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, Japan

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 time that is proportional to the number of cells in the LUT cascade.

1 Introduction

An **address table** relates k different registered vectors to the addresses from 1 to k [12]. An **address generation function** represents the address table [12]. Applications of address generation functions include address lists for the Internet [5][8], memory patch circuits [3], pattern matching circuits [10], and dictionaries [12].

An address table can be directly implemented by a Content Addressable Memory (CAM) [7]. However, in this case, special hardware is needed [11]. By using ordinary memories and gates, we can implement circuits [2][6][9] that are equivalent to CAMs. In [13], we showed a realization method of CAM functions by LUT cascades. For a given size of an address table, an LUT cascade can realize an arbitrary address table by changing the personality of the LUTs without changing the architecture of the cascade. This paper shows the realization an address generation function using an LUT cascade on an FPGA, and also shows a modification method for the address table by an embedded processor on the FPGA. Compared with CAMs produced by the Xilinx Core Generator, our implementations are smaller and faster, although the time for the modification of address table is longer than conventional methods.

1-4244-0910-1/07/$20.00 ©2007 IEEE.

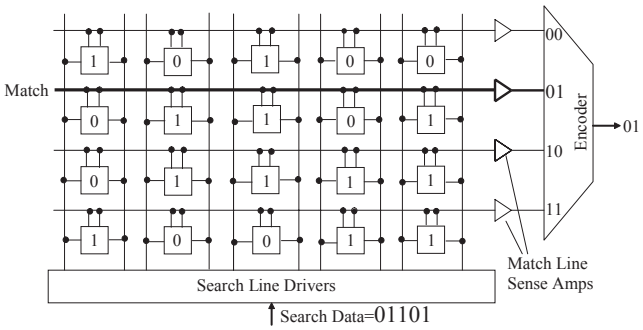

Figure 1. CAM device.

2 Definitions and Basic Properties

Definition 2.1 *A mapping $F(\vec{X}) : B^n \rightarrow \{0, 1, \ldots, k\}$, where $\vec{a}_i \in B^n$ $(i = 1, 2, \ldots, k)$, is an* **address generation function with weight k** *if $F(\vec{a}_i) = i$ $(i = 1, 2, \ldots, k)$ for k different registered vectors, and $F = 0$ for other $(2^n - k)$ input vectors. In other words, an address generation function produces addresses ranging from 1 to k for k different registered vectors and produces 0 for other vectors. The multiple-output logic function that represents the output values by binary numbers is the* **address generation logic function** *and is denoted by $\vec{F}$.*

Fig. 1 shows an example of a CAM, where the number of bits is five and the number of words, k, is four. A CAM directly realizes an address generation (logic) function. CAMs can be classified into two: BCAM (Binary CAM) and TCAM (Ternary CAM). The BCAM treats only two-valued vectors without *don't cares*, while the TCAM treats three-valued vectors with *don't cares*. This paper considers only BCAMs.

Example 2.1 *Table 1 is the address table where the number of inputs is $n = 5$ and the weight is $k = 7$. The address generation function for Table 1 can be implemented by a CAM with 5 bits and 7 words. (End of Example)*

Table 1. Example of address table.

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

From here, we will consider realization methods of the address generation functions using ordinary memory.

Definition 2.2 *Consider a function* $F(\vec{X}) : B^n \rightarrow \{0, 1, \ldots k\}$*, where* $B = \{0, 1\}$ *and* $\vec{X} = (x_1, x_2, \ldots, x_n)$*. Let* $(\vec{X}_L, \vec{X}_H)$ *be a partition of* $\vec{X}$ *with two parts. A* **decomposition chart** *of* F *is the two-dimensional matrix, where each column label has distinct assignment of elements in* $\vec{X}_L$*, and each row label has distinct assignment of elements in* $\vec{X}_H$*, and the corresponding matrix value is* $F(\vec{X}_L, \vec{X}_H)$*. The number of different column patterns in the decomposition chart is the* **column multiplicity***.* $\vec{X}_L$ *denotes* **bound variables***, and* $\vec{X}_H$ *denotes* **free variable**.

