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.

# ハイブリッド法を用いたアドレス生成関数の構成法と更新法について

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 $k$ までの固有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. 本稿ではハッシュ法と LUT カスケードを用いたアドレス生成関数の実現法 (ハイブリッド法) について述べる. ハイブリッド法を用いた回路のハードウェア量を示す. また, 登録ベクトルを更新する方法についても述べる. 提案手法を FPGA 上に実現し, 従来手法と比較を行った. 面積に関しては実験に用いたパラメータでは, Xilinx 社の 4 入力 LUT を用いた CAM の IP の 12%となり, Xilinx 社の BRAM を用いた CAM の IP の 8%となり, LUT カスケードのみで設計した場合の 35%となった. またハイブリッド法での登録ベクトルを更新するプログラムは多くのメモリを必要とするが, 実用可能な量であった. 本手法は従来の手法で 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, 必要なハードウェアは大幅に削減可能である.

**キーワード**　CAM, LUT カスケード

# A Method of Design and Update for An Address Generator Using a Hybrid Method

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 indices from 1 to $k$. An address generation function represents the address table. This paper presents a realization of an address generation function with a hybrid method using a hash memory and a look-up table (LUT) cascade. The amount of hardware of the hybrid method is shown. Also, an update method for registered vectors is presented. We compared three different realizations: the hybrid method, CAMs produced by the Xilinx Core Generator, and the multiple LUT cascades. Experimental results show that the area for hybrid method is only 8 to 12 % of the area for Xilinx CAMs, and is 35% of area for the multiple LUT cascades. Although our update method is complicated, the hybrid method requires smaller area and faster than conventional methods.

**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 $k$ までの固有のアドレス (インデックス) を対応させた表を, **アドレス表** [16] という. アドレス表を表現する関数をアドレス生成関数 [16] 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 [11], 辞書などに応用可能である.

アドレス表は Content Addressable Memory(CAM) [7] で直接実現可能である. CAM を実現するには特別のハードウェアが必要であるため [12], 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 [17] 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. この方法では要素数が一定値以下のアドレス表は LUT の内容を変更することにより実現でき, 回路構造は変更する必要はない. 本稿では, ハッシュ法と LUT カスケードを用いてアドレス生成回路を FPGA 上に実現する方法, 及び FPGA 上の組込みプロセッサを用いたアドレス生成回路の更新法について述べる. また Xilinx 社の提供している CAM [22] や LUT カスケードのみでアドレス生成回路を実現した場合との比較を行う. 本手法は 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.

## 2. 諸定義及び基本的性質

### 2.1 アドレス生成関数

[定義 2.1]　$B = \{0, 1\}$ とする. 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

表 1　アドレス生成関数の例.

| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $f$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 | 2 |
| 0 | 0 | 1 | 0 | 1 | 0 | 3 |
| 0 | 0 | 1 | 0 | 1 | 1 | 4 |
| 0 | 0 | 0 | 0 | 0 | 1 | 5 |
| 1 | 1 | 1 | 0 | 1 | 1 | 6 |
| 0 | 1 | 0 | 1 | 1 | 1 | 7 |

表 2　$f(X_1, X_2)$ の分解表.

| | 0 0 0 0 1 1 1 1 | $y_3$ |
|---|---|---|
| | 0 0 1 1 0 0 1 1 | $y_2$ |
| | 0 1 0 1 0 1 0 1 | $y_1$ |
| 000 | 0 0 0 0 0 0 0 0 | |
| 001 | 0 0 0 0 0 0 0 0 | |
| 010 | 1 0 2 0 3 0 0 0 | |
| 011 | 0 0 0 0 4 0 0 0 | |
| 100 | 5 0 0 0 0 0 0 0 | |
| 101 | 0 0 0 0 0 0 0 0 | |
| 110 | 0 0 0 0 0 0 0 6 | |
| 111 | 0 0 7 0 0 0 0 0 | |
| $x_6, x_5, x_4$ | | |

表 3　$\hat{f}(Y_1, X_2)$ の分解表.

| | 0 0 0 0 1 1 1 1 | $y_3$ |
|---|---|---|
| | 0 0 1 1 0 0 1 1 | $y_2$ |
| | 0 1 0 1 0 1 0 1 | $y_1$ |
| 000 | 0 0 0 0 0 0 0 0 | |
| 001 | 0 0 0 0 0 0 0 0 | |
| 010 | 2 0 1 0 0 0 3 0 | |
| 011 | 0 0 4 0 0 0 0 0 | |
| 100 | 0 5 0 0 0 0 0 0 | |
| 101 | 0 0 0 0 0 0 0 0 | |
| 110 | 0 0 0 0 6 0 0 0 | |
| 111 | 0 0 0 0 0 7 0 0 | |
| $x_6, x_5, x_4$ | | |

表 4　$\hat{f}_1(Y_1, X_2)$ の分解表.

| | 0 0 0 0 1 1 1 1 | $y_3$ |
|---|---|---|
| | 0 0 1 1 0 0 1 1 | $y_2$ |
| | 0 1 0 1 0 1 0 1 | $y_1$ |
| 000 | 0 0 0 0 0 0 0 0 | |
| 001 | 0 0 0 0 0 0 0 0 | |
| 010 | 2 0 1 0 0 0 3 0 | |
| 011 | 0 0 0 0 0 0 0 0 | |
| 100 | 0 5 0 0 0 0 0 0 | |
| 101 | 0 0 0 0 0 0 0 0 | |
| 110 | 0 0 0 0 6 0 0 0 | |
| 111 | 0 0 0 0 0 7 0 0 | |
| $x_6, x_5, x_4$ | | |

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.

