THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS
TECHNICAL REPORT OF IEICE.

BDD を用いた多出力関数の分解の一手法について

笹尾　勤† 松浦 宗寛†

† 九州工業大学情報工学部電子情報工学科 〒 820–8502 福岡県飯塚市大字川津 680–4
E-mail: †{sasao,matsuura}@cse.kyutech.ac.jp

あらまし 与えられた多出力関数を, 中間出力を有する二つの論理回路に分解する方法を示す. 設計には, 多出力論理関数の特性関数を表現する BDD(BDD for CF) を用いる. また, 多数のベンチマーク関数を中間出力を有する LUT カスケードで実現した場合の実験結果を示す.

キーワード LUT カスケード, 多出力論理関数, 特性関数, 二分決定グラフ, BDD for CF

A Method to Decompose Multiple-Output Logic Functions

Tsutomu SASAO† and Munehiro MATSUURA†

† Department of Computer Science and Electronics, Kyushu Institute of Technology
680–4, Kawazu, Iizuka, Fukuoka, 820–8502 Japan
E-mail: †{sasao,matsuura}@cse.kyutech.ac.jp

Abstract This paper shows a method to decompose a given multiple-output circuit into two circuits with intermediate outputs. We use a BDD for characteristic function (BDD for CF) to represent a multiple-output function. Many benchmark functions were realized by LUT cascades with intermediate outputs.

Key words LUT cascade, Multiple-output function, Characteristic function, Binary decision diagram, BDD for CF

1. はじめに

論理関数の分解 [1] は, FPGA [19] の設計に応用されている. FPGA の設計では, 各モジュールの入力数は高々5 程度で, BDD を用いて分解する手法が一般的である [3], [7], [10]. 一般に, ランダムな (ほとんどすべての) 論理関数は, 分解不能である [17] が, コンピュータ等で使用する制御回路や算術演算回路等の多くの実用的関数は, 分解可能なものが多い [12].

関数が $f(X_1, X_2) = g(h(X_1)X_2)$ と表現できるとき, $h(X_1)$ と $g(h, X_2)$ を別々に実現し, 直列に接続することを繰り返すことにより, LUT 型の FPGA の設計が可能である. 単一出力の論理関数を関数分解を用いて実現するのは, 比較的容易である. しかし, 多出力関数の場合の分解・構成法に関しては, まだ決定的な方法は知られていない. 現在までに,

（1） MTBDD を用いる方法 [7]
（2） 出力をいくつかのグループに分割して構成する方法 [7]
（3） 分割理論を用いる方法 [15], [18]
（4） 代入による方法 [14]
（5） Hyper Function を用いる方法 [5]
（6） ECFN を用いた時分割実現による方法 [13]
（7） 以上の方法の組み合わせによる方法 [14]

などが提案されている.

本論文では, 多出力論理関数の特性関数を表現する BDD(BDD for CF [2], [16]) を用いた関数分解の方法を示す. 本方法は, BDD for CF を用いており, 中間出力を有するカスケードの合成に適している. アルゴリズムは比較的単純であるため, 設計の見通しはよい.

従来の FPGA では, 接続線の遅延の方が論理部の遅延よりもはるかに大きい. これは, FPGA の速度の基本的な制限の一つとなっている.

2. 諸定義ならびに基本的性質

［定義 1］ 入力変数を $X = (x_1, x_2, \ldots, x_n)$. 多出力関数を $F = (f_1(X), f_2(X), \ldots, f_m(X))$ とする. **多出力関数の特性関数**を

$$\chi(X, Y) = \bigwedge_{i=1}^{m} (y_i \equiv f_i(X))$$

とする. ここで y_i は出力を表す変数である.

n 入力 m 出力関数の特性関数は, $(n + m)$ 変数の二値論理関数であり, 入力変数 x_i $(i = 1, 2, \ldots, n)$ の他に, 各出力 f_i に対して出力変数 y_i を用いる. $B = \{0, 1\}$ とする. $\vec{a} \in B^n$ か

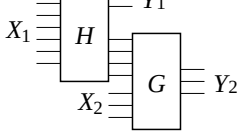


図 1　中間出力のある関数分解

つ $F(\vec{a}) = (f_0(\vec{a}),\ f_1(\vec{a}), \ldots, f_{m-1}(\vec{a})) \in B^m$ とする．いま，$\vec{b} \in B^m$ とすると，

$$\begin{aligned}\chi(\vec{a},\vec{b}) &= 1 \quad (\vec{b} = F(\vec{a})) \\ &= 0 \quad (\text{上以外の時})\end{aligned}$$

