論理が AND の場合と XOR の場合のみを考慮すれば良いことになる．以下では，最適な XOR による分解を求める手法を例を用いて簡単に説明するが， AND による分解もほぼ同様である．(詳細は [7] を参照して頂きたい．)

以下では説明上，ある関数 f の f' への変形は， f と f' を表す真理値表の違っている部分に注目して，「f を表す真理値表のある部分の値 (0, 1, $*$ (ドントケア)) を別のある値に変更する．」と表現することにする．そのため，真理値表の変化させたい部分を表すために以下のような表現を使う．図 2において，

- (A) の部分を「f の x_3=0 の部分」と呼ぶ．
- (B) の部分を「f の x_3=1 の部分」と呼ぶ．
- 斜線部分を「f の x_3 非対称な部分」と呼ぶ．これは， $f_{|x_3=1} \oplus f_{|x_3=0}$ で計算できる．なお対称を考える時 $*$ は無視している．

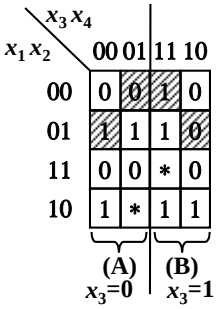

図 2: 関数の真理値表

今， ON-set が $(x_2 + \overline{x}_4) \oplus (x_1 \cdot x_4 + \overline{x}_1 \cdot \overline{x}_3)$ で， DC-set が $x_3 \cdot (x_2 \cdot \overline{x}_4 + \overline{x}_1 \cdot \overline{x}_2 \cdot x_4)$ であるような不完全指定論理関数 f の最適な XOR 分解を考える． f の真理値表を図 3の左上に示す．また， f の最適な XOR 分解の生成手順の探索木の一部を図 3に示す．