[例 2.1]　表 1 に $n = 6, k = 7$ のアドレス生成関数 $f$ の例を示す. (例終)

### 2.2　ハッシュ法を用いたアドレス生成関数の実現

$f(X_1, X_2)$ を重み $k$ のアドレス生成関数とする. $n$ 入力アドレス生成関数を直接メモリで実現する場合, メモリ量が $2^n\lceil\log_2(k+1)\rceil$ 必要であり, $n$ が大きいと実用的ではない. $X_1 = (x_1, x_2, ..., x_p)$ を $Y_1 = (y_1, y_2, ..., y_p)$ に置き換えた関数を $\hat{f}(Y_1, X_2)$ とする. ただし, $y_i = x_i \oplus x_j$, $x_j \in \{X_2\}$, $p \geqq \lceil\log_2(k+1)\rceil$ である.

[例 2.2]　例 2.1 に示したアドレス生成関数の**分解表**を表 2 に示す. 列ラベルは $X_1 = (x_1, x_2, x_3)$ を表し, 行ラベルは $X_2 = (x_4, x_5, x_6)$ を表す. 表の値は関数値を表す. $Y_1 = (x_1 \oplus x_6, x_2 \oplus x_5, x_3 \oplus x_4)$ と変数変換を行った場合の $\hat{f}(Y_1, X_2)$ の分解表を表 3 に示す. 列ラベルは $Y_1$ を示し, 行ラベルは $X_2$ を示す. $f$ の分解表では非零要素を 2 つ以上持つ列が 3 つであるのに対し, $\hat{f}$ では非零要素を 2 つ以上持つ列が 1 つに減少している. (例終)

表 3 において, 列 010 の要素 4 を別の回路で実現すれば, この要素は $\hat{f}$ から削除できる. $\hat{f}$ から要素 4 を削除した関数を $\hat{f}_1$ とする. 表 4 に $\hat{f}_1$ の分解表を示す. $\hat{f}_1$ の各列には非零要素が高々1 つしか存在しない. よって $\hat{f}_1$ は $Y_1$ のみを入力とした**ハッシュメモリ**で実現できる. 表 5 に $\hat{f}_1$ のハッシュメモリを示す. ハッシュメモリは $2^p$ の集合を $2^p$ の集合へ写す写像を表現できる. ハッシュメモリは $\hat{f}_1$ の出力値を与えるが, この値は必ずしも $f$ の値と等しいとは限らない. ハッシュメモリの出力が非零の場合でも, $X_2$ の値を調べなければ, $\hat{f}_1$ の値が正しい値か否かわからない. $\hat{f}_1$ の値が零の場合は, $f$ の値も零である. そこで**補助メモリ**を付加し, 補助メモリにハッシュメモリに登録したベクトルに対応する $X_2$ を登録する. そして入力 $X_2$ と比較を行い, ハッシュメモリの値の正誤判定を比較器で行う. 図 1 に**ハッシュメモリを用いた実現法 (ハッシュ法)** を示す. 入力 $\{X_1, X_2\}$ からハッシュ入力 $Y_1$ を生成する. ここで, $|X_1| = |Y_1|$ である. $Y_1$ を用いてハッシュメモリを参照し出力 $q$ を得る. $q$ を用いて補助メモリを参照し出力 $X_2'$ を得る. $X_2'$ と入力 $X_2$ を比較し, 一致すれば $q$ を出力する. 不一致の場合は, 0 ベクトルを出力する.

表 5　$\hat{f}_1(Y_1)$ のハッシュメモリ.

| $y_3$ | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|
| $y_2$ | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| $y_1$ | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| $\hat{f}_1$ | 2 | 5 | 1 | 0 | 6 | 7 | 3 | 0 |

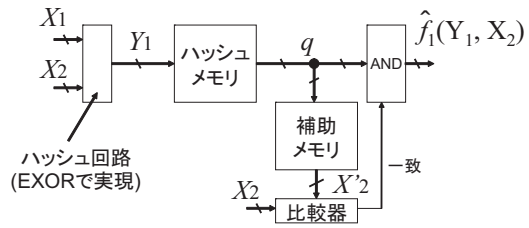


図 1　ハッシュメモリを用いた実現法 (ハッシュ法).

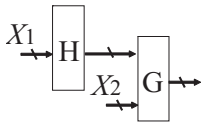


図 2　関数分解.

[定理 2.1]　[18] 重みが $k$ の $n$ 入力アドレス生成関数において, ハッシュメモリで実現される登録ベクトルの割合は

$$\delta \simeq 1 - \frac{1}{2}(\frac{k}{2^p}) + \frac{1}{6}(\frac{k}{2^p})^2 \qquad (1)$$

である. ただし, ハッシュメモリの入力数を $p$ とすると $k \leqq 2^p$ であり, アドレス生成関数の分解表において, 非零要素は一様に分布しているものとする.

