社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

LUTカスケードを用いたCAMエミュレータについて

中原　啓貴† 　笹尾　勤† 　松浦　宗寛†

† 九州工業大学情報工学部〒 820-8502 福岡県飯塚市大字川津 680-4

あらまし　k 個の異なる登録ベクトルに対して 1 から k までのアドレスを対応させた表を, アドレス表という. アドレス表を表現する関数をアドレス生成関数という. アドレス生成関数を LUT カスケードを用いて FPGA 上に実現する方法を示す. アドレス生成関数を実現する部分は FPGA 上の組込メモリを用いて実現する. 登録ベクトルの追加と削除は FPGA 上の組込プロセッサが行う. Xilinx 社が提供している CAM を用いた実現法と比較して, 本手法はハードウェア量を大幅に削減可能できる. また, 登録ベクトルの追加と削除は, LUT カスケードのセル数に比例する時間で実現可能であることを示す.

キーワード　CAM, LUT カスケード

A CAM Emulator Using Look-Up Table Cascades

Hiroki NAKAHARA†, Tsutomu SASAO†, and Munehiro MATSUURA†

† Department of Computer Science and Electronics, Kyushu Institute of Technology, Iizuka 820-8502

Abstract　An address table relates k different registered vectors to the addresses from 1 to k. An address generation function represents the address table. This paper presents a realization of an address generation function with an LUT cascade on an FPGA. The address generation function is implemented by BRAMs of an FPGA, while the addition and the deletion of registered vectors are implemented by an embedded processor on the FPGA. Compared with CAMs produced by the Xilinx Core Generator, our implementations are smaller and faster. This paper also shows that the addition and deletion of a registered vector can be done in a time that is proportional to the number of cells in the LUT cascade.

Key words　CAM, LUT cascade

1. はじめに

k 個の異なる登録ベクトルに対して 1 から k までのアドレスを対応させた表を, **アドレス表** [12] という. アドレス表を表現する関数をアドレス生成関数 [12] という. アドレス生成関数はインターネットのアドレスリスト [5] [8] やメモリの修正回路 [3], パターンマッチング [10], 辞書などに応用可能である [12].

アドレス表は Content Addressable Memory(CAM) [7] で直接実現可能である. CAM を直接実現するには特別のハードウェアが必要であるため [11], 通常のメモリやゲートを組合わせて CAM と同等の機能を実現する方法が考案されている [2] [6] [9]. 文献 [13] では CAM 機能を LUT カスケードで実現可能する方法を示している. 要素数が一定値以下のアドレス表はカスケードの LUT の内容を変更することにより実現でき, 回路構造は固定のままでよい. 本稿では, LUT カスケードを用いたアドレス生成回路を Xilinx 社の FPGA 上に実現する方法, 及び FPGA 上の組込プロセッサを用いたアドレス生成回路の修正方法について述べる. また Xilinx 社の提供している CAM [16] との比較を行い, 本手法の有効性について述べる. 本手法は従来の方法で FPGA 上に実現した CAM に比べ登録ベクトルの更新には余分の時間がかかるものの, 必要ハードウェアは大幅に削減可能である.

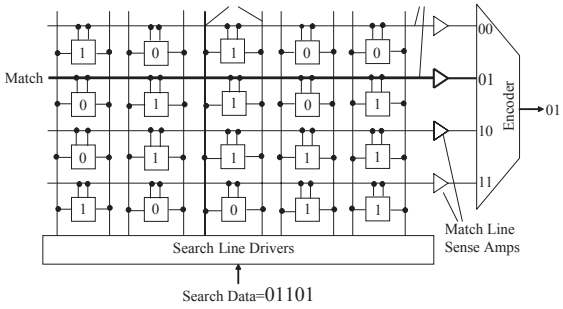

図 1　CAM

2. 諸定義及び基本的性質

[定義 2.1]　関数 $F(\vec{X}) : B^n \rightarrow \{0, 1, \ldots, k\}$ において k 個の異なる登録ベクトル $\vec{a}_i \in B^n$ $(i = 1, 2, \ldots, k)$ に対して, $F(\vec{a}_i) = i$ $(i = 1, 2, \ldots, k)$ が成立し, それ以外の $(2^n - k)$ 個の入力ベクトルに対しては, $F = 0$ が成立するとき, $F(\vec{X})$ を**重み k のアドレス生成関数**という. アドレス生成関数は, k 個の異なる 2 値ベクトルに対して, 1 から k までのアドレス (インデックス) を生成する. アドレス生成関数の出力値を 2 進数で表現する多出力論理関数を**アドレス生成論理関数**といい, $\vec{F}$ で表わす.

表 1　アドレス表の例.

x_1	x_2	x_3	x_4	x_5	F
0	0	0	1	0	1
0	0	1	0	1	2
0	1	0	0	0	3
0	1	1	0	0	4
0	1	1	1	0	5
0	1	1	1	1	6
1	1	0	0	1	7

x1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
x2	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
x3	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
x4	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
x5=0		1			3		4	5								
x5=1			2					6					7			

図 2　関数 F の分解表.

y1	0	0	0	0	1	1	1	1
y2	0	0	1	1	0	0	1	1
y3	0	1	0	1	0	1	0	1
x5=0		1		3	4	5		
x5=1			2			6	7	

図 3　関数 G の分解表.