最初に，関数 f から暫定的な解となる初期解 g_1 および g_2 を以下の規則に従って生成する．

- $f = 1$ ならば $g_1 = 1, g_2 = 0$ となる．

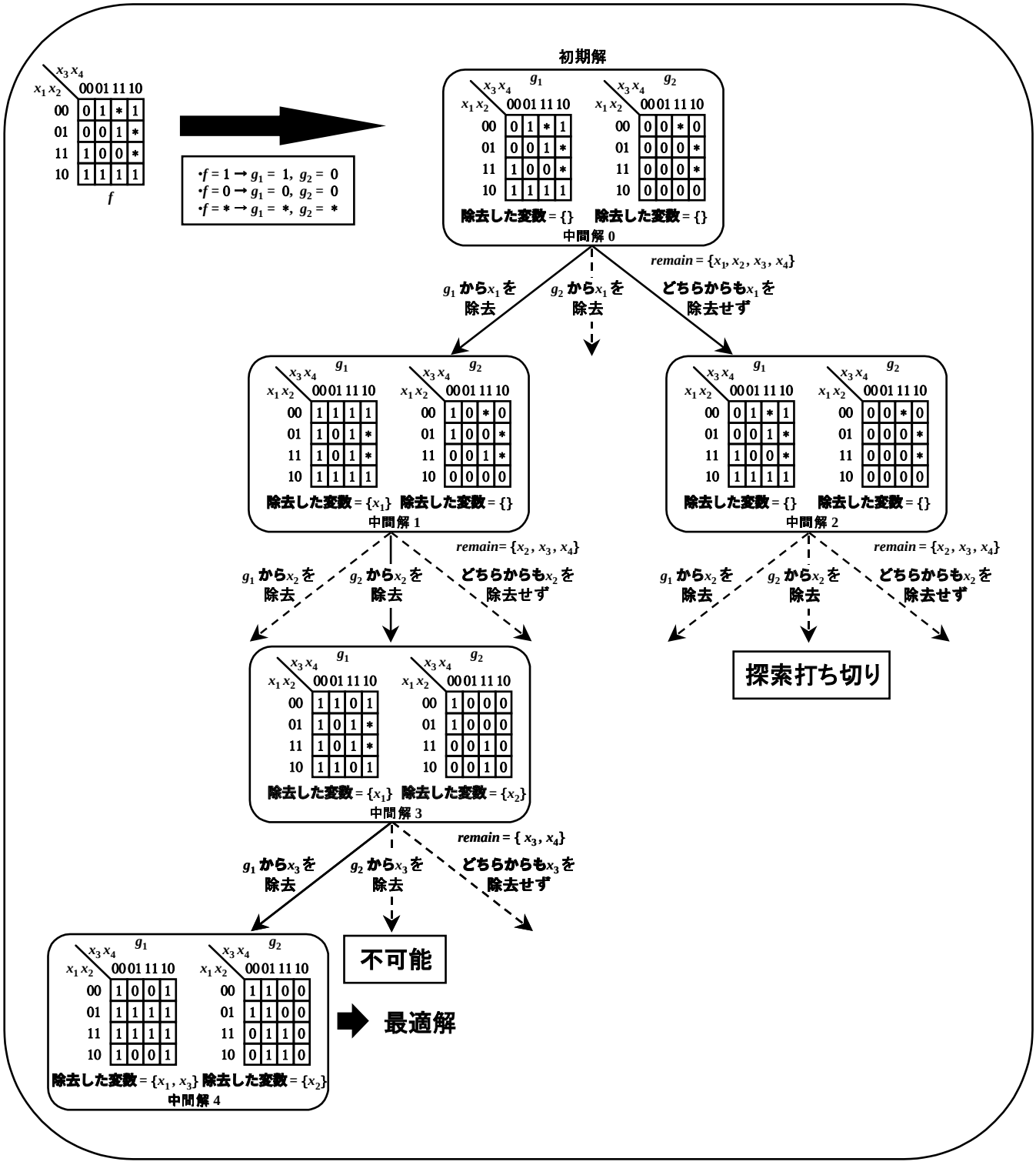

図 3: 最適な XOR 分解の探索

- $f=0$ ならば $g_1=0, g_2=0$ となる．
- $f=*$ ならば $g_1=*, g_2=*$ となる．

これは，図中の中間解０に相当する．この中間解０は明らかに $f = g_1 \oplus g_2$ を満たしている．本稿で述べる手法は，この中間解０から改善した中間解を次々と作り，分枝限定法に基づいて最適な分解を見つける．その基本的な方針は次の通りである．最適な分解を表す g_1, g_2 はその依存する変数の数が最小であるのだから，中間解 0 の g_1 および g_2 が依存する変数の数を可能な限り減らして行く．しかし，一度に複数の変数の依存度を除去するのは，選択すべき変数集合の組み合わせが非常に膨大となるため非現実的である．そこで，一つずつ変数を除去していくだけで全ての場合を調べれるようにする．１つの変数の依存度を除去するのが分枝限定法における 1 ステップとなる．

この方針に従った分枝限定法での解の改善は，以下の 3 通り (図 3参照) となる．

分枝１ 中間解の g_1 から x_i の依存度を可能であれば除去して，探索を続ける．

分枝２ 中間解の g_2 から x_i の依存度を可能であれば除去して，探索を続ける．

分枝３ 中間解の g_1, g_2 のどちらからも x_i の依存度を除去せずに，探索を続ける．

x_i の依存度は，良くても g_1 または g_2 の一方からしか除去できないため，分枝１または分枝２を考慮する．また， x_i の依存度を g_1 (または g_2) から除去したために，その後最適な解を見つけられないということが起こり得るので，分枝３も必要である．例えば，中間解０において g_1 から x_1 の依存度の除去を行えば中間解１が， g_1 および g_2 のどちらからも x_1 の依存度の除去を行わないと決定すれば中間解２が得られる．図中の $remain$ は，その中間解以降に除去される可能性がある変数の集合を表す．図のように次々と中間解を改善しながら最適な解を見つける．この際に，ある中間解からの探索をある条件で打ち切ることによって処理の高速化を図れる．この例では $g_1 = x_2 + \overline{x}_4$, $g_2 = x_1 \cdot x_4 + \overline{x}_1 \cdot \overline{x}_3$ (中間解４) が最終的に最適な解となるが，もし中間解４が中間解２より先に見つかって暫定最適解となっていれば，中間解２以降の探索は打ち切られる．その理由は以下の通りである．中間解２の (g_1 の依存変数の数 (4)+ g_2 の依存変数の数 (4) - $remain$ に含まれる変数の数 (3)) が 5 であるので，中間解２以降の探索で得られる解は，最も良くても g_1 と g_2 の依存する変数の合計が 5 にしかならない．しかし，暫定最適解の (g_1 の依存変数の数 (2) + g_2 の依存変数の数 (3)) が 5 であるので，中間解２以降の探索は無駄だとわかるからである．