$\frac{k}{2^p} = \frac{1}{4}$ とすると $\delta \simeq 0.8854$ であり, $\frac{k}{2^p} = \frac{1}{2}$ とすると $\delta \simeq 0.792$ である. よって, 登録ベクトル数の約 4 倍のアドレス空間をもつハッシュメモリを用意すれば登録ベクトルの約 90%をハッシュメモリで実現できる.

[定理 2.2]　[19] $Y_1 = (y_1, y_2, ..., y_p)$(ただし $y_i = x_i \oplus x_j, x_j \in \{X_2\}$) をハッシュメモリの入力とした場合, 図 1 の回路ではハッシュメモリの正誤判定のため, 補助メモリと比較器が必要である. 補助メモリは $X_2$ のみ出力すればよい.

### 2.3　LUT カスケードを用いたアドレス生成関数の実現

アドレス生成関数の入力数を $n$, 重みを $k$ とする. $n$ が大きいとき, アドレス生成関数を単一メモリで実現するのは実用的でない. $k << 2^n$ のとき, **LUT カスケード**を用いることで必要メモリ量を大幅に削減できる.

[定義 2.2]　[14] 分解表における異なる列パターンの個数を**列複雑度**という.

[定理 2.3]　[14] 与えられた関数 $f$ の入力の分割を $X = \{X_1, X_2\}$ とする. $X_1$ を分解表の列に割り当て, $X_2$ を行に割り当てる. $\mu$ を列複雑度とすると, 関数 $f$ は図 2 に示す回路で実現できる. この場合, $H$ と $G$ 間の接続線数は $\lceil\log_2 \mu\rceil$ である.

$\lceil\log_2 \mu\rceil < |X_1|$ のとき関数 $f$ を実現するメモリ量を削減できる. 図 2 に示す分解は**関数分解**と呼ばれ, LUT カスケードは関数 $f$ に対し, 関数分解を繰り返し適用することで得られる. 図 3 に LUT カスケードを示す. LUT カスケードにおいて, 各メモリを**セル**と呼び, セル間の接続線を**レール**と呼ぶ.

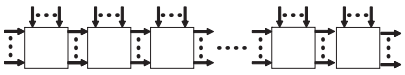

図 3　LUT カスケード.

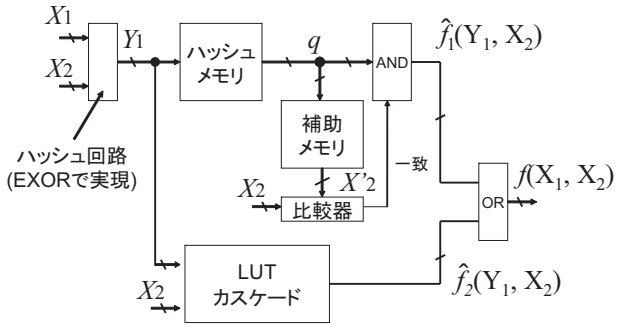


図 4　ハイブリッド法 [18].

[定理 2.4]　[18] 分解表の列複雑度が高々$\mu$ の関数 $f$ は $r+1$ 入力 $r$ 出力のセルを持つ LUT カスケードで実現可能である. ただし, $r = \lceil \log_2 \mu \rceil$ である.
[定理 2.5]　[15] $n$ を外部入力数, $r$ をレール数, $p$ をセル入力数とする. セル数 $s$ は $s \leqq \lceil \frac{n-r}{p-r} \rceil$ である. ただし, $p > r$ である.
[定理 2.6]　[17] 重み $k$ のアドレス生成関数の列複雑度は高々 $k+1$ である.

上記の定理より, 重みを決めれば LUT カスケードに必要なメモリ量が決まる. しかしながら, $k$ が大きい場合, LUT カスケードで実現しても LUT カスケードが大きくなりがちである. 出力関数を分割して別々の LUT カスケードで実現することで総メモリ量を減らすことができる [13]. この方法では, 各カスケードの出力の結合部に特殊なエンコーダを必要とする.

## 3. ハイブリッド法を用いたアドレス生成関数の実現

### 3.1 回路の構成

図 4 に**ハイブリッド法**を用いたアドレス生成関数を実現する回路 [18] を示す. アドレス生成関数を $f(X_1, X_2)$ とし, 分解 $f(X_1, X_2) = \hat{f}_1(Y_1, X_2) \vee f_2(Y_1, X_2)$ を考える. ハッシュ法で $\hat{f}_1(Y_1, X_2)$ を実現する. LUT カスケードで $\hat{f}_2(Y_1, X_2)$ を実現する. LUT カスケードの入力は $\{X_1, X_2\}$ でもよいが, 後述する登録ベクトルの削除を高速に行うため, $X_1$ ではなく $Y_1$ を用いる. 入力を変えることで $f_2$ が $\hat{f}_2$ に変わるが, $\hat{f}_2$ もアドレス生成関数である. $\hat{f}_2$ と $\hat{f}_2$ が格納する登録ベクトル数は変わらないので, 定理 2.6 より最大レール数も変わらない. 従って, $X_1$ を $Y_1$ に置き換えても同量の LUT カスケードで実現可能である. 本手法は登録ベクトルの更新を行うために, マイクロプロセッサを付加する.