アドレス生成 (論理) 関数は図 1 に示す CAM を用いることにより直接実現できる. この図はビット数が 5, ワード数が 4 の CAM の例である. CAM はアドレス生成関数を直接実現する. CAM には BCAM と TCAM がある. BCAM はドント・ケアのない 2 値ベクトルを検出し, TCAM はドント・ケアのある 3 値ベクトルを検出する. 本論文では BCAM のみ取り扱う.

［例 2.1］ 表 1 に, 入力数 $n = 5$, 重み $k = 7$ のアドレス表を示す. 表 1 に示すアドレス生成関数は, ビット数 5, ワード数 7 の CAM で実現可能である. (例終)

以下ではアドレス生成関数を CAM を用いないで通常のメモリを用いて実現する方法について述べる.

［定義 2.2］ 関数 $F(\vec{X}) : B^n \to \{0, 1, \ldots k\}$, ここで $B = \{0, 1\}$, ならびに $\vec{X} = (x_1, x_2, \ldots, x_n)$ が与えられているものとする. $(\vec{X}_L, \vec{X}_H)$ を $\vec{X}$ の分割とする. F の**分解表**とは二次元のマトリックスで, 列のラベルは $\vec{X}_L$ に B の構成要素を全ての可能な組み合わせに対して割り当てたものであり, また行のラベルは $\vec{X}_H$ に B の構成要素を全ての可能な組み合わせに対して割り当てたものである. また, 対応する マトリックスの値は $F(\vec{X}_L, \vec{X}_H)$ の値に等しい. 分解表の異なる列パターンの個数を**分解表の列複雑度**という. $\vec{X}_L$ を**束縛変数**, $\vec{X}_H$ を**自由変数**という.

［補題 2.1］ 重み k のアドレス生成関数の分解表の列複雑度は高々 $k + 1$ である.

［補題 2.2］ F がアドレス生成関数のとき,

$$F(\vec{X}_1, \vec{X}_2) = G(H(\vec{X}_1), \vec{X}_2)$$

なる関係を満たす, 二つのアドレス生成関数 G と H が存在し, F の重みと G の重みは等しい.

［例 2.2］ 図 2 に示す 5 変数のアドレス生成関数 $F(\vec{X})$ の分解表を考える. いま, 関数 $F(\vec{X})$ を $F(\vec{X}_1, \vec{X}_2) = G(\vec{H}(\vec{X}_1), \vec{X}_2)$, ここで, $\vec{X}_1 = (x_1, x_2, x_3, x_4)$, $\vec{X}_2 = (x_5)$ と分解する. このとき, 分解表 (図 2) の列複雑度は 7 である. $\vec{H}$ は, 表 2 に示す 4 入力 3 出力関数であり, 重み 6 のアドレス生成論理関数となっている. また, 関数 G の分解表を図 3 に示す. このように, 重み 7 のアドレス生成関数 F を分解して得られる関数 G もまた, 重み 7 のアドレス生成関数となる. (例終)

表 2　関数 $\vec{H}$ の真理値表.

x_1	x_2	x_3	x_4	y_1	y_2	y_3
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	0	0
0	1	0	0	0	1	1
0	1	0	1	0	0	0
0	1	1	0	1	0	0
0	1	1	1	1	0	1
1	0	0	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	0	0	0
1	0	1	1	0	0	0
1	1	0	0	1	1	0
1	1	0	1	0	0	0
1	1	1	0	0	0	0
1	1	1	1	0	0	0

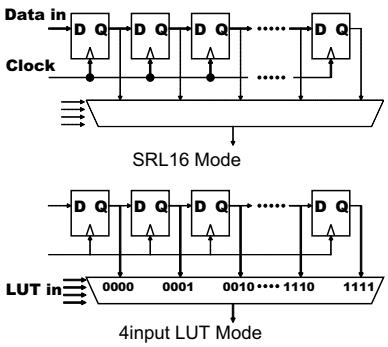

図 4　Xilinx FPGA の 4 入力 LUT.

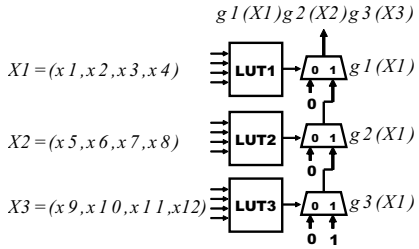

図 5　4 入力 LUT を用いた 12bit の登録ベクトル検出回路.

3. アドレス生成関数の実現

本章では Xilinx 社の FPGA 上にアドレス生成回路を実現する手法について述べる. なお, k を登録可能な登録ベクトル数, n を登録ベクトルのビット数とする.