上述した分枝限定法による探索を可能とするには，解の改善を行って分枝する際に以下の条件を満たすようにしなければならない．

条件１ 改善後の解でも $f = g_1 \oplus g_2$ が成り立つこと．

条件２ g_1 および g_2 は， 改善前に依存度が除去されていた変数に再び依存しないこと．

条件１を満たすためには， g_1, g_2 の真理値表上で同じ部分の $0, 1$ の反転を行う操作しか許されない．また，条件２を満たすためには，反転する部分は改善前に g_1, g_2 の依存度から除去された変数全てに依存していてはならない．つまり，反転する部分は改善前に g_1, g_2 の依存度から除去された変数全てに関して対称でなければならない．

以下では，図 3の中間解１の g_2 から x_2 の依存度を除去する例を用いて説明する．処理の大まかな流れを図 5に示す．

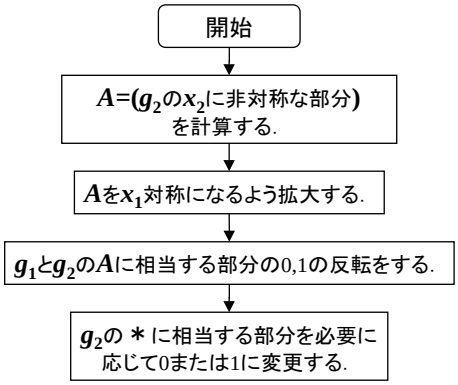

図 5: 変数除去の操作の流れ図

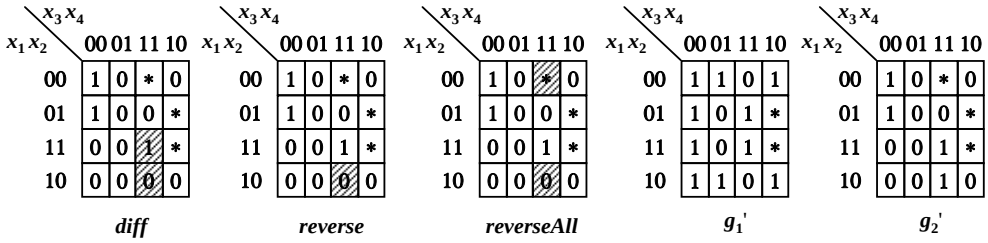

図 4: 変数除去の操作で現れる関数

まず，g_2 の真理値表で x_2 非対称な部分を計算する．これに該当するのは，図 4の *diff*の斜線部分である．この部分のために，g_2 が x_2 に依存していることになる．この部分に相当するうち，$x_2 = 0$，または $x_2 = 1$ のいずれかの部分の $0,1$ の反転を行えば，g_2 の真理値表は x_2 対称となる．つまり，g_2 から x_2 の依存度が除去できる．ここでは，*diff*の x_2=0 の部分の反転を行うことにする．これに該当するのは，図 4の $reverse$ の斜線部分である．

しかし，単に g_1 と g_2 の $reverse$ の部分のみを反転させただけでは，g_1 の依存度から既に除去したはずの x_1 に，g_1 が再び依存するようになってしまう．そのため，反転する部分は x_1 に依存しないようにしなければならない．以上をまとめると，反転する部分は以下のようになっていなければならない．

- 少なくとも $reverse$ を含んでいること．
- x_1 対称であること．

この条件を満たす最小の部分は，$reverse_{|x_1=1} + reverse_{|x_1=0}$ で計算できる．直感的には，この演算は $reverse$ が x_1 対称となるように必要最低限の拡大を行うことに相当する．この演算結果に該当するのは，図 4の $reverseAll$ の斜線部分である．