となる．

［定義 2］　多出力関数 $F = (f_1, f_2, \ldots, f_m)$ の BDD for CF とは，F の特性関数 χ を表現する BDD である．但し，BDD の変数は，根節点を最上位としたとき，変数 y_i は，f_i に依存する変数の下に置く．

［定義 3］　BDD for CF において，出力を表す節点 y_i とその節点から出る枝のうち，定数 0 に接続する枝を取り除き，それ以外の枝がさす節点に，節点 y_i の親から直接枝をつなぐ．この作業を y_i を表現する全ての節点に対して実行する操作を**出力変数 y_i の短絡除去**という．

［定義 4］　BDD の高さ k での幅とは，変数 z_k と z_{k+1} の間の枝の本数をいう．ここで，同じ節点に接続している枝は一つと数える．

［定理 1］　X_1, X_2 を入力変数の集合，Y_1, Y_2 を出力変数の集合とする．BDD for CF の変数順序を (X_1, Y_1, X_2, Y_2) とするとき，BDD for CF の (X_1, Y_1) における幅を W とする．ここで，W を数える際，出力を表現する変数から，定数 0 に向かう枝は無視する．多出力関数を図 1 の回路で実現する場合，二つの回路 H と G の間の必要かつ十分な接続線数は $\lceil \log_2 W \rceil$ である．

(証明) BDD for CF の構成法から，出力関数 Y_1 と Y_2 が，図 1 の回路で表現可能であることは，明らかである．BDD for CF において，Y_1 の出力を表現する節点を短絡除去すると，Y_1 以外の関数を表現する BDD for CF が得られる．また，この操作によって，BDD の幅は増えることはない．X_1 と X_2 の間を分離する節点数を W とする．いま，関数分解 $g(h(X_1), X_2)$ の分解表を考えると，その列複雑度は W に等しい．従って，回路 H と回路 G の間の配線は，$\lceil \log_2 W \rceil$ 本あれば十分である．また，分解表の列複雑度は W なので，二つのブロック間の配線は少なくとも $\lceil \log_2 W \rceil$ 本必要である．　(証明終)

BDD for CF の変数順序を (X_1, Y_1, X_2, Y_2)，$Y_1 = (y_1, y_2, \ldots, y_k)$ とする．$f_i(X_1)$ $(i = 1, 2, \ldots, k)$ は，図 1 の回路 H で直接実現する．(X_1, Y_1) における BDD の幅を W とする．W 本の線に $t = \lceil \log_2 W \rceil$ ビットの異なる 2 進数を割り当てる．二つのブロック間を接続する線が実現する関数を $u_1, u_2, \ldots, u_t$ とすると，$Y_2 = (f_{k+1}, f_{k+2}, \ldots, f_m)$ は，$(u_1, u_2, \ldots, u_t, X_2)$ の関数として表現可能であり，その BDD for CF は図 2 のようになる．

［例 1］　二つの 2 ビットの 2 進数を加算する回路 (ADR2) を中

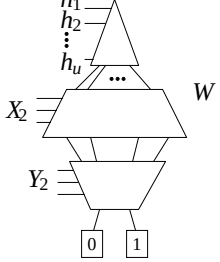


図 2　BDD for CF による Y_2 の実現法

間出力を有する関数分解を用いて実現する．ADR2 の入出力の関係は次のように定義できる．

$$\begin{array}{cccc} & & a_1 & a_0 \\ +) & & b_1 & b_0 \\ \hline & s_2 & s_1 & s_0 \end{array}$$

これより，

$$\begin{aligned} s_0 &= a_0 \oplus b_0 \\ s_1 &= a_0 b_0 \oplus (a_1 \oplus b_1) \\ s_2 &= MAJ(a_0 b_0, a_1, b_1) = a_0 b_0 (a_1 \vee b_1) \vee a_1 b_1. \end{aligned}$$

変数の分割として，$X_1 = (a_0, b_0)$, $Y_1 = (s_0)$, $X_2 = (a_1, b_1)$, $Y_2 = (s_1, s_2)$ とおく．　変数順序は，$(X_1, Y_1, X_2, Y_2) = (a_0, b_0, s_0, a_1, b_1, s_1, s_2)$ となる．