3.1 Xilinx 社の 4 入力 LUT を用いた実現

図 4 に Xilinx 社の FPGA の 4 入力 LUT を示す. Xilinx 社の LUT ではモードを切り替えることで, シフトレジスタとして使用でき, FPGA の動作中に書き換え可能である. 図 5 に 4 入力 LUT を 3 個用いた 12bit の登録ベクトル検出回路を示す. 1 つの 4 入力 LUT は 4bit の登録ベクトルを実現する. 図 5 に示すように FPGA 内部のマルチプレクサと組合わせることで, 登録ベクトルの bit 数を拡張できる. 4 入力 LUT 回路に登録ベクトルの値を書き込むことにより, 1 つの回路で 1 つの登録ベクトルを記憶検出する回路を実現できる. この回路を登録ベクトル数だけ用意し, エンコーダを付加することにより, アドレス生成回路を実現できる. このようにして構成した回路を **4-LUT を用いた実現法という**. この方法では, 1 つの登録ベクトルに対して 4 入力 LUT が $\lceil \frac{n}{4} \rceil$ 個必要であり, k 個の登録ベクトルに対しては 4 入力 LUT が $k\lceil \frac{n}{4} \rceil$ 個必要である. また, エンコーダを実現する 4 入力 LUT の個数は $\lceil \frac{k-2}{6} \rceil \lceil \log_2(k+1) \rceil$ で十分である.

3.2 Xilinx 社の CAM の IP を用いた実現

図 6(b) に BRAM を用いたアドレス生成回路の実現法を示す. この実現法では, RAM のアドレスは登録ベクトルに割り当てたインデックスを表す. RAM を通常のメモリとして使用す

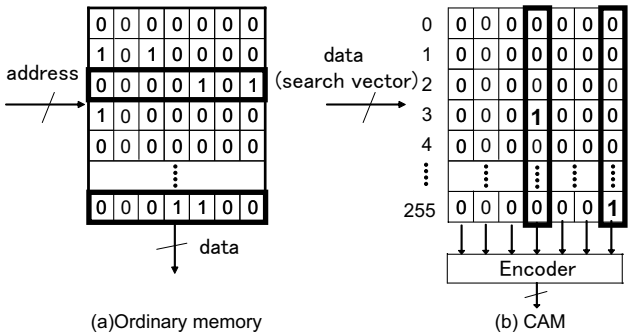

図 6　BRAM を用いたアドレス生成回路の実現.

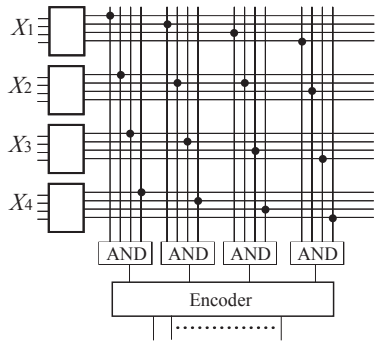

図 7　BRAM+CLB を用いた実現法.

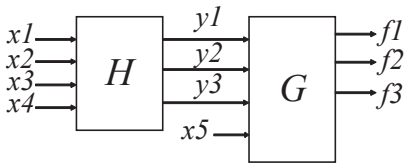

図 8　アドレス生成論理関数 F の実現.

る場合は, データは RAM のデータ配列に横方向に書き込む (図 6(a)). 一方, RAM をアドレス生成回路 (CAM) として使用する場合は, 1-hot 符号化した登録ベクトルを RAM のデータ配列に縦方向に書き込む. 検索ベクトルを RAM のアドレスに入力した場合, そのベクトルが登録されていれば非零のデータが読み出される. BRAM の出力部に FPGA の CLB を用いたエンコーダを付加することで, アドレス生成回路を実現できる. この実現法を以下では, **BRAM+CLB を用いた実現法**と呼ぶ. 複数の RAM を使用することにより登録ベクトル数を増やすことができる. また, 図 7 に示すように, 複数の RAM 出力の AND を取ることにより, 登録ベクトルのビット数を拡張できる. Xilinx 社のデュアルポート BRAM(1 個のメモリ容量が 18Kbit) の場合は, 1 個の BRAM で 16 word × 8 bit のアドレス生成回路を 2 個実現可能である. 従って, 必要な BRAM 数は $\frac{1}{2}\lceil\frac{k}{16}\rceil\lceil\frac{n}{8}\rceil$ である.

[例 3.3]　図 6(b) の場合, 入力ベクトルが 00000011 の場合, 第 4 列目の出力が 1 となる. また, 入力ベクトルが 11111111 の場合, 右端の出力が 1 となる.　(例終)

3.3　LUT カスケードを用いた実現

[定義 3.3]　***pq* 素子**とは, 任意の p 入力 q 出力論理関数を実現するメモリであり，その**メモリ量**は, $2^p \cdot q$ である.

[定理 3.1]　重み k のアドレス生成論理関数は, pq 素子を $\lceil\frac{n-q}{p-q}\rceil$ 個用いた LUT カスケードで実現可能である．ここで, $p > q$, $q = \lceil\log_2(k+1)\rceil$ である.

[例 3.4]　表 1 のアドレス表の要素数は，$k = 7$ である．従って，$q = \lceil\log_2(k+1)\rceil = \lceil\log_2(7+1)\rceil = 3$ となり，図 8 に示すように，4 入力 3 出力素子で実現可能である.　(例終)

定理 3.1 は，関数分解を繰り返すことにより，アドレス生成回路を pq 素子のカスケードとして合成可能なことを示している.