結局，g_1 および g_2 の $reverseAll$ の部分の 0,1 を反転すれば良いことになる．g_1, g_2 の該当部分を反転した結果の真理値表を図 4の g_1', g_2' に示す．ここまでの操作は $*$ の部分を考慮に入れていなかったので，g_2 が x_2 に依存しないように，g_2' の $*$ の必要な部分を 0 または 1 に変更する．以上の操作で図 3の中間解 3 を得ることができる．

もちろん全ての場合に g_1（または g_2）の依存度からある変数 x_i を除去できるとは限らない．(例えば，図 3の中間解 3 の g_2 から x_3 の依存度を除去することは不可能である．そのような場合には，矛盾無く上記のような操作ができない．詳細は文献 [7] を参照して頂きたい．) そのような場合には，分枝限定法においてそれ以降の探索は打ち切られることになる．

上述した変数除去の方法によれば，x_i, x_j の順なら変数の依存度の除去が可能だが，x_j, x_i の順なら変数の依存度を除去が不可能であるということは起こりえない．そのため，どのような変数除去の順番でも最適な解を見つけることができる．

2.3 実験結果および考察

我々は，最適な bi-decomposition を求める手法の有効性を調べるために，MCNC ベンチマーク回路 [8] に対して簡単な実験を行った．具体的には，与えられた関数に対して，AND による bi-decomposition，XOR による bi-decompositon および単なる shanon 展開の中で，最適なものを選ぶという関数分解手法を再帰的に用いて，回路を 2 入力のノードのみに分解する実験を行った．この結果を表 1に示す．表中の node，lev が合成回路のノード数および段数を示す．比較として，文献 [3] の結果 (段数は示されていない) と UCB の SIS で文献 [3] と同じスクリプトを用いた合成結果を示す．我々の実現した手法による結果は提案手法の欄で，CPU は Sun Ultra 2 2200 上で測定した計算時間を示す．

表 1: 2 入力ノードへの分解結果

回路名	[3]	SIS	提案手法
	node	node/lev	node/lev/CPU
5xp1	73	100/22	77/7/0.28
9symml	199	243/14	202/10/1.04
con1	15	17/4	16/4/0.03
duke2	550	641/26	759/11/30.80
e64	390	877/11	655/7/5.53
f51m	60	99/26	65/6/0.12
misex1	56	42/12	74/6/0.11
misex2	109	127/8	158/5/0.16
misex3c	757	457/45	820/18/78.36
rd53	21	32/9	30/5/0.05
rd73	64	93/15	100/8/0.32
rd84	80	169/18	172/10/0.70
sao2	117	144/28	158/10/1.01
z4ml	16	46/13	69/9/0.18
合計	2507	3087/251	3355/116/-
比率	0.81	1.00/1.00	1.08/0.46/-

比較した 2 手法では内部でノードの共有を行っているが，我々の手法では行っていない．そのため，我々の手法ではややノード数が多くなっているが逆に段数は少ない場合が多い．我々の手法では，bi-decomposition がうまくできないような場合には shanon 展開を用いていることになる．実際には，そのような場合には他の分解手法を使うべきであり，そうすればより結果は良くなるだろうと考えられる．

3 LUT 回路の簡単化手法

本章では，特に LUT 回路に向いている論理関数の自由度の新しい表現方法とそれを利用した回路簡単化手法について述べる．

3.1 LUT 回路の簡単化手法

まず，回路の簡単化手法について簡単な例で説明する．今，図 6で示すような LUT の回路が設計されているとする．一般に回路内部の LUT の出力関数がある一定の範囲内で変化しても，回路全体の出力関数が変化しないことがある．その範囲のことを，該当する関数の**自由度**と呼ぶことにする．例えば，何らかの手法で L_2 の出力となっている関数 g_2 の自由度が計算されており，L_3 の出力となっている関数 g_3 がその自由度の範囲にあれば，図 6に示すような結線の変換を行っても回路の出力は変化しない．この例のように，内部のある LUT の出力論理の自由度を計算し，その自由度の範囲内にある回路内の他の LUT の出力論理で置き換えることを繰り返すことによって，回路の LUT 数を削減することが可能である．この操作は一般に簡単化と呼ばれる．なお，図中の 4 要素のベクトルは，g_1, g_2, g_3 の関数を表しており，次節の説明で用いる．(回路の外部入力に対する回路内部の LUT の出力関数は，外部入力数を n とすると 2^n 要素の真理値表で表現される．しかし，本稿では説明を簡単にするために，このうちの 4 要素だけを考慮する．)