### 3.2 ハードウェア量の見積り

重み $k$ の $n$ 入力アドレス生成関数を実現するのに必要なメモリ量を求める. $p$ をハッシュメモリの入力数とする. $c$ を LUT カスケードのセルの入力数, $s$ を LUT カスケードのセル数とする. 定理 2.1 より, ハッシュメモリで実現される登録ベクトルの割合は $\delta$ であり LUT カスケードで実現される登録ベクトルの割合は $1-\delta$ である. また, ハッシュメモリのメモリ量 $M_{hash}$ は

$$M_{hash} = 2^p \lceil \log_2(k+1) \rceil \tag{2}$$

であり, 補助メモリのメモリ量 $M_{aux}$ は

$$\begin{aligned} M_{aux} &= 2^{\lceil \log_2(k+1) \rceil}(n-p) \\ &= (k+1)(n-p) \end{aligned} \tag{3}$$

であり, LUT カスケードのメモリ用 $M_{cas}$ は

$$M_{cas} = 2^c k(1-\delta)s \tag{4}$$

である. ここでセル数 $s$ は定理 2.1, 2.5 および 2.6 より,

$$s = \left\lceil \frac{n - \lceil \log_2 k(1-\delta) \rceil}{c - \lceil \log_2 k(1-\delta) \rceil} \right\rceil \tag{5}$$

である. ハイブリッド法を実現するときは, これらの式を元に評価関数を作成し, 設計を行うとよい. 例えば, 面積を最小にする場合は総メモリ量 $M_{total} = M_{hash} + M_{aux} + M_{cas}$ を最小にするように $p$, $c$ を決めればよい. 遅延時間に関しては, 多くの場合 LUT カスケードがクリティカルパスになるのでセル数 $s$ が制約面積 $M_{total}$ 下の中で最小になるようにパラメータを決めるとよい. ハッシュ関数の生成法は文献 [18], [19] で述べられている.

## 4. 登録ベクトルの更新法

通常の CAM [12] 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, ソーティングのための時間を無視すると一定時間内に実行可能である. FPGA 上に実現した 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 カスケードは情報が複数のセルに分散しているため, 高速に登録ベクトルを変更可能か否かは自明ではなかった. 文献 [10] では LUT カスケードの登録ベクトルの更新について述べている. 本論文では, LUT カスケード, 及びハッシュメモリや補助メモリの更新操作を追加し, ハイブリッド法の登録ベクトルの更新法を示す.

### 4.1 LUT カスケードの登録ベクトルの更新

LUT カスケードにおいて, 登録ベクトルの更新はベクトルの追加とベクトルの削除の基本操作に分解可能である. LUT カスケードは登録ベクトルの部分ベクトルを複数のセルに分散して格納する. 共通する部分ベクトルを同一の場所に格納するため, 登録ベクトルを圧縮できる. しかし, 更新の時に共有する部分ベクトルを覚えておかなければならない. 共有部を考慮せずに削除を行うと, 格納した部分ベクトルを誤って削除し, 登録ベクトルの整合がとれなくなってしまう. 各セルに割り当てた部分ベクトルを保持するために, 各セルに**参照テーブル**を用意する. 参照テーブルはソフトウェア上のメモリで実現する. 登録ベクトルが追加された時, 更新プログラムは参照テーブルに該当する部分ベクトルの登録回数を書き込む. 削除する時は, 更新プログラムが参照テーブルから該当する部分ベクトルの登録回数を 1 つ減らし, 登録されてないことが確認できれば該当する部分ベクトルをセルから削除する.

以下に LUT カスケードで実現したアドレス生成関数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]　(LUT カスケードの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.
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]　(LUT カスケードの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.

### 4.2 ハイブリッド法における登録ベクトルの更新

ハイブリッド法ではアドレス生成関数をハッシュメモリ, 補助メモリ, 及び LUT カスケードで実現する. 登録ベクトルの更新は, ハッシュメモリ, 補助メモリ, 及び LUT カスケードの更新を組合せて行う. 回路面積削減のため, 登録ベクトルの 90%をハッシュメモリで実現する. よって, 登録ベクトルの追加

表 6　$\hat{g}(Y_1, X_2)$ の分解表.

| | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | $y_3$ |
|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | $y_2$ |
| | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | $y_1$ |
| 000 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | |
| 001 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | |
| 010 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | |
| 011 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 0 | |
| 100 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 0 | |
| 101 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 0 | |
| 110 | 0 | 0 | 7 | 0 | 0 | 0 | 0 | 0 | |
| 111 | 0 | 0 | 8 | 0 | 0 | 0 | 0 | 0 | |
| $x_6, x_5, x_4$ | | | | | | | | | |

表 7　$\hat{g}'(Y_1, X_2)$ の分解表.

| | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | $y_3$ |
|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | $y_2$ |
| | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | $y_1$ |
| 000 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | |
| 001 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | |
| 010 | 0 | 0 | 3 | 0 | 9 | 0 | 0 | 0 | |
| 011 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 0 | |
| 100 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 0 | |
| 101 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 0 | |
| 110 | 0 | 0 | 7 | 0 | 0 | 0 | 0 | 0 | |
| 111 | 0 | 0 | 8 | 0 | 0 | 0 | 0 | 0 | |
| $x_6, x_5, x_4$ | | | | | | | | | |

は, まずハッシュメモリに格納できるか調べ, 格納できない場合は LUT カスケードに格納する. 以下にハイブリッド法における登録ベクトルの追加アルゴリズムを示す.