Lemma 2.1 *[12] The column multiplicity of the decomposition chart for an address generation function with weight* k *is at most* $k + 1$.

Lemma 2.2 *[12] Let* F *be an address generation function. Then, there exists a functional decomposition*

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

such that G *and* H *are address generation functions and the weights of* F *and* G *are equal.*

Example 2.2 *Consider the decomposition chart in Fig. 2, which shows an address generation function* $F(\vec{X})$ *with weight 7. Let the function* $F(\vec{X})$ *be decomposed as* $F(\vec{X}_1, \vec{X}_2) = G(\vec{H}(\vec{X}_1), \vec{X}_2)$*, where* $\vec{X}_1 = (x_1, x_2, x_3, x_4)$ *and* $\vec{X}_2 = (x_5)$*. Then, the column multiplicity of the decomposition chart shown in Fig. 2 is* 7*. Table 2 shows the function* $\vec{H}$*. It is a* 4*-input* 3*-output address generation logic function with weight 6. The decomposition chart for G is shown in Fig. 3. As shown in this example, the functions obtained by decomposing the address generation function* F *are also address generation functions, and the weights of* F *and* G *are both* 7*. (End of Example)*

3 Realization of Address Generation Functions

In this section, we show three methods for realizing address generation functions on Xilinx FPGAs. Let k be the number of registered vectors, and n be the number of bits in the registered vectors.

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				

Figure 2. Decomposition chart for function F.

Table 2. Truth table for function $\vec{H}$.

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

3.1 CAM using Xilinx 4-input LUTs

Fig. 4 shows a 4-input LUT of a Xilinx FPGA. A Xilinx LUT can also be used as a shift register. Thus, we can dynamically change the contents of LUTs. Fig. 5 shows a circuit to detect a 12-bit registered vector using three 4-input LUTs. Each 4-input LUT realizes a registered vector of 4-bits. Fig. 5 shows a method to increase the number of bits in a registered vector. It uses cascaded multiplexers to perform the AND functions. We can realize an arbitrary registered vector by rewriting LUTs. Let k be the number of registered vectors. By using k copies of this circuits, and by attaching the encoder, we can realize an address generation function. We call this realization method as **4-LUT**

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	

Figure 3. Decomposition chart for function G.

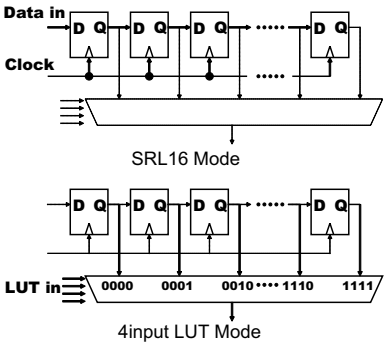

Figure 4. 4-input LUT of Xilinx FPGA.

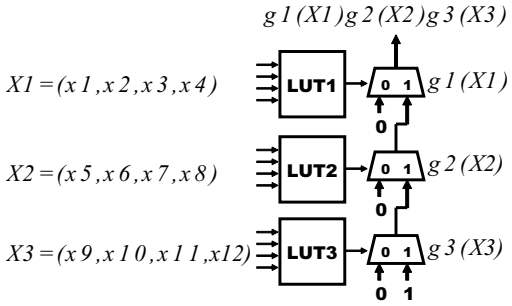

Figure 5. 12-bit registered vector detection circuit using 4-input LUTs.

method. This method requires $k\lceil\frac{n}{4}\rceil$ 4-input LUTs for registrable vectors, and $\lceil\frac{k-2}{6}\rceil\lceil\log_2(k+1)\rceil$ 4-input LUTs to realize the encoder for k vectors.