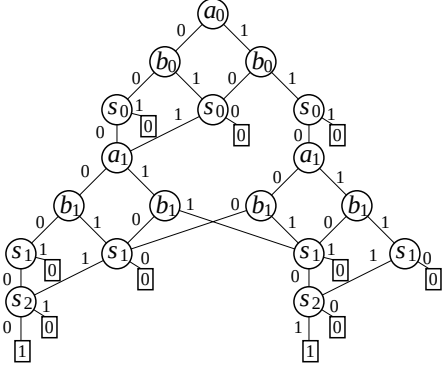
BDD for CF は，図 3(a) のようになる．$X = (X_A, X_B)$, $X_A = (X_1, Y_1)$, $X_B = (X_2, Y_2)$ と分割すると，X_A での BDD の幅 W は 2 となる．従って，回路 H と回路 G の間を連結する線は $\lceil \log_2 W \rceil = 1$ 本あればよい．出力 s_0 は，X_1 のみの関数として表現可能である．次に，新しい中間変数 u_1 を導入し，BDD の上部を u_1 を変数とする決定木で置き換える (図 3(b))．

次に，図 3(c) に示すように，u_1, a_1, b_1 を入力，s_1, s_2 を出力とする MTBDD を構成する．これより，図 4 に示すような，ADR2 の回路の真理値表が得られる．　(例終り)

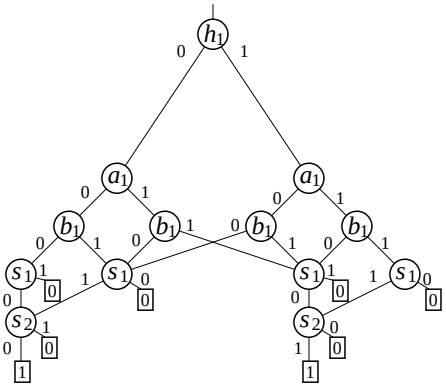
3. LUT 回路構成への応用

関数分解を繰り返し適用することにより，LUT カスケードや LUT ランダム回路を構成できる．ここで，LUT の入力数を $k \geqq 3$ とする

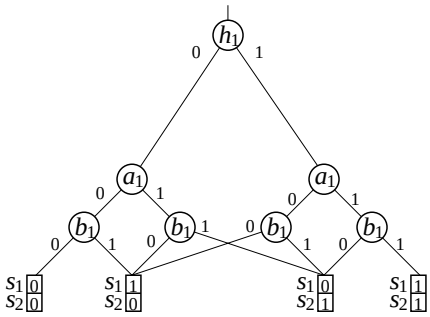
BDD for CF を最小化し，根の部分から，k 変数抽出する．この際，出力を表す変数は数えない．次に，BDD の幅 W を求める．この際，出力を表現する変数から定数 0 に向かう枝は無視する．$u = \lceil \log_2 W \rceil$ 個の中間変数を導入する．$u \geqq k$ の場合には，関数分解では対処できない．W 個の部分関数に，u ビットの二進符号を割り当てる．ここでは，簡単のため，一つの部分関数に一つの二進符号を割り当てる．ドント・ケアも特に考慮しない．未使用の符号は，適当な部分関数 (1 つ) に割り当てる．LUT で，k 入力 $(u + w)$ 出力の回路を構成する．ここで，w は，この抽出で，実現される中間出力の個数を表す．抽出した部分の出力変数を短絡除去する．k 個の変数を u 個の中間変数で置き換え，BDD for CF を再構築する．以下同様に，根から k 変数抽出する．



(a) 回路 H を表現する BDD for CF



(b) 回路 G を表現する BDD for CF



(c) 回路 G を表現する MTBDD

図 3　BDD を用いた加算器の合成

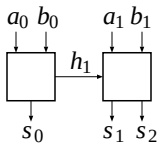


図 4　2 ビット加算器 (ADR2)

4. LUT カスケード設計アルゴリズム

一般に, 実用的な多出力論理関数は 1 つの BDD for CF で表現するとメモリ量が大きくなり過ぎる. また, 1 つの BDD for CF で表現できる場合でも, LUT カスケードで実現できない場合がある. そのため, ここでは出力をいくつかのグループに分割して, 各出力グループを別々にカスケード実現することを考える. 出力を分割する時, それぞれの出力集合の依存変数は少ない方が望ましい. ここでは, まず依存変数がなるべく増えないような順序に出力関数を並べ, 次にその順序で出力集合を分割する方策を用いる. アルゴリズム 1 を用いて決定した出力順序で, 出力数を増やしながら LUT カスケードで実現可能かどうか確認していく. 本手法では, LUT カスケードで実現不可能な出力関数グループは生じにくい.