[アルゴリズム 4.3]　(ハイブリッド法における登録ベクトルの追加)

1.　外部入力 $\{X_1, X_2\}$ を与え, ベクトルが未登録か調べる. その入力に対する出力が非零の場合は登録済みなので, 終了.

2.　ハッシュメモリの入力 $Y_1$ を生成し, ハッシュメモリを読み出す.

2.1.　ハッシュメモリの出力が非零の場合は, 衝突が発生したので LUT カスケードにベクトルを追加して終了.

2.2.　ハッシュメモリの出力が零の場合は, 登録ベクトルをハッシュメモリに追加する. また, 外部入力 $X_2$ を補助メモリに登録して終了.

3.　終了.

提案回路から登録ベクトルを削除する場合, ハッシュメモリと LUT カスケードでのデータ配置の調整が必要である. 調整を行わなずにハッシュメモリからベクトルを削除すると, ベクトルの追加が不可能となる例を示す.

[例 4.3]　図 6 に, 関数 $g(x_1, x_2, x_3, x_4, x_5, x_6)$ の入力の一部を $\{y_1, y_2, y_3\}$(ただし $y_i = x_i \oplus x_{6-i}$) に変換した関数 $\hat{g}(y_1, y_2, y_3, x_4, x_5, x_6)$ の分解表を示す. 関数 $\hat{g}$ は出力 1 を持つベクトル $(0, 1, 0, 0, 0, 0)$ をハッシュメモリに格納し, 非零出力を持つ残りのベクトルを LUT カスケードに格納することで実現できる. ここでハッシュメモリから出力 1 を持つベクトル $(0, 1, 0, 0, 0, 0)$ を削除し, 新たに $g(0, 0, 1, 0, 0, 0) = 1, g(0, 0, 1, 0, 1, 0) = 9$ を追加した関数 $\hat{g}'$ を図 7 に示す. このとき, 新たに追加したベクトルが衝突するので, どちらかを LUT カスケードに格納しなければならない. しかし, LUT カスケードに登録可能なベクトルが 7 個の場合, 追加できない.　(例終)

例 4.3 では, ハッシュメモリからベクトルを削除する際, 削除するベクトルと衝突しているベクトルを LUT カスケードからハッシュメモリに移動することで解決できる. 例えば, $(0, 1, 0, 0, 0, 1) = 2$ をハッシュメモリに移せばよい. 以下にハイブリッド法における登録ベクトルの削除アルゴリズムを示す. ただし, $B = \{0, 1\}$ であり, $X_2' \in B^{n-p}$ である.

[アルゴリズム 4.4]　(ハイブリッド法における登録ベクトルの削除)

1.　外部入力 $\{X_1, X_2\}$ を与え, ベクトルが未登録か調べる. その入力に対する出力が零の場合は未登録なので, 終了.

2.　ハッシュメモリの入力 $Y_1$ を生成し, ハッシュメモリを読み出し, $\hat{f}_1(Y_1, X_2)$ を評価する.

2.1.　$\hat{f}_1(Y_1, X_2) = 0$ の場合は, LUT カスケードからベクトルを削除して終了.

2.2.　$\hat{f}_1(Y_1, X_2) \neq 0$ の場合は, ハッシュメモリからベクトルを削除する. また, 補助メモリから対応するベクトルに対する $X_2$ を削除する.

2.3.　LUT カスケードの出力が非零となる入力 $\{Y_1, X_2'\}$ を探す.

2.3.1　非零となる $\{Y_1, X_2'\}$ が見つからない場合は終了.

2.3.2　見つかった場合は $\{Y_1, X_2'\}$ を LUT カスケードから削除し, ハッシュメモリと補助メモリに $\{Y_1, X_2'\}$ を追加して終了.

3.　終了.

LUT カスケードから非零となる $\{Y_1, X_2'\}$ を探す場合, $Y_1$ は固定なので, 最悪 $2^{n-p}$ 回 LUT カスケードを調べなければならない. しかし, LUT カスケードの各セルを調べることで, 探索に必要な手数を大幅に削減できる. 以下に, 各セルを調べ非零出力を持つベクトルを探すアルゴリズムを示す.

[アルゴリズム 4.5]　(LUT カスケードから非零出力を持つベクトルの探索)

1.　$c \leftarrow 0$ とする.

2.　セル $c$ の外部入力が $Y_1$ の一部に依存する場合, 依存する外部入力と前段のレール値 $r$ を用いてセルをアクセスし, $r \leftarrow$(セルの値) として, 4 へ.

3.　セル $c$ の外部入力が $X_2'$ の一部に依存する場合, 非零となる $\{r, X'\}$ を探す. 非零となる $\{r, X'\}$ が見つからない場合は, 非零出力を持つベクトルが見つからなかったので終了. そうでなければ, 非零となる $\{r, X'\}$ を用いてセルをアクセスし, $r \leftarrow$(セルの値) とする.

4.　$c =$(総セル数) ならば終了. そうでなければ $c \leftarrow c + 1$ とし, 2 へ.