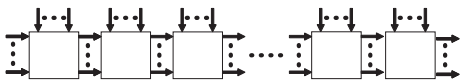

図 9　アドレス生成論理回路のカスケード実現.

表 3　登録ベクトル除去後の $\vec{H}$ の真理値表.

x_1	x_2	x_3	x_4	y_1	y_2	y_3
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	0	0
0	1	0	0	**0**	**0**	**0**
0	1	0	1	0	0	0
0	1	1	0	1	0	0
0	1	1	1	1	0	1
1	0	0	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	0	0	0
1	0	1	1	0	0	0
1	1	0	0	1	1	0
1	1	0	1	0	0	0
1	1	1	0	0	0	0
1	1	1	1	0	0	0

4.　登録ベクトルの追加と削除

通常の CAM の場合, 登録ベクトルの追加や削除は CAM の 1 ワードの追加や削除に対応し, 高速 (一定時間内) に実行可能である. また, 図 5 のような 4-LUT を用いた CAM や, 図 7 のような BRAM+CLB を用いた CAM の場合, 登録ベクトルの追加や削除は n に比例する時間内に実行可能である. 一方, LUT カスケードの場合は, 情報が複数の LUT に分散するため, 高速に登録ベクトルを変更可能か否かは自明ではない. 以下では LUT カスケードの場合も, 高速に登録ベクトルを変更可能であることを示す. アドレス表の修正は, 登録ベクトルの追加と削除という 2 つの基本操作に分解可能である. 本章では, 各手法の登録ベクトルの追加と削除について述べる.

4.1　LUT カスケードの場合

LUT カスケードは, 関数分解を繰り返して実現したものである. 従って, アドレス生成関数の修正は, アドレス表を変化させた際の, 関数分解 $F(X_1, X_2) = G(H(X_1), X_2)$ における関数 H と関数 G の修正法を考えれば十分である.

4.1.1　登録ベクトルの除去

アドレス $(\vec{a}, \vec{b})$ に対する F の出力値を 0 にする（除去する）場合, 関数分解 $F(X_1, X_2) = G(H(X_1), X_2)$ において以下の操作を行う.

1. $G(H(\vec{a}), \vec{b})$ を 0 とする.
2. $G(H(\vec{a}), X_2) \neq 0$ となる X_2 が 1 個しか存在しない場合, $H(\vec{a})$ を 0 とする.

[例 4.5]　表 1 のアドレス表が, 図 8 の回路で実現されているとする. まず, 第 3 番目の登録ベクトル $(x_1, x_2, x_3, x_4, x_5) = (0, 1, 0, 0, 0)$ を除去する場合を考えよう. この入力に対応する関数 $\vec{H}$ の出力は, $(y_1, y_2, y_3) = (0, 1, 1)$ である. また, 関数 G の出力は 3 である. まず, $G(0, 1, 1, 0)$ の出力値を 0 とする. 次に, $G(H(0, 1, 0, 0), X_2) \neq 0$ となる入力の組み合わせは, 一つしかないので, $\vec{H}(0, 1, 0, 0)$ の出力値を 0 とする.

次に, 第 5 番目の登録ベクトル $(x_1, x_2, x_3, x_4, x_5) = (0, 1, 1, 1, 0)$ を除去する場合を考えよう. この入力に対応する関数 $\vec{H}$ の出力は, $(y_1, y_2, y_3) = (1, 0, 1)$ である. また, 関数 G の出力は 5 である. まず, $G(1, 0, 1, 0)$ の出力値を 0 とする. 次に, $G(H(0, 1, 1, 1), X_2) \neq 0$ となる入力の組み合わせは, 二つ存在するので, $\vec{H}(0, 1, 1, 1)$ の出力値は変化させない. 二つの登録ベクトルを除去した後の関数 $\vec{H}$ と, 関数 G を, 表 3 と図 10 に示す.

(例終)

4.1.2　登録ベクトルの追加

アドレス $(\vec{a}, \vec{b})$ における F の出力値を c とする登録ベクトルを追加する場合, 関数分解 $F(X_1, X_2) = G(H(X_1), X_2)$ において以下の操作を行う.

1. $H(\vec{a}) \neq 0$ のとき, $G(H(\vec{a}), \vec{b})$ を c とする.

y1	0	0	0	0	1	1	1	1
y2	0	0	1	1	0	0	1	1
y3	0	1	0	1	0	1	0	1

x5=0		1			4			
x5=1			2			6	7	

図 10　登録ベクトル除去後の関数 G の分解表.

表 4　登録ベクトル追加後の $\vec{H}$ の真理値表.

x_1	x_2	x_3	x_4	y_1	y_2	y_3
0	0	0	0	**1**	**1**	**1**
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	0	0
0	1	0	0	0	1	1
0	1	0	1	0	0	0
0	1	1	0	1	0	0
0	1	1	1	1	0	1
1	0	0	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	0	0	0
1	0	1	1	0	0	0
1	1	0	0	1	1	0
1	1	0	1	0	0	0
1	1	1	0	0	0	0
1	1	1	1	0	0	0

y1	0	0	0	0	1	1	1	1
y2	0	0	1	1	0	0	1	1
y3	0	1	0	1	0	1	0	1

x5=0		1			4			9
x5=1		8	2			6	7	

図 11　登録ベクトル追加後の関数 G の分解表.