4.1　出力順序の決定

［アルゴリズム 1］ (出力関数の順序付け)

出力関数の初期順序を $(f_0, f_1, \ldots, f_{m-1})$ とする.

（1） $i \leftarrow 0$, $j \leftarrow 0$, $minT \leftarrow \infty$, $minorder \leftarrow$ 初期順序.

（2） f_i と f_j の位置を入れ換える.

（3） $T \leftarrow \sum_{k=0}^{m-1} |\bigcup_{l=0}^{k} sup(f_l)|$ を計算する [2]. ここで, $sup(f_l)$ は f_l の依存変数の集合を示す.

（4） $T < minT$ ならば $minT \leftarrow T$, $minorder \leftarrow$ 現在の出力順序 とする.

（5） $j < m-1$ ならば $j \leftarrow j+1$ として 2 へ戻る.

（6） $j \leftarrow 0$, $i < m-1$ ならば $i \leftarrow i+1$ として 2 へ戻る.

（7） $minT$ が更新されたならば $i \leftarrow 0$ として 2 へ戻る. そうでなければ $minorder$ を出力順序として終了する.

ここで用いる値 T は, ある順序で出力関数をグループに加えて行く際のグループの依存変数の個数を示す. 出力の順序を変えながら T の値を計算し, 依存変数の個数の増加を抑える順序を求める.

4.2　変数順序の決定

与えられた多出力関数を BDD for CF を用いて $g(h_1(Z_1), h_2(Z_1), \ldots, h_u(Z_1), Z_2)$ の形に関数分解することを考える. ここで, Z_i は入力変数または出力変数を表す. 集合 $\{Z_1\}$ が出力変数を含む場合, その段の LUT は出力変数が表す関数を出力する. つまり, LUT カスケードの途中で出力を生成する. 出力変数が根に近ければカスケードの入力に近い LUT で出力が得られ, 後段の LUT の入力数を削減できる. そこで, BDD for CF の初期変数順序として, 出力変数がなるべく根に近い位置になるような変数順序を考える.

［アルゴリズム 2］ (変数順序の決定)

（1） アルゴリズム 1 を適用し, 得られた出力順序を $(f_0, f_1, \ldots, f_{m-1})$ とする.

（2） f_i の依存変数の集合を $sup(f_i)$, f_i を表す出力変数を y_i とする.

（3） BDD の根から $sup(f_0), y_0, sup(f_1) - sup(f_0), y_1, sup(f_2) - sup(f_1) - sup(f_0), y_2, \ldots, sup(f_{m-1}) - sup(f_{m-2}) - \cdots - sup(f_0), y_{m-1}$ となるように変数順序を決定する. このとき, $sup(f_i)$ の変数順序は SBDD の節点数を最小化した変数順序をそのまま使う.

関数分解を行う時は, この変数順序を初期変数順序として, BDD for CF の幅の総和を減らすように変数順序を変更する.

4.3　BDD for CF を用いた LUT カスケード実現法

ここでは, BDD for CF が与えられた時に LUT カスケードを実現するアルゴリズムを示す.

LUT の最大入力数を k, 最大出力数を r とする. (Z_1, Z_2) を変数の分割として, $f(Z) = g(h_1(Z_1), \ldots, h_u(Z_1), Z_2)$ の形で関数分解した時, $|Z_1| \leqq k$ かつ, (Z_1 中の出力数$+\lceil \log_2 W \rceil$) $\leqq r$

の時 LUT で実現可能である．ここで，$\{Z_1\}$ を束縛集合という．また，W は高さ $|Z_2|$ における BDD の幅である．

［アルゴリズム 3］ (BDD for CF を用いた LUT カスケード実現)

与えられた多出力関数を $F = (f_0, f_2, \ldots, f_{m-1})$, F の依存変数を Z, 入力数を n, 出力数を m とする．F を表現する BDD for CF を bdd_{cf} とする．根の高さを $n+m$, 終端節点の高さを 0 とする．高さ i に対応する変数を z_i, $\{Z_a\}$ を変数の集合とする．