3.2 CAM using Xilinx IP

Fig. 6 illustrates a realization of an address generator using a BRAM. When the BRAM is used as a memory, data is written to the array by rows (Fig. 6(a)). On the other hand, when the BRAM is used as an address generator (or a CAM), registered vectors represented by a 1-hot code are written to the array by columns. For a given search vector, when the vector is registered, the BRAM produces a non-

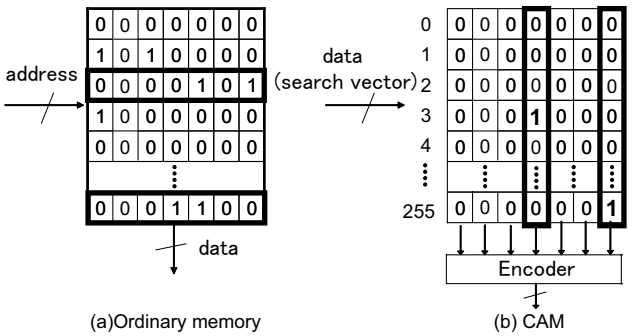

Figure 6. Address generator using BRAM.

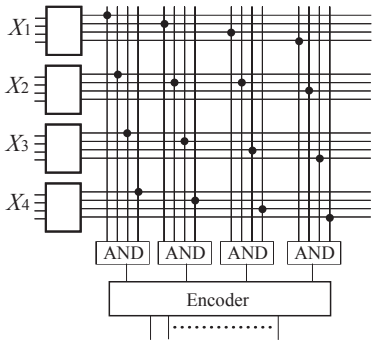

Figure 7. Address generator using BRAM+CLB.

zero output. An address generator is realized by attaching an encoder to the output of the BRAM (Fig. 6(b)). The number of registered vectors can be increased by using multiple BRAMs. Also, the number of bits for registered vectors can be increased by attaching AND gates to the outputs of the BRAMs (Fig. 7). We call this realization a **BRAM+CLB method**, since CLBs are used to implement the encoder. A Xilinx dual-port BRAM (each memory has 18 Kbits) can realize a pair of 16 word $\times$ 8 bit address generators. Thus, the total number of BRAMs necessary to implement an n-bit k-word CAM is $\frac{1}{2}\lceil\frac{k}{16}\rceil\lceil\frac{n}{8}\rceil$.

Example 3.3 *In Fig. 6(b), assume that columns have two different 1-hot codes and all zero codes. When the input vector is 00000011, only the fourth output is 1. Also, when the input vector is 11111111, only the last output is 1. (End of Example)*

3.3 CAM using LUT cascade

Definition 3.3 *A* ***pq*-element** *implements an arbitrary p-input q-output logic function. Its memory size is $2^p \cdot q$.*

Theorem 3.1 *[12] An address generation logic function with weight k can be realized by an LUT cascade with at most $\lceil\frac{n-q}{p-q}\rceil$ pq-elements, where $p > q$, and $q = \lceil\log_2(k+1)\rceil$.*

Example 3.4 *In Table* 1*, the number of registered vectors is $k = 7$. Thus, $q = \lceil\log_2(k+1)\rceil = \lceil\log_2(7+1)\rceil = 3$. The address generator can be realized by using* 4*-input* 3*-output elements as shown in Fig.*8*. (End of Example)*

Theorem 3.1 shows that any address generator can be realized as a cascade of pq-elements.

4 Addition and Deletion of Registered Vectors

The addition and deletion of registered vectors in an ordinary CAM can be done like an ordinary memory, and

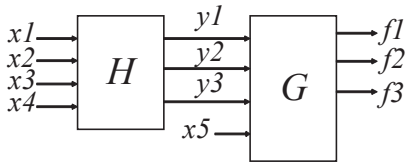

Figure 8. Realization of address generation logic function F.

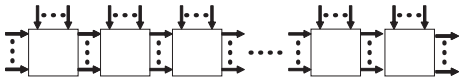

Figure 9. Cascade realization of address generation function.

these operation can be done quickly. For the CAM function implemented by the 4-LUT method (Fig. 5) or the BRAM+CLB method (Fig. 7), the time to add or delete registered vectors is proportional to n. On the other hand, for the CAM function implemented by LUT cascades, addition and deletion of a registered vector may not be done so quickly, since the data for the registered vector is distributed among many LUTs.