2.　$H(\vec{a}) = 0$ のとき, $H(\vec{a}) = e$, $G(\vec{e}, \vec{b}) = c$ とする. ここで, e は未使用の整数で値が最小のもの.

［例 4.6］　表 1 のアドレス表が, 図 8 の回路で実現されているとする. 入力ベクトル $(x_1, x_2, x_3, x_4, x_5) = (0, 0, 0, 1, 1)$ と, それに対応するインデックス値 8 を追加する場合を考えよう. この入力に対応する関数 $\vec{H}$ の出力は, $H(0, 0, 0, 1) \neq 0$ であるので, 関数 G の出力値を $G(H(0, 0, 0, 1), 1) = 8$ とする.

次に, 入力ベクトル $(x_1, x_2, x_3, x_4, x_5) = (0, 0, 0, 0, 0)$ と, それに対応するインデックス値 9 を追加する場合を考えよう. この入力に対応する関数 $\vec{H}$ の出力は, $H(0, 0, 0, 0) = 0$ であるので, $\vec{H}(0, 0, 0, 0) = (1, 1, 1)$ とし, 関数 G の出力値を $G(H(0, 0, 0, 0), 0) = 8$ とする. 二つの登録ベクトルを追加した後の関数 $\vec{H}$ と, 関数 G を, 表 4 と図 11 に示す.

(例終)

4.1.3　LUT カスケードの修正法

LUT カスケードの修正時間を短縮するため, 各セル毎に割り当て済みの**レイル・ベクトル**を保持する**参照テーブル** (ソフトウェアで実現) を用意する. 図 12 に参照テーブルの例を示す. ここで, アドレスは割当てたレイル・ベクトルを示し, 参照テーブルの値はそのレイル・ベクトルを参照する登録ベクトルの個数を示す. 登録ベクトルを新たに追加する場合は, 参照テーブルを使用し, 未参照レイル・ベクトルを探す. 登録ベクトルを削除する場合には, そのレイル・ベクトルが何回参照されているか調べ, 未参照の場合は, 割り当てたレイル・ベクトルをカスケードのセルから削除する (そのベクトルを 0 とする).

以下に LUT カスケードで実現したアドレス生成回路のベクトル追加と削除のアルゴリズムを示す.

［アルゴリズム 4.1］　(登録ベクトルの追加)

1.　ベクトルが未登録か調べる. その入力に対する LUT カスケードの出力が非零の場合は登録済みなので, 終了.

2.　各セルに対して入力を加え, セルの内容を読み出し, 以下の操作を行う.

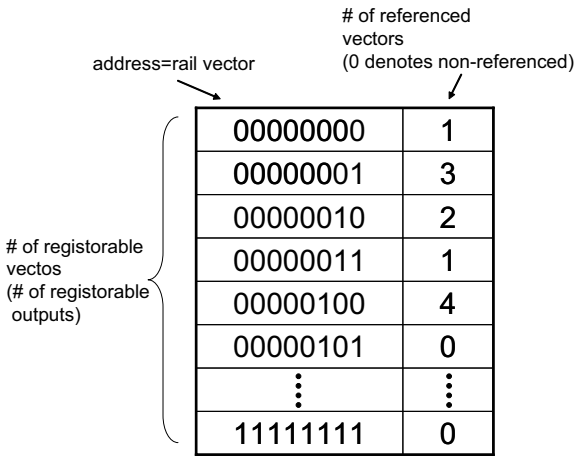

図 12　参照テーブル.

2.1.　セルの出力が 0 ならば, 参照テーブルを走査し, 未参照のレイル・ベクトルを探す (手数は高々登録ベクトル数). セルに割り当てたレイル・ベクトルを書き込む.

2.2.　登録ベクトルに対する参照テーブルの値を+1 する.

3.　終了.

［アルゴリズム 4.2］　(登録ベクトルの削除)

1.　ベクトルが登録済みか調べる. LUT カスケードの出力が 0 の場合は未登録なので, 終了.

2.　各セルに対して入力を加え, セルの内容を読み出し, 以下の操作を行う.

2.1.　読み出した値 (レイル・ベクトル) に対応する参照テーブルの値を-1 する.

2.2.　参照テーブルの値が 0 ならば, セルに 0 を書き込む.

3.　終了.

LUT カスケードに登録ベクトルを 1 つ追加するのに必要なステップ数は以下の式で見積もることができる.

$$Add_{cas} = Op.Cas + s(Op.Cas + k \times Acc.Mem + Acc.Mem) \quad (1)$$

ここで, $Op.Cas$ は LUT カスケードにアクセスするために必要なステップ数, $Acc.Mem$ は参照テーブルにアクセスするために必要なステップ数, s はセル数, k は登録ベクトル数を示す. 第一項はアルゴリズム 4.1-1 を行うのに必要なステップ数の見積もりを示し, 第二項はアルゴリズム 4.1-2 を行うのに必要なステップ数の見積もりを示す. 同様に, LUT カスケードから登録ベクトルを 1 つ削除するのに必要なステップ数は以下の式で見積もることができる.

$$Del_{cas} = Op.Cas + s(Op.Cas + Acc.Mem) \quad (2)$$

第一項はアルゴリズム 4.2-1 を行うのに必要なステップ数の見積もりを示し, 第二項はアルゴリズム 4.2-2 を行うのに必要なステップ数の見積もりを示す.