（1） $i \leftarrow n+m+1$, $a \leftarrow 0$, $Z_a \leftarrow \phi$ とする．

（2） $i \leftarrow i-1$, $\{Z_a\} \leftarrow \{Z_a\} \cup \{z_i\}$

（3） $\{Z_a\} \cup \{z_i\}$ を束縛集合とする．関数分解し，LUT 実現可能であれば 2 へ戻る．

（4） $|Z_a| = 1$ ならば LUT カスケード実現不可能として終了する．

（5） (Z_a, Z_b) を変数の分割として $g(h(Z_a), Z_b)$ の形で関数分解を行う．ここで，$\{Z_b\} = \{Z\} - \{Z_a\}$．$Z_a$ の部分の MTBDD を作り，LUT 回路として出力する．Z_a の部分のすべての出力変数を短絡除去する．列複雑度を μ_a, $u_a = \lceil \log_2 \mu_a \rceil$ とすると，$u_a + |Z_b| \leqq k$ ならば関数 g を LUT 回路として出力し終了する．

（6） $i \leftarrow |Z_b| + 1$, $a \leftarrow a+1$, $\{Z\} \leftarrow \{Z\} - \{Z_a\}$, $\{Z_a\} \leftarrow \phi$, $bdd_{cf} \leftarrow$ 関数 g を表現する BDD for CF として 2 へ戻る．

BDD for CF に対して，$\{Z_1\}$ を束縛集合として関数分解を行った場合，列複雑度を μ とすると $u = \lceil \log_2 \mu \rceil$ 個の中間変数と $\{Z_1\}$ 中の出力変数の個数がこの段の LUT の出力数となる．k 入力 r 出力 LUT を用いて関数分解が可能な条件は，(前段からの中間変数の個数 + 入力変数の個数) $\leqq k$ かつ，(出力変数の個数 + この段の中間変数の個数u) $\leqq r$ の二つの条件を満たすことである．

4.4 LUT カスケード実現法

ここでは，与えられた多出力論理関数をカスケード実現するアルゴリズムを示す．

［アルゴリズム 4］ (多出力論理関数の LUT カスケード実現)

アルゴリズム 1 で得られた出力順序を $(f_0, f_1, \ldots, f_{m-1})$ とし，F_a を関数の集合とする．

（1） $i \leftarrow 0$, $F_a \leftarrow \phi$.

（2） $F_a \cup \{f_i\}$ の BDD for CF を構成し，変数順序を最適化する．初期変数順序はアルゴリズム 2 を使用して求める．

（3） アルゴリズム 3 で LUT カスケード実現することを試みる．

（4） LUT カスケード実現可能な場合

(a) $i = m$ ならば F_a の LUT カスケード回路を出力して終了する．

(b) そうでなければ，$F_a \leftarrow F_a \cup \{f_i\}$, $i \leftarrow i+1$ として 2 へ戻る．

（5） LUT カスケード実現不可能な場合

(a) $|F_a| = 0$ ならば LUT カスケード実現不可能として終了する．

(b) そうでなければ，F_a の LUT カスケード回路を出力し，$F_a \leftarrow \phi$ として 2 へ戻る．

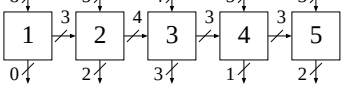


図 5 vg2 の LUT カスケード回路

5. 実験結果

第 4 章のアルゴリズムを C 言語で実装し，MCNC89 ベンチマーク関数に適用した．LUT の入力数 k を 8〜10 にした場合の結果を表 1 に示す．表中の Name は関数名，In は入力数，Out は出力数，LUT は総 LUT 数，Lvl は最大段数，Cas はカスケード数を表す．表中の記号 – は本手法を用いてカスケード実現することができなかったことを示す．本手法は関数分解に基づいているため，BDD の幅を μ とすると，関数分解が可能な条件は $\lceil \log_2 \mu \rceil < k$ となる．従って，μ が大きい過ぎると関数分解不可能となり，カスケード実現できない場合がある．実行環境は，IBM PC/AT 互換機，Pentium4 2.0GHz，メモリ 512MByte．OS は Windows2000，cygwin 上で gcc を用いてコンパイルした．本アルゴリズムでは，出力のグループ分けを行うために，一つずつ出力をグループに加えて BDD for CF を構成し変数順序最適化を行う．k が大きくなると，LUT カスケードに対応する BDD for CF が大きくなる．LUT カスケードで構成可能な限り，グループの出力を一つずつ増やしながら変数順序最適化を行うため，大きな BDD for CF の変数順序最適化を行う回数が増える．このため，k が大きくなると多くの実行時間が必要となる．実行時間のほとんどは，BDD for CF の変数順序最適化のための時間である．