[定理 4.7]　図 4 に示したハイブリッド法でアドレス生成関数を実現する場合, $k$ 入力 $r$ 出力セルを持つ LUT カスケードから非零出力を持つベクトル $\{Y_1, X_2'\}$ を探すには高々

$$2^{k-r}\left\lceil \frac{|X_2'|}{k-r} \right\rceil \tag{6}$$

回 LUT カスケードの各セルを調べれば十分である.

**(証明)** 各セルの外部出力数は $k - r$ である. レール値 $r$ ビットは前段のセルから既に求まっているので, 非零出力を持つベクトルを探すには, 1 個のセルに関して $2^{k-r}$ 回調べれば十分である. $Y_1$ は既に定まっているので, $X_2'$ を依存変数とする $\lceil \frac{|X_2'|}{k-r} \rceil$ 個のセルに関して, 非零出力を持つかどうか調べればよい. 従って, 式 (6) を得る.　(証明終)

[例 4.4]　$|Y_1| = 12, |X_2'| = 20$ とし, $k = 11, r = 7$ としよう. $X_2'$ に値を割り当てて非零出力を持つベクトルを探す場合, 率直に行うと最悪 $2^{20} = 1048576$ 回セルを調べなければならない. アルゴリズム 4.5 を用いた場合, 定理 4.7 より高々 $2^{11-7}\lceil \frac{20}{11-7} \rceil = 80$ 回セルを調べれば十分である.　(例終)

ハッシュ法による関数生成部ではハッシュメモリと補助メモリで登録ベクトル情報を保持しているが, 参照テーブルを用意する必要はない. 従って, ハッシュ法による関数生成部のほうが LUT カスケードよりも更新時間は短い. 定理 2.1 から, ハッシュメモリの入力数 $p$ を増やすと $\delta$ が増加するので, ハッシュメモリや補助メモリに対する追加や削除の割合も増加する. よって, 登録ベクトルの更新時間は短くなる.

## 5. 実験結果

### 5.1 実験内容

$n = 32$, $k = 1000$ として Xilinx 社の FPGA 上にハイブリッド法を実装した. ハイブリッド法, 及び比較に用いた回路は以下の通りである.

a)　4 入力 LUT を用いた Xilinx の CAM IP [22]

Xilinx FPGA の 4 入力 LUT は 4bit の登録ベクトル検出回路を実現できる (図 5). さらに, CLB 内部のマルチプレクサを用いて 4 入力 LUT を接続することで, 4bit 以上の登録ベクトル検出回路を実現できる (図 6). 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で, アドレス生成回路を実現できる. また, 4 入力 LUT の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, 動作中に書き換えることができる. 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. 登録ベク

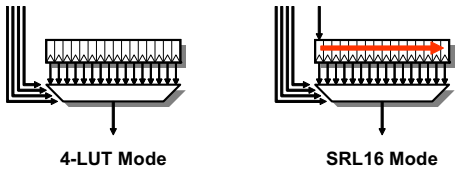


図 5　Xilinx 4 入力 LUT.

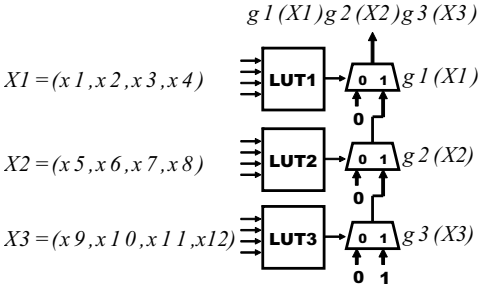


図 6　12bit 登録ベクトル検出回路.

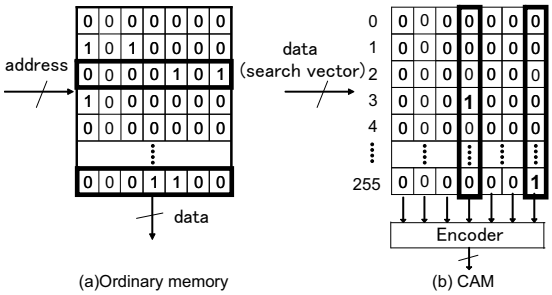


図 7　BRAM を用いた登録ベクトル検出回路.

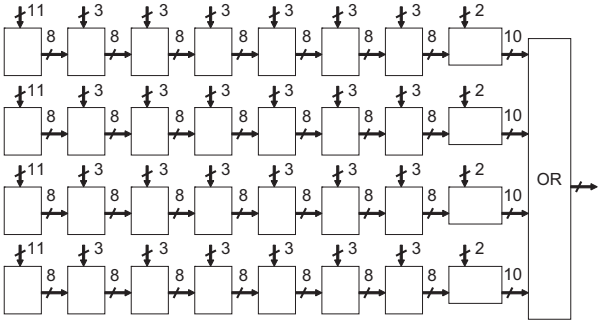


図 8　設計した並列 LUT カスケード.