式 1, 2 より, LUT カスケードの登録ベクトルの追加に要するステップ数は s と k に, 削除に要するステップ数は s にほぼ比例することがわかる.

4.2　4-LUT 及び BRAM+CLB を用いた CAM の場合

4-LUT(図 5) や BRAM+CLB を用いたアドレス生成回路 (図 6, 7) を更新する場合, ベクトルが登録済みであるか否かを記憶する**インデックス・テーブル** (ソフトウェアで実現) を用いると管理が容易となる(注1). 図 13 にインデックス・テーブルを示す.

(注1)：Xilinx 社の Core Generator で生成した CAM [16] ではインデックスの管理をユーザが行わなければならない. 未使用インデックスを出力するプライオリティ・エンコーダを CAM に付加することで実現することも可能である.

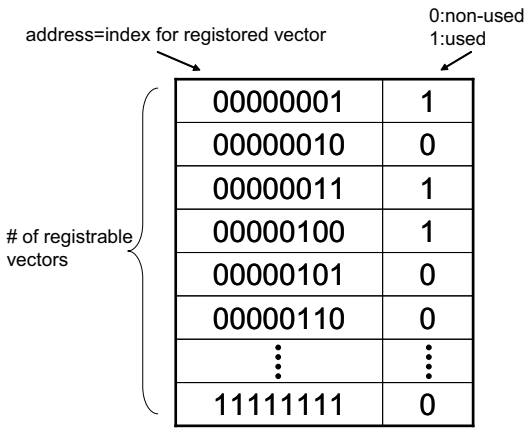

図 13　インデックス・テーブル.

インデックス・テーブルのアドレスはベクトルに割当てたインデックスに対応し, テーブルの値はそのインデックスが使用されているか否かを示す. 登録ベクトルを追加する場合は, インデックス・テーブルを参照し, 未使用であれば追加する. 登録ベクトルを削除する場合は, CAM を動作させ登録済みであるか否か調べ, 登録されていればインデックス・テーブルの値を未使用にし, CAM から登録ベクトルを削除する.

以下に, 4-LUT や BRAM+CLB で実現したアドレス生成回路の登録ベクトルの追加と削除のアルゴリズムを示す.

［アルゴリズム 4.3］ (登録ベクトルの追加)

1. ベクトルが未登録か調べる. 出力が非零の場合は登録済みなので, 終了.
2. インデックス・テーブルを走査し, 未使用インデックスを探す.
3. インデックス・テーブルに 1 を書き込む. アドレス生成回路に登録ベクトルを書き込む.
4. 終了.

［アルゴリズム 4.4］ (登録ベクトルの削除)

1. ベクトルが登録済みか調べる. 出力が 0 の場合は未登録なので, 終了.
2. 登録ベクトルを削除する. また, 対応するインデックス・テーブルの値を 0 とする.
3. 終了.

CAM に登録ベクトルを 1 つ追加するのに必要なステップ数は以下の式で見積もることができる.

$$Add_{CAM} = Op.CAM + (Op.CAM + k \times Acc.Mem + Acc.Mem) \quad (3)$$

ここで, $Op.CAM$ は CAM にアクセスするために必要なステップ数を示す. 第一項はアルゴリズム 4.3-1 を行うのに必要なステップ数の見積もりを示し, 第二項はアルゴリズム 4.3-2 を行うのに必要なステップ数の見積もりを示す. 同様に, CAM から登録ベクトルを 1 つ削除するのに必要なステップ数は以下の式で見積もることができる.

$$Del_{CAM} = Op.CAM + (Op.CAM + Acc.Mem) \quad (4)$$

第一項はアルゴリズム 4.4-1 を行うのに必要なステップ数の見積もりを示し, 第二項はアルゴリズム 4.4-2 を行うのに必要なステップ数の見積もりを示す.

式 3, 4 より, CAM の登録ベクトルの追加に要するステップ数は k に, 削除に要するステップ数は一定数であることがわかる. これらの見積もりと LUT カスケードの修正に必要なステップ数の見積もりより, $Op.Cas$, $Op.CAM$, $Acc.Mem$ が同じであると仮定すると, LUT カスケードの修正に必要なステップ数は 4-LUT や BRAM+CLB で実現した CAM を修正するのに必要なステップ数より s 倍多いことがわかる.

しかし, 多くのアプリケーションは CAM とソフトウェアを用いる場合が多いので, 本論文ではインデックスをソフトウェアで管理する部分を実現した.

表 5　実装に用いた FPGA と合成ツール.