In the rest of this section, we will show that, in the LUT cascade, we can modify the registered vectors quickly. The modification of an address table for an LUT cascade can be divided into two basic operations: a *deletion of a vector* and an *addition of a vector*.

This part considers addition and deletion of a registered vector for three methods.

4.1 Modification of LUT Cascades

An LUT cascade is obtained by applying functional decompositions repeatedly. Thus, to find a method of modification, we need only to consider the modification for functions H and G, when F is decomposed as $F(X_1, X_2) = G(H(X_1), X_2)$.

4.1.1 Deletion of a Registered Vector

When the output value of F for an input vector $(\vec{a}, \vec{b})$ is reset to zero (i.e., to delete the vector), we perform the following operations to the functional decomposition $F(X_1, X_2) = G(H(X_1), X_2)$.

1. Reset $G(H(\vec{a}), \vec{b})$ to zero.
2. If there exists only one registered vector X_2 that satisfies $G(H(\vec{a}), X_2) \neq 0$, then reset $H(\vec{a})$ to 0.

Example 4.5 *Suppose that Table* 1 *is realized by a cascade structure shown in Fig.* 8*. Let us delete the third registered vector* $(x_1, x_2, x_3, x_4, x_5) = (0, 1, 0, 0, 0)$*. The output of* $\vec{H}$ *corresponding to this vector is* $(y_1, y_2, y_3) = (0, 1, 1)$*. The output of* G *is* 3*. First, we reset the output value of* $G(0, 1, 1, 0)$ *to* 0*. Next, since there exists only one input* X_2 *such that* $G(H(0, 1, 0, 0), X_2) \neq 0$*, we reset the output value of* $\vec{H}(0, 1, 0, 0)$ *to* 0.

Next, let us delete the fifth registered vector $(x_1, x_2, x_3, x_4, x_5) = (0, 1, 1, 1, 0)$*. The output of* $\vec{H}$ *corresponding to this vector is* $(y_1, y_2, y_3) = (1, 0, 1)$*. The output of* G *is* 5*. First, we reset the output value of* $G(1, 0, 1, 0)$ *to* 0*. Next, since there exits two inputs* X_2 *such that* $G(H(0, 1, 1, 1), X_2) \neq 0$*, we keep the output value* $\vec{H}(0, 1, 1, 1)$ *unchanged.*

Table 3 *and Fig.* 10*, respectively, show the functions* $\vec{H}$ *and* G *after deletion of two registered vectors.* ***(End of Example)***

Table 3. Truth table for $\vec{H}$ after deletion of registered vectors.

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

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	

Figure 10. Decomposition chart for G after deletion of registered vectors.

4.1.2 Addition of a Registered Vector

When we append a registered vector that makes the output value of F for address $(\vec{a}, \vec{b})$ to c, we perform the following operations to the functional decomposition $F(X_1, X_2) = G(H(X_1), X_2)$.

1. If $H(\vec{a}) \neq 0$, then set $G(H(\vec{a}), \vec{b})$ to c

Table 4. Truth table for $\vec{H}$ after addition of registered vectors.

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	

Figure 11. Decomposition chart for G after addition of registered vectors.

2. If $H(\vec{a}) = 0$, then set $H(\vec{a})$ to e and set $G(\vec{e}, \vec{b})$ to c, where e is the minimum unused integer.

Example 4.6 *Suppose that the address table in Table* 1 *is realized by a cascade structure in Fig.* 8*. Let us add the input vector* $(x_1, x_2, x_3, x_4, x_5) = (0, 0, 0, 1, 1)$ *whose index value is 8. Since the output of* $\vec{H}$ *corresponding to this input vector is* $H(0, 0, 0, 1) \neq 0$*, we set the output value of* G *to* $G(H(0, 0, 0, 1), 1) = 8$*.*