ベンチマーク関数 vg2 を $k = 8$ で LUT カスケード実現した際の回路構造を図 5 に示す．LUT 数は 21 個 (8-LUT15 個，6-LUT2 個)，段数は 5 段，2〜4 段目に中間出力がある．中間出力が有効に活用されていることがわかる．

次に，文献 [8] の方法との比較を行った．文献 [8] の方法は，MTBDD に基づいており，多出力関数の出力をいくつかのグループに分割して MTBDD で表現し，LUT カスケードで実現する．MTBDD の幅が大きすぎるため分解不可能な場合は，OR 分割を用いて，カスケードを分割する．$k = 10$ とした場合の本手法と文献 [8] の方法との比較を表 2 に示す．文献 [8] の方法では，一つのグループの出力数は 8 として，出力を分割している．

多くの場合，本手法で実現した方が文献 [8] の方法に比べて少ない LUT 数で実現できた．また，カスケードの個数も少ないが，段数は多くなっている．文献 [8] の方法では，実現した LUT カスケードの LUT の個数や段数にかかわらず，分割するグループの出力数を固定しているため，カスケードの個数は多くなり，段数は小さくなる傾向にある．本手法では，LUT カスケード可能な限り，一つのグループの出力の個数を増やすので，段数は多くなり，カスケードの個数は少なくなる．

MCNC ベンチマーク関数に加え，16-bit の二進数を 5 桁の BCD コードに変換する回路を設計した．この変換回路は，様々

Name	In	Out	$k=8$			$k=9$			$k=10$		
			LUT	Lvl	Cas	LUT	Lvl	Cas	LUT	Lvl	Cas
C1908	33	25	–	–	–	433	19	6	320	11	5
C432	36	7	143	17	2	115	16	1	78	11	1
apex1	45	45	272	23	2	167	19	1	115	12	1
apex2	39	3	56	12	1	37	8	1	38	8	1
apex3	54	50	254	21	2	165	19	1	129	14	1
apex6	135	99	479	28	5	470	22	5	395	18	4
apex7	49	37	200	24	2	159	19	1	111	13	1
b9	41	21	51	9	1	42	7	1	39	6	1
c8	28	18	79	10	2	75	9	2	66	8	2
cc	21	20	30	4	1	30	4	1	27	3	1
cht	47	36	64	10	1	56	8	1	53	7	1
cm150a	21	1	12	4	1	9	4	1	8	3	1
comp	32	3	11	5	1	11	5	1	9	4	1
count	35	16	27	6	1	29	6	1	25	5	1
duke2	22	29	59	7	1	49	5	1	46	4	1
e64	65	65	74	10	1	72	8	1	72	8	1
example2	85	66	309	41	1	219	27	1	168	19	1
frg1	28	3	19	6	1	15	5	1	13	4	1
k2	45	45	272	23	2	167	19	1	115	12	1
lal	26	19	30	5	1	29	4	1	26	4	1
misex2	25	18	29	5	1	26	4	1	25	4	1
mux	21	1	12	4	1	9	4	1	8	3	1
my_adder	33	17	27	6	1	20	4	1	23	4	1
pcler8	27	17	52	8	2	58	8	1	43	6	1
rot	135	107	1124	28	15	1028	38	11	1248	34	8
seq	41	35	153	20	1	104	13	1	84	9	1
term1	34	10	120	18	1	65	10	1	50	8	1
too_large	38	3	56	12	1	37	8	1	38	8	1
ttt2	24	21	74	9	2	74	9	2	54	6	1
unreg	36	16	34	7	1	31	6	1	28	5	1
vg2	25	8	21	5	1	18	4	1	16	4	1
x1	51	35	312	21	4	259	18	4	224	14	3
x3	135	99	479	28	5	470	22	5	395	18	4
x4	94	71	272	24	3	214	17	3	161	13	3