FPGA device: Xilinx Spartan III	
Device type:	XC3S256FG
Number of Slices:	1920
(4-LUTs):	3840
(Slice Flip-Flops):	3840
I/O pins:	173
Number of Embeded Multipliers :	12
合成ツール: Xilinx, ISE Web Pack 7.2i	

5. 実験結果

5.1 ハードウェア量の比較

いくつかの k と n の値に対して, Xilinx 社の FPGA 上にアドレス生成回路を実装した. 実装に使用した FPGA と合成ツールを表 5 に示す. また, 結果を表 6 に示す. Xilinx 社の BRAM+CLB を用いた CAM は Core Generator にパラメータを与えて生成した. 従って, 回路のパイプライン化は考慮されてない. 4-LUT を用いた CAM も Core Generator で生成可能であるが, 本実験ではエンコーダ部をパイプライン化するため手設計の回路を用いた. 4-LUT を用いた場合, CAM の 1 ワードの書き換えに 16 clock 必要であり, BRAM+CLB を用いた場合, CAM の 1 ワードの書き換えに 2 clock 必要である. 一方, LUT カスケードを用いた場合, 各 BRAM の書き換えは 1 clock で行える. これらの書き換え部もハードウェアで実現した. 後述するが, どの手法でも書き換えに必要なクロック数は, 書き換えデータを生成するクロック数よりはるかに少ない. 表 6 において, *4-LUT* は 4-LUT を用いた場合 (図 5) のハードウェア量, *BRAM+CLB* は Xilinx Block RAM と CLB を用いた場合 (図 7) のハードウェア量, *Cascade* は Xilinx Block RAM を用いて LUT カスケードを構成した場合 (図 9) のハードウェア量を示す. また, 面積比較を行うために, 1 つの 4-LUT が BRAM96bit に相当するものとし [14], **正規化面積**を求めた. 正規化面積を求める式を以下に示す.

$$\text{正規化面積} = \text{Slice 数} \times 2 + \text{BRAM 数} \times 192 \quad (5)$$

a) LUT カスケードと 4-LUT の比較

LUT カスケードによる実現は 4-LUT による実現の 16%〜27%の面積でアドレス生成回路を実現できた. また, 動作周波数は約 3 倍高速にできた. これは, LUT カスケードのほうが必要面積が少なく, 配置配線の自由度が高かったからであると考えられる. また, 4-LUT による実現ではエンコーダが必要であるのに対して, LUT カスケードによる実現では BRAM だけで実現でき, クリティカルパスを短くできたことも動作周波数を向上できた原因と考えられる.

b) LUT カスケードと BRAM+CLB の比較

LUT カスケードによる実現では BRAM+CLB による実現の 8%〜11%の面積でアドレス生成回路を実現できた. 動作周波数は約 4〜5 倍向上できた. これは, 4-LUT 実現の場合と同じく, 配置配線の自由度が高かったためと, エンコーダが不要であることが原因と考えられる. ただし, 4-LUT 実現と LUT カスケード実現ではパイプラインを用いているのに対し, BRAM+CLB 実現ではパイプラインを用いていない(注2). パイプライン化を行った回路に対してはさらに実験が必要である.

5.2 更新プログラムの実行コードサイズの比較

Xilinx FPGA 上に組込プロセッサ Micro Blaze を実現し, 登録ベクトル更新回路を実装した. Micro Blaze は I/O とタイマの最小構成で実装した. プロセッサに割り当てたメモリ量は 8 KByte である. また, 単一アプリケーションを動作させるため, OS は Stand Alone を採用した. 各手法の更新アルゴリズムを C 言語で実装し, 必要メモリ量を求めた結果を表 7 に示す. ただし, ワークサイズ (スタック+ヒープ) は余ったメモリ

(注2)：実験では Xilinx 社の IP コアを使用したため回路を変更できず, パイプラインを実装できなかった.

表 6　ハードウェア量と速度の比較.

	$k=255, n=32$			$k=255, n=40$			$k=511, n=32$		
	4-LUT	BRAM +CLB	Cascade	4-LUT	BRAM +CLB	Cascade	4-LUT	BRAM +CLB	Cascade
Slice 数 (4 入力 LUT 数)	2495 (4226)	1026 (1894)	70 (0)	2990 (5127)	1204 (2298)	128 (0)	4984 (8456)	2014 (3688)	110 (0)
Block RAM 数 (セル数)	— (—)	32 (—)	4 (8)	— (—)	40 (—)	6 (11)	— (—)	64 (—)	6 (12)
正規化面積	4226	8038	908	5127	12807	1408	8456	15976	1372
最大動作周波数 [MHz] (遅延時間 [ns])	55.16 (18.12)	46.56 (21.47)	188.75 (5.29)	76.44 (13.08)	44.85 (22.29)	233.91 (4.27)	52.67 (18.98)	42.11 (23.74)	172.95 (5.78)

表 7　CAM を管理するためのソフトウェア量の比較.

ライブラリと Stand Alone 用 OS	11806 [Byte]
4-LUT 又は BRAM を更新するコード	892[Byte]
LUT カスケードを更新するコード	2293[Byte]

表 8　CAM 更新のための必要クロック数の比較.

	追加	削除
4-LUT 又は BRAM+CLB	1590	1184
LUT カスケード	12698	6393
比率 (4-LUT 又は BRAM+CLB/カスケード)	7.98	5.39

を割り当てたため, 表 7 には載せていない. 表 7 に示すように, LUT カスケード更新用コードは 4-LUT(BRAM+CAM) 更新用のコードより大きかったが, ライブラリと OS の必要サイズの割合に比べると十分小さいため無視できる.