従来，回路の簡単化の際には，関数の自由度を，$0, 1, *$ の 3 値をとる不完全指定論理関数で表現するのが一般的であった．しかし，不完全指定論理関数では，LUT の内部論理が自由に変更できるということも考慮に入れて関数の自由度を表現することが困難である．

そこで我々は，不完全指定論理関数ではなく **SPFD (Sets of Pairs of Functions tobe Distinguished**) と呼ばれる「関数の対の集合」を用いることで関数の自由度を表現する手法を開発した [6]．SPFD による自由度の表現は，LUT の内部論理が変更できることを考慮に入れているため，不完全指定論理関数よりも広い範囲の自由度が表現可能である．そのため，SPFD を利用すれば LUT 回路の簡単化の際により多くの置き換え可能な関数を検出でき，そのことによってより

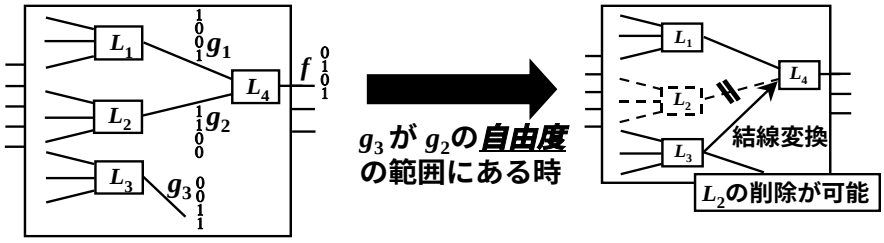

図 6: LUT 回路の簡単化

回路を簡単化できることが期待できる．

SPFD に関する考察が，文献 [9] で述べられている．これによると， SPFD は LUT 回路の簡単化以外にも応用があることが示唆されている．例えば，基本ゲートの回路において従来の技術では簡単化できないが SPFD を用いれば簡単化が可能である例を紹介している．また， SPFD は従来技術では見つけられない boolean divisor を見つけるのに利用できるかもしれないとも言及されている．

3.2 SPFD の概念

本節では， SPFD の概念を簡単に説明する．

まず SPFD の説明のために以下の定義を行う．論理関数 f と g において， $g \cdot \overline{f}$ が恒偽関数となる時 f は g を**包含する**と言う．直感的には， g が 1 となる変数割り当てでは必ず f が 1 となる時, f は g を包含すると言われる．

定義 1 以下の 2 つの条件のいずれかを満たしている場合，関数 f は 2 つの関数 g_1 と g_2 を**区別**しているという．

条件 1 f が g_1 を包含し， $\overline{f}$ が g_2 を包含する．

条件 2 f が g_2 を包含し， $\overline{f}$ が g_1 を包含する．

これは， f の ON-set および OFF-set の一方に g_1 が，他方に g_2 が含まれるということである．

この定義を用いて LUT 回路内の論理関数について，以下のことが言える．

回路の中間地点での論理関数の役割は，「その論理関数の真理値表において，ある部分が 1 であり，別のある部分が 0 であるという情報をその論理関数が入力となっている後段の素子へと伝えること」である．この考えに基づいて，ある論理関数の真理値表において， 1 でも 0 でもどちらの情報を後段に伝えても良い部分を $*$(ドントケア) として表現するのが，従来の不完全指定論理関数による自由度の表現方法と言える．

もし，素子が LUT であれば入力の否定を自由に取ることができるため，論理関数の働きは「その論理関数の真理値表において，ある部分とある部分の値が違う (一方が 1 なら他方は 0) という情報をその論理関数が入力となっている後段の素子へと伝えること」であると言い換えられる．つまり，中間地点での論理関数の働きは，「その関数によってある関数と別のある関数が**区別**されなければならない」と表現できる．より一般的にいうと，ある関数の働きはその関数が区別しなければならない関数の組を複数指定することによって表現できる．(これは後述する例の計算結果を参照して頂きたい．) この関数の自由度を表現する，複数の関数の組を **SPFD** と呼ぶことにした．