トルはソフトウェアを用いて管理する.

b）BRAM を用いた Xilinx の CAM IP [22]

FPGA の組込みメモリを用いても CAM を実現できる. 図 7 に FPGA の組込メモリを用いたアドレス生成回路を示す. メモリのアドレスを登録ベクトルと一致させ, 登録ベクトルの番号をメモリに書き込むことにより登録ベクトル検出回路をメモリで実現できる. メモリの入力数を $n$, 出力数を $m$ とすると, $2^n$ ビットの登録ベクトルを $m$ 個格納できる. 4 入力 LUT を用いる方法と同様に, 登録ベクトルのビット数を拡張するには複数のメモリの出力に AND ゲートを付加すればよい. また, 多ビット出力メモリを用いて, 割り当て番号を直接書き込めばエンコーダを付加する必要はない. 4 入力 LUT の場合と同様に, 登録ベクトルはソフトウェアを用いて管理する.

c）並列 LUT カスケード

設定したアドレス生成関数を実現するために, 11 入力 8 出力セルを 8 個持つ LUT カスケードを 4 本設計した. 設計した並列 LUT カスケードを図 8 に示す. 各 LUT カスケードは登録ベクトルを 255 個格納できる. 各 LUT カスケードの出力を OR で結合して, 登録ベクトルを 1000 個格納する回路を構成した. スループットを上げるため, 各セル間にレジスタを挟みパイプライン化を実現した. 登録ベクトルの更新は第 4 章で述べた手法を用いる.

d）ハイブリッド法

設定した登録ベクトルの個数は 1000 個なので, その 9 割に当たる 900 個の登録ベクトルをハッシュメモリに格納し, 残りの 100 個を LUT カスケードに格納する回路を設計した. 定理 2.1 より, ハッシュメモリの入力数を $p = k + 2 = 12$ とし, ハッシュメモリの出力は $\lceil \log_2(1000+1) \rceil = 10$ とした. 補助メモリの入力数は 10 とし, 出力数は $(n - p) = 32 - 12 = 20$ とした. 定理 2.5, 2.6 より, LUT カスケードはレール数を $\lceil \log_2(100+1) \rceil = 7$ とし, 11 入力 7 出力のセルを 7 個用いて実現した.

実装に使用した FPGA ボードと合成ツールを表 8 に示す. ハードウェア量と読み出し速度を表 9 に示す. ハイブリッド法以外は使用した FPGA 内に格納できなかったので, 論理合成時の見積もり値を用いた. 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 [20], **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.

$$\text{正規化面積} = \text{Slice 数} \times 2 + \text{BRAM 数} \times 192 \qquad (7)$$

表 9 より, 提案手法を用いることで, 設定したパラメータに関しては Xilinx 社の 4 入力 LUT を用いた CAM の IP の 12%となり, Xilinx 社の BRAM を用いた CAM の IP の 8%となり, LUT カスケードのみで設計した場合の 35%となった.

表 8　実装に用いた FPGA ボードと合成ツール.

| | |
|---|---|
| **FPGA board: Xilinx Spartan 3E Starter Kit** | |
| Clocks<br>Memories | 50 MHz<br>128 Mbit Parallel Flash<br>16 Mbit SPI Flash<br>64 MByte DDR SDRAM |
| **FPGA device: Xilinx Spartan 3E** | |
| Device type:<br>Number of Slices:<br>I/O pins:<br>Number of BRAMs:<br>(Amount of BRAMs)<br>Number of Embeded Multipliers : | XC3S500E<br>4656<br>232<br>20<br>360(kbit)<br>20 |
| **Synthesis Tool: Xilinx, ISE 9.1i** | |

表 9　ハードウェア量と読み出し速度.

| | スライス数 | BRAM 数 | 正規化面積 | 最大動作周波数 [MHz] |
|---|---|---|---|---|
| Xilinx CAM (4-LUT Based) | 9980 | 0 | 19960 | 52.67 |
| Xilinx CAM (BRAM Based) | 4104 | 128 | 32784 | 42.11 |
| Multiple LUT Cascade | 21 | 36 | 6954 | 233.91 |
| Hybrid Method | 39 | 12 | 2382 | 211.60 |

表 10　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 Standalone 用 OS | 10506 [Byte] |
| 4-LUT 又は BRAM を更新するコード | 15577 [Byte] |
| 並列 LUT カスケードでの更新コード | 20030 [Byte] |
| ハイブリッド法での更新コード | 22739 [Byte] |

### 5.2　更新プログラムの必要メモリ量の比較

Xilinx 社の組込みプロセッサ MicroBlaze を各アドレス生成回路に付加し, 登録ベクトル更新回路を実装した. MicroBlaze は I/O とタイマ, 及び SDRAM コントローラを組込んだ. MicroBlaze のスタック領域, ヒープ領域, 実行コード領域, 及びデータ領域は 64MByte の SDRAM 上に格納した. また, 単一アプリケーションを動作させるため, OS は Standalone 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, 必要なメモリ量を求めた結果を表 10 に示す. 表 10 に示すように, ハイブリッド法での登録ベクトルを管理するためのソフトウェア量が一番多い. しかし, MicroBlaze とアドレス生成回路を単一の FPGA で実現するには FPGA に内蔵されている BRAM では足りず, ソフトウェアを外付けの SDRAM に格納する必要がある. よって, ハイブリッド法での更新法は若干メモリ量が多いものの, 外付けの SDRAM を用意すれば十分である. また実際の組込みアプリケーションでは RTOS を用いることが考えられ, この場合, これらの更新プログラムは RTOS の必要メモリ量と比較して十分に無視できるサイズである.