5.3　更新プログラムの実行時間の比較

Micro Blaze にクロック計数回路を付加し, 各手法の登録ベクトルの追加と削除に必要なクロック数を求めた. 結果を表 8 に示す. なお, $s=8$, $k=255$, $n=32$ とし, 登録ベクトル 255 個を更新した. 表 8 より, LUT カスケードにベクトルを追加するのに必要なクロック数は, 計算量の見積もり通りセル数倍のクロック数が必要であった. 一方, 登録ベクトルの削除は計算量の見積もりよりも少ない 5.39 倍のクロック数であった. これは, 4-LUT や BRAM+CLB による実現では登録ベクトルテーブルの使用率が高いのに対して, LUT カスケードでは, 中間変数を共有するため参照テーブルの使用率が低くなり, 参照テーブルの読み出し時間が最悪値よりも短かったからである.

6. ま と め

アドレス生成関数を LUT カスケードを用いて FPGA 上に実現する方法を示した. アドレス生成関数を実現する部分は FPGA 上の組込メモリ (BRAM) を用いて実現した. 登録ベクトルの追加と削除は FPGA 上の組込プロセッサで行った. Xilinx 社が提供している CAM に比べ, 本手法はハードウェア量を大幅に (実験例では約 5 分の 1 に) 削減可能であることを示した. また, 回路が小型になるために, 動作速度は 4〜5 倍に改善できた. また, 登録ベクトルの追加と削除は, LUT カスケードのセル数に比例する時間で実現可能である. また, 本論文で述べた LUT カスケードによる方法では, アドレス生成関数を通常の SRAM とマイクロプロセッサを用いて実現可能であるため, CAM を用いた場合よりもシステムの消費電力を削減でき, またコストを削減できる. 欠点は従来の方法に比べアドレステーブルの更新にセル数程度 (実験例ではセル数を 8 とすると 5〜8 倍) の余分の時間がかかることである.

7. 謝　　辞

本研究は, 一部, 日本学術振興会・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] ALTERA, “Implementing high-speed search applications with Altera CAM, ” *Application Note 119*, Altera Corporation.

[2] J. Ditmar, K. Torkelsson, and A. Jantsch, “A dynamically reconfigurable FPGA-based content addressable memory for internet protocol,” *International Conference on Field Programmable Logic and Applications 2000*, (FPL2000), pp.19-28.

[3] C. H. Divine, “Memory patching circuit with increased capability,” US Patent 4028679.

[4] S. A. Guccione, D. Levi and D. Downs, “A reconfigurable content addressable memory,” In Jose Rolim et al. editors, Parallel and Distributed Processing, pp.882-889, Springer-Verlag, Berlin, May 2000. Proceedings of the 15th International Parallel and Distributed Processing Workshops, IPDPS 2000. Lecture Notes in Computer Science 1800.

[5] P. Gupta, S. Lin, and N. McKeown, “Routing lookups in hardware at memory access speeds, ” *Proc. INFOCOM*, IEEE Press, Piscataway, N.J., 1998, pp. 1240-1247.

[6] P. B. James-Roxby and D.J. Downs, “An efficient content-addressable memory implementation using dynamic routing,” *FCCM'01 2001*, pp.81-90, 2001.

[7] T. Kohonen, *Content-Addressable Memories*, Springer Series in Information Sciences, Vol. 1, Springer Berlin Heidelberg 1987.

[8] H. Liu, “Routing table compaction in ternary CAM, ” *IEEE Micro*, Vol. 22, No.1, Jan.-Feb. 2002, pp. 58-64.

[9] K. McLaughlin, N. O'Connor, and S. Sezer, “Exploring CAM design for network processing using FPGA technology,” *Proceedings of the Advanced Int'l Conference on Telecommunications and Int'l Conference on Internet and Web Applications and Services* (AICT/ICIW 2006), p.84.

[10] G. Nilsen, J. Torresen, O. Sorasen, “A variable word-width content addressable memory for fast string matching,” *Norchip Conference*, 2004

[11] K. Pagiamtzis and A. Sheikholeslami, “A Low-power content-addressable memory (CAM) using pipelined hierarchical search scheme, ” *IEEE Journal of Solid-State Circuits*, Vol. 39. No. 9, Sept. 2004, pp.1512-1519.

[12] T. Sasao, ”Design methods for multiple-valued input address generators,” *ISMVL-2006*(invited paper), Singapore, May 17-20, 2006.

[13] T. Sasao and J. T. Butler, “Implementation of multiple-valued CAM functions by LUT cascades,” *ISMVL-2006*, Singapore, May 17-20, 2006.

[14] T. Sproull, G. Brebner, and C. Neely, ”Mutable codesign for embedded protocol processing,” *International Conference on Field Programmable Logic and Applications 2005*, (FPL2005), Aug. 24-26, 2005, pp. 51- 56.

[15] J.P. Wade and C.G. Sodini, ”A ternary content addressable search engine, ” *IEEE J. Solid-State Circuits*, Vol. 24, No. 4, Aug. 1989, pp. 1003-1013.

[16] Xilinx Inc., “Content-Addressable Memory, ” *Data Sheet 253*, Nov. 2004, pp. 1-13.