3.3 SPFD の計算方法

本節では，簡単な例を用いて SPFD の計算方法を説明する．より詳細な計算方法等は文献 [6] を参照して頂きたい．

今，図 6に示す回路において L_4 の入力となっている 2 つの関数 g_1， g_2 の自由度を計算することを考える．なお， L_4 の内部論理は XOR となっている．計算過程は以下に示す通りである．また，図 7にも計算過程を示す．

ステップ1 g_1, g_2 それぞれのの否定と肯定の全ての組み合わせ (2^2=4 通り) の論理積を求める．これらの論理積を $a_{00}, \cdots, a_{11}$ とする．この 4 つの論理積は，g_1 と g_2 を組み合わせることによってお互いに区別することができる 4 つの関数と考えられる．(L_4 の内部論理を変更することによって，L_4 の出力関数が $a_{00}, \cdots, a_{11}$ のある関数と別のある関数を区別するようにする事が必ずできるということである．例えば，現在 L_4 の出力は a_{00} と a_{01} を区別しているが，a_{00} と a_{11} を区別していない．しかし，例えば L_4 の内部論理を $g_1 \cdot g_2$ とすれば a_{00} と a_{11} を区別できるようになる．)

ステップ2 L_4 の出力関数の自由度は図 7のステップ 2 に示した真理値表上の X と X' に相当する関数を区別することと考えられる．X, X' に相当する関数は，ステップ 1 で計算した $a_{00}, \cdots, a_{11}$ を用いるとそれぞれ $a_{00}+a_{11}$ と $a_{01} + a_{10}$ となる．そのため，L_4 の出力関数を表す SPFD は，$\{(a_{00}, a_{01}), (a_{00}, a_{10}), (a_{11}, a_{01}), (a_{11}, a_{10})\}$ とより細分した条件で表現しなおすことができる．

ステップ3 ステップ 2 で求めた L_4 の区別しなければならない 4 つの関数の組は，必ず g_1 または g_2 のどちらかが区別しなければならない．(逆に，g_1 または g_2 のどちらかがステップ 4 で求めた 4 組の関数を区別するなら，L_4 の内部論理を適当に変更することで，必ず L_4 の出力関数が上記 4 組の関数を区別するようにできる．) そこで，1 組ずつ g_1 または g_2 の自由度に割り当てていく．例えば，(a_{00}, a_{01}) は g_2 が区別しているので，g_2 の自由度に割り当てる．

なおこの例では，$a_{00}, \cdots, a_{11}$ は全て単なる最小項となっているが，一般には関数となることに注意して頂きたい．

最終的に，g_1，g_2 の自由度は図 8のように表現できる．関数 g_2 に着目するとその真理値表の上から 1 番目と 4 番目の要素 (図の C と C' に相当) および 2 番目と 3 番目の要素 (図の D と D' に相当) がそれぞれ違っていれば良いということになる．この条件を満たす関数は 4 通り存在する事になるが，図 6の L_3 の出力もこの条件を満たしている．そのため，図 6で示したように，L_2 の出力を L_3 で置き換えて L_2 を削除することが可能となる．(ただしこの場合 L_4 の内部論理を XOR から XNOR に変更しなければならない.)

3.4 実験結果および考察

UCB で開発された論理合成ツール SIS を用いて 5 入力 LUT にマッピングした回路を初期回路として，3.1で述べた簡単化手法を適用した結果を表 2に示す．CPU は Sparc Station 20 上で測定した計算時間を示す．この結果から，SPFD による論理関数の自由度は現実的な時間で計算可能であり，これを用いた簡単化が有効であることがわかった．