な実現法が知られている．文献 [9] には, 一つ実現法として 13 個のモジュール (ROM) を使った回路が示されている．図 6 にアルゴリズム 4 で LUT カスケード実現した際の回路構造を示す．入力の二進数は $x_1, x_2, \ldots, x_{16}$, 出力の BCD コードは $f_1, f_2, \ldots, f_{19}$ と表現している．この回路では, 3 個の 11 入力 LUT を使用している．ここで, 最上位桁の最上位 bit である f_0 は, 常に 0 となるので省略している．

さらに, この回路を SIS を用いて設計した．真理値表から設計した場合, 時間がかかり過ぎ, 設計できなかったため, この回路を SBDD で表現して節点数最小化を行い, マルチプレクサ回路を生成して, これを初期回路とした．以下のコマンドを適用した場合, LUT の個数は 644 個, 最大段数は 15 となった．

> script.rugged

> xl_split -n 10

> xl_pratition -n 10

同様に, 次のコマンドを適用した場合は, LUT の個数は 215 個, 最大段数は 4 となった．

> xl_split -n 10

> xl_pratition -n 10

この例では, アルゴリズム 4 は SIS と比較して, かなりより結果となった．

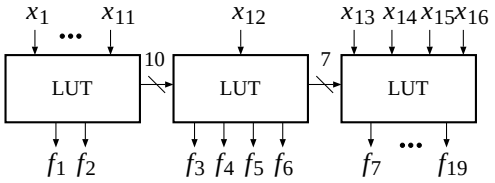


図 6　二進数 BCD 変換回路

6. 今後の課題

本手法を用いると, 多出力関数を効率良く LUT カスケードで実現できる．しかし, k の値が大きいとき, BDD for CF が大きくなり, 変数順序最適化に非常に時間がかかる．また, 本手法では, 中間出力を有効に使うことを考え, BDD for CF の初期変数順序を決定している．この変数順序が BDD for CF の節点数を増大させる場合, BDD for CF を構成できない場合もある．これらの問題を解決するために, アルゴリズムを適用する前に出力をいくつかのグループに分割する方法や, 節点数を増大させないような初期変数順序の決定法, そして効率の良い変数順

Name	In	Out	本手法			文献 [8]		
			LUT	Lvl	Cas	LUT	Lvl	Cas
C1908	33	25	320	11	5	3015	18	30
C432	36	7	78	11	1	80	11	1
apex1	45	45	115	12	1	213	10	5
apex2	39	3	38	8	1	29	7	1
apex3	54	50	129	14	1	213	9	6
apex6	135	99	395	18	4	728	18	18
apex7	49	37	111	13	1	119	6	5
c8	28	18	66	8	2	138	8	3
cht	47	36	53	7	1	128	4	8
comp	32	3	9	4	1	8	4	1
count	35	16	25	5	1	134	8	3
duke2	22	29	46	4	1	51	3	4
e64	65	65	72	8	1	162	9	8
example2	85	66	168	19	1	327	13	9
frg1	28	3	13	4	1	13	4	1
k2	45	45	115	12	1	213	10	5
lal	26	19	26	4	1	24	2	3
misex2	25	18	25	4	1	18	3	2
my_adder	33	17	23	4	1	81	7	2
o64	130	1	30	16	1	30	16	1
pcler8	27	17	43	6	1	59	6	2
rot	135	107	1248	34	8	1383	24	16
seq	41	35	84	9	1	132	8	4
term1	34	10	50	8	1	37	6	2
too_large	38	3	38	8	1	32	7	1
ttt2	24	21	54	6	1	47	4	3
unreg	36	16	28	5	1	142	8	3
vg2	25	8	16	4	1	29	5	1
x1	51	35	224	14	3	340	17	5
x4	94	71	161	13	3	324	9	10

序最適化法の開発が必要である.

7. あとがき

本論文では, BDD for CF を用いて, 多出力論理回路の分解を行う方法を示した. 本手法は, 中間出力を有するカスケードの実現に有効である. 従来の MTBDD を用いた方法では, 中間出力を有する回路の分解は, 取り扱えなかった. BDD for CF において, 出力を表す変数が根に近いところに存在すれば, 本手法を用いると中間出力として生成できるので, 次段の回路の入力数を削減できる. また, 多くの関数では, MTBDD よりも BDD for CF の節点数は小さくなる. そのため MTBDD を構成できないような関数に対しも, 本手法が有効な場合がある. ただし, 出力を表す変数が全て, 葉に近いところに存在する場合には, 本手法は, MTBDD を用いた場合とほぼ等価となり, あまり効果は期待できない.