### 5.3　更新プログラムの実行時間の比較

各手法を用いたアドレス生成回路に MicroBlaze と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デー

表 11　アドレス生成回路の更新時間 [msec] の比較.

| | 追加 | 削除 |
|---|---|---|
| 4-LUT 又は BRAM+CLB | 1572 | 1572 |
| 並列 LUT カスケード | 2881 | 332 |
| ハイブリッド法 | 262 | 109 |

タ生成に必要なクロック数を求めた. 得られたクロック数に MicroBlaze の動作周波数 50MHz の周期である 20ns を掛け, 1000 個の登録ベクトルを更新するのに必要な時間 (msec) を求めた. 結果を表 11 に示す.

表 11 より, ハイブリッド法での更新時間は LUT カスケードの更新時間と比較して追加に関しては約 14 倍高速であり, 削除に関しては約 3 倍高速であった. 実験で設定したパラメータは LUT カスケードが $k = 1000, s = 32$ であるのに対し, ハイブリッド法での LUT カスケードは $k = 100, s = 7$ なので, LUT カスケードだけで比較を行うと約 45 倍高速である. しかし高速化が 14 倍で留まった理由は, ハッシュメモリや補助メモリの更新の際, これらの回路の評価を行わなければならなかったのが挙げられる. また, 組込みプロセッサとアドレス生成回路のハッシュメモリを直接バスで接続したため, ハッシュ回路をソフトウェアで実現する必要があった. 複雑なビット操作を行ったので, ハッシュメモリや補助メモリの更新に時間がかかった. LUT カスケードから 1 個のベクトルを削除するのは一定時間で可能である. 実装したハイブリッド法では単一の LUT カスケードを用いたのに対し, 並列 LUT カスケードは 4 本の LUT カスケードを用いている. ハッシュメモリでの登録ベクトルの削除の時間を無視すると, ハイブリッド法の削除時間は並列 LUT カスケードの削除時間の約 4 分の 1 と予想できる. しかし高速化が 3 倍で留まった理由は, 先述したハッシュ回路の評価時間とハッシュメモリの操作に関する時間が影響を及ぼしたと考えられる. ハイブリッド法での更新は, Xilinx の CAM の更新と比較して追加に関しては約 8 倍高速であり, 削除に関しては約 15 倍高速であった(注1).

## 6. ま と め

本論文では, ハッシュ法と LUT カスケードを併用したアドレス生成関数の実現法 (ハイブリッド法) について述べた. ハッシュ法と LUT カスケードで必要な回路のハードウェア量を示し, 登録ベクトルを更新する方法について述べた. 従来手法とハードウェア量の比較を行った. 実験に用いたパラメータでは, Xilinx 社の 4 入力 LUT を用いた IP より面積を 88%削減でき, Xilinx 社の BRAM を用いた IP より面積を 92%削減でき, LUT カスケードのみで設計した場合より面積を 65%削減できた. また登録ベクトルを更新するプログラムは提案手法では多くのメモリを必要とするが, 実用可能な量である. 本手法では従来の方法で 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, 必要なハードウェア量は大幅に削減可能である.

## 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・知的クラスター創成事業 (第二期) 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,” *IPDPS 2000 Workshop*, Cancun, Mexico, May 2000. IEEE Computer Society Press.

[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] H. Nakahara, T. Sasao and M. Matsuura, “A CAM emulator using look-up table cascades,” *RAW2007*, March 2007, Long Beach California, USA, CD-ROM RAW-9-paper-2.

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

[12] 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.

[13] H. Qin, T. Sasao, and J. T. Butler, “On the design of LPM address generators using multiple LUT Cascades on FPGAs,” *International Journal of Electronics*, Vol. 94, Issue 5, May 2007, pp.451-467,

[14] T.Sasao, “Logic Synthesis and Optimization,” *Kluwer Academic Publishers*, Norewll, MA, 1993.

[15] T. Sasao, M. Matsuura, and Y. Iguchi, ”A cascade realization of multiple-output function for reconfigurable hardware,” *IWLS01*, Lake Tahoe, CA, June 12-15, 2001, pp.225-230.

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

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

[18] T. Sasao, “A Design method of address generators using hash memories,” *IWLS-2006*, Vail, Colorado, U.S.A, June 7-9, 2006, pp.102-109.

[19] T. Sasao and M. Matsuura, “An implementation of an address generator using hash memories,” *DSD 2007, 10th EUROMICRO Conference on Digital System Design, Architectures, Methods and Tools*, Aug. 27 - 31, 2007, Lubeck, Germany, pp.69-76.

[20] 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.

[21] 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.

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

(注1)：Xilinx の CAM を更新するプログラムは組込み用途を目的としてコードサイズを削減する手法を採用した. 従って, 登録ベクトルが更新される度に空いている番号を見つけるため逐一メモリを参照する. 更新プログラムは単純であり必要メモリ量は少ない反面, 登録ベクトル数に比例した更新時間がかかる. しかし, RTOS を利用して, プロセッサの空き時間に未登録の番号を探索したり, CAM を模擬するデータ構造を用いることで, 更新時間を大幅に削減できると考えられる.