表 2: SPFD による簡単化の結果

回路名	初期回路		簡単化結果		
	LUT	段数	LUT	段数	CPU
alu2	109	19	97	17	1.45
alu4	208	24	198	22	5.5
apex6	194	10	181	10	4.72
cordic	17	8	12	8	0.17
dalu	331	16	287	9	10.97
frg2	339	8	278	8	15.21
lal	36	4	31	3	0.17
t481	404	21	379	21	11.75
term1	69	7	45	6	0.68
ttt2	53	4	47	4	0.36
x1	111	6	99	6	3.19
x4	140	4	110	4	1.2
合計	2011	131	1764	118	55.3
比率	1.00	1.00	0.88	0.90	(秒)

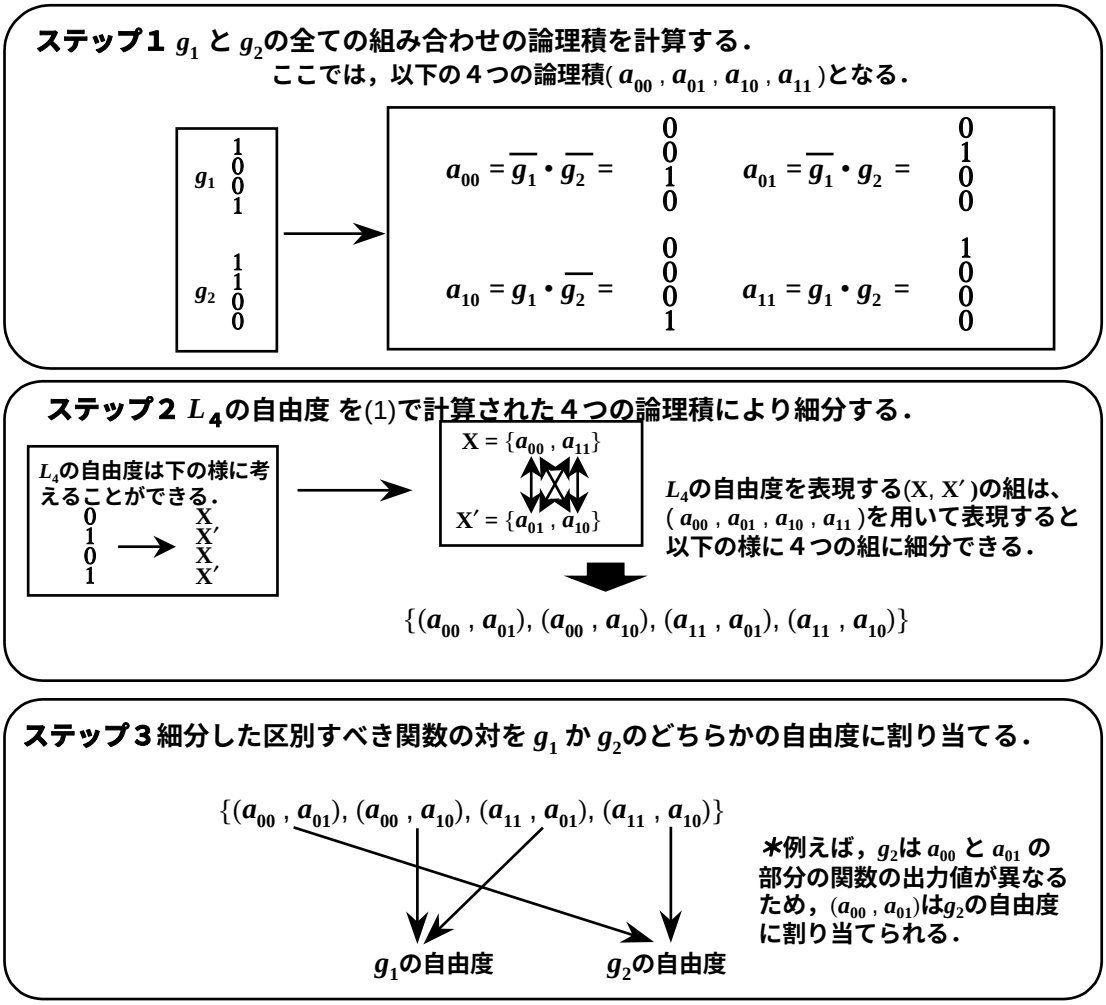

図 7: SPFD の計算過程

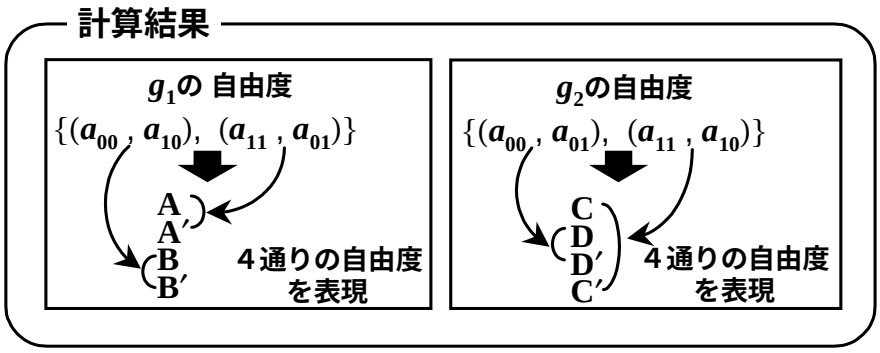

図 8: SPFD による自由度の表現

4 LUT 回路の合成手法

2, 3章で述べた手法を用いて，以下の様な LUT 回路の合成手法を実現する予定である．

ステップ１ 2章で述べた分解手法，および既存の分解手法などを用いて LUT 回路を生成する．

ステップ２ ステップ１で生成された LUT 回路を 3章で述べた手法を用いて簡単化する．

この合成手法は，できるだけ段数の少ない LUT 回路を生成することを目標としたものである．そのために，以下のような工夫を行う．

- ステップ１の分解では，段数を増やすような部分関数の共有は行わない．また段数の少なくなるような分解を優先するようにする．
- ステップ２の簡単化では，段数を増やすような結線のつなぎ換えは行わない．

つまり，部分関数の共有を考慮せずにできるだけ段数の少ない初期回路を作った後で，SPFD を用いて段数を増やさないように部分関数の共有を行おうという戦略である．

5 おわりに

本稿では，LUT 回路合成手法のための２つの要素技術についてその概要を述べた．また，それらの技術をどのように統合するかの予定についても述べた．今後は，本稿で紹介したような LUT 向けの合成機能を PARTHENON に組み入れていきたいと考えている．

参考文献

[1] Y. Nakamura, K. Oguri, A. Nagoya, M. Yukishita, and R. Nomura. High-Level Synthesis Design at NTT Systems Labs. In *Proc. of the Synthesis and Simulation Meeting and International Interchange*, pages 344–353, 1992.