謝　辞

本研究は, 一部, 文部科学省・科学研究費補助金, および, 文部科学省・北九州地域知的クラスター創成事業の補助金による.

文　献

[1] R. L. Ashenhurst, "The decomposition of switching functions," *In Proceedings of an International Symposium on the Theory of Switching*, pp. 74-116, April 1957.

[2] P. Ashar and S. Malik, "Fast functional simulation using branching programs," *Proc. International Conference on Computer Aided Design*, pp. 408-412, Nov. 1995.

[3] Ting-Ting Hwang, R. M. Owens, M. J. Irwin, and Kuo Hua Wang, "Logic synthesis for field-programmable gate arrays," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, Vol. 13, No. 10, pp. 1280-1287, Oct. 1994.

[4] 井口幸洋, 笹尾勤, "LUT カスケード方式アーキテクチャ," 電子情報通信学会 コンピュータシステム専門委員会 第 2 種研究会第 1 回 リコンフィギャラブルシステム研究会, 2003 年 9 月 18 日 ~19 日, 熊本大学.

[5] J.-H. R. Jian, J.-Y. Jou, and J.-D. Huang, "Compatible class encoding in hyper-function decomposition for FPGA synthesis," *Design Automation Conference*, pp. 712-717, June 1998.

[6] 草野将樹, 笹尾勤, 松浦宗寛, 井口幸洋, "順序回路方式 LUT カスケードにおけるメモリパッキングについて," 電子情報通信学会 VLSI 設計技術研究会, (発表予定), 小倉 (2003-11).

[7] Y-T. Lai, M. Pedram and S. B. K. Vrudhula, "BDD based decomposition of logic functions with application to FPGA synthesis", *30th ACM/IEEE Design Automation Conference*, June 1993.

[8] A. Mishchenko and T. Sasao, "Logic Synthesis of LUT Cascades with Limited Rails," 電子情報通信学会 VLSI 設計技術研究会, VLD2002-9, 琵琶湖 (2002-11).

[9] S. Muroga, *VLSI System Design*, John Wiley & Sons, 1982, pages 293-306

[10] R. Murgai, R. K. Brayton, and A. Sangiovanni-Vincentelli, *Logic Synthesis for Field-Programmable Gate Arrays*, Kluwer, 1995.

[11] Qin Hui, 笹尾 勤, 松浦宗寛, 永山 忍, 中村和之, 井口幸洋, "順序回路方式 LUT カスケードについて," 電子情報通信学会, 第 2 種研究会・第 7 回システム LSI ワークショップ, 2003 年 11 月 25 日 ~27 日, 北九州 (発表予定).

[12] T. Sasao and M. Matsuura, "DECOMPOS: An integrated system for functional decomposition," *1998 International Workshop on Logic Synthesis*, Lake Tahoe, June 1998.

[13] T. Sasao, M. Matsuura, and Y. Iguchi, "A cascade realization of multiple-output function for reconfigurable hardware," *International Workshop on Logic and Synthesis(IWLS01),* Lake Tahoe, CA, June 12-15, 2001, pp.225-230.

[14] H. Sawada, T. Suyama, and A. Nagoya, "Logic synthesis for look-up table based FPGAs using functional decomposition and support minimization," *ICCAD*, pp. 353-359, Nov. 1995.

[15] C. Scholl and P. Molitor, "Communication based FPGA synthesis for multi-output Boolean functions," *Asia and South Pacific Design Automation Conference*, pp. 279-287, Aug. 1995.

[16] C. Scholl, R. Drechsler, and B. Becker, "Functional simulation using binary decision diagrams," *ICCAD'97*, pp. 8-12, Nov. 1997.

[17] C. E. Shannon, "The synthesis of two-terminal switching circuits," *Bell Syst. Tech. J.* 28, 1, pp. 59-98, 1949.

[18] B. Wurth, K. Eckl, and K. Anterich, "Functional multiple-output decomposition: Theory and implicit algorithm," *Design Automation Conf.*, pp. 54-59, June 1995.

[19] http://www.xilinx.com