[2] H. Sawada, T. Suyama, and A. Nagoya. Logic Synthesis for Look-up Table Based FPGAs Using Functional Decomposition and Support Minimization. In *Proc. ICCAD*, pages 353–358, November 1995.

[3] B. Steinbach and A. Wereszczynski. Synthesis of Multi-Level Circuits Using EXOR-Gates. In *Proc. of Reed-Muller'95*, pages 161–168, August 1995.

[4] T. Sasao and Jon T. Butler. On Bi-Decompositions of Logic Functions. In *Notes of International Workshop on Logic Synthesis (IWLS'97)*, May 1997.

[5] S. Muroga, Y. Kambayashi, H. C. Lai, and J. N. Culliney. The Transduction Method-Design of Logic Networks Based on Permissible Functions. *IEEE Transactions on Computers*, 38(10):1404–1424, October 1989.

[6] S. Yamashita, H. Sawada, and A. Nagoya. A New Method to Express Functional Permissibilities for LUT based FPGAs and Its Applications. In *Proc. ICCAD*, pages 254–261, November 1996.

[7] S. Yamashita, H. Sawada, and A. Nagoya. New Methods to Find Optimal Non-Disjoint Bi-Decompositions. In *ASP-DAC '98, to appear*, February 1998.

[8] S. Yang. Logic synthesis and optimization benchmarks user guide version 3.0. *MCNC*, January 1991.

[9] R. K. Brayton. Understanding SPFDs: A New Method for Specifying Flexibility. In *Notes of International Workshop on Logic Synthesis (IWLS'97)*, May 1997.