Also, let us add the input vector $(x_1, x_2, x_3, x_4, x_5) = (0, 0, 0, 0, 0)$ *whose index value is 9. Since the output of* $\vec{H}$ *corresponding to this vector is* $H(0, 0, 0, 0) = 0$*, we set to* $\vec{H}(0, 0, 0, 0) = (1, 1, 1)$*, and also set the output value of* G *to* $G(H(0, 0, 0, 0), 0) = 9$*.*

Table 4 *and Fig.* 11*, respectively, show the function* $\vec{H}$ *and function* G *after addition of two registered vectors.*

(End of Example)

4.1.3 Modification Method for LUT Cascades

To reduce the modification time of an LUT cascade for each cell, we use a **reference table** (controlled by software) that maintains assigned **rail vectors**. Fig. 12 shows an example of a reference table. In this table, the address denotes an assigned rail vector, while the value of the table denotes the number of references for the rail vector. When we add a new registered vector, we first check the reference table, then search for the unreferenced rail vector. Also, when we delete a registered vector, we first check the number of references for this rail vector. If the number of references is zero, we delete the corresponding rail vector from the cell in the LUT cascade (reset the vector to zero).

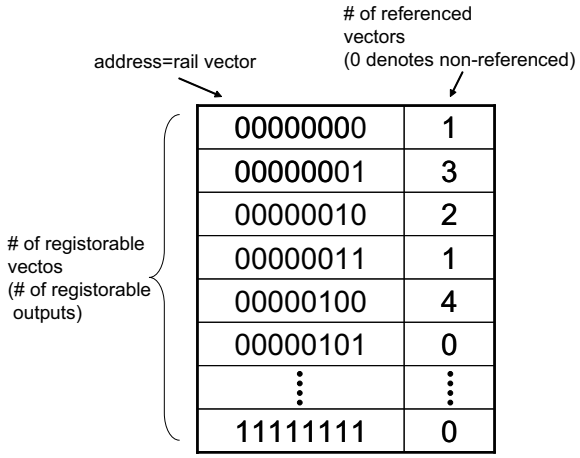

Figure 12. Reference table.

We show algorithms to add and delete registered vectors of an address generator implemented by an LUT cascade.

Algorithm 4.1 *(Addition of a registered vector)*

1 Check whether the vector is registered or not. If the output of the LUT cascade for the vector is non-zero, then go to 3, since non-zero denotes that the vector is already registered.

2 Apply the input to each cell, and read the output of each cell. Then, perform the following operations:

2.1 If the output of the cell is 0*, then search the reference table for the cell, and find the unreferenced rail vector. The number of steps is at most* k*, the number of the registered vectors. Next, write the assigned rail vector to the cell.*

2.2 Increment the value of the reference table for the assigned rail vector.

3 Terminate.

Algorithm 4.2 *(Deletion of a registered vector)*

1 Check whether the vector is registered or not. If the output of the LUT cascade for the vector is zero, then the vector is unregistered, and go to 3.

2 Add the input to each cell, and read the output. Then, perform the following operations:

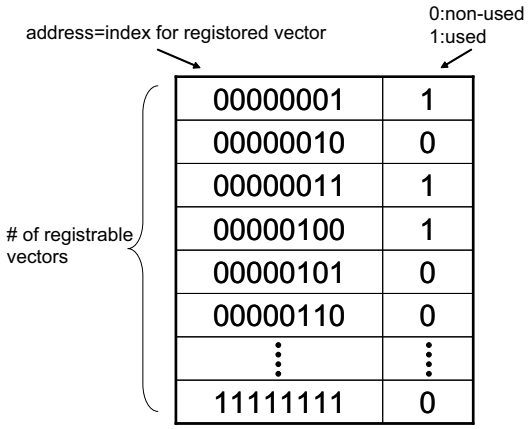

Figure 13. Index table.

2.1 Decrement the value of the reference table corresponding to the rail vector.

2.2 If the value of the reference table is zero, then write a zero to the cell.

3 Terminate.

The number of steps to add one registered vector to the LUT cascade is estimated as follows:

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

where *Op.Cas* is the number of steps to operate the LUT cascade, *Acc.Mem* is the number of steps to access the memory that stores reference tables, s is the number of cells, and k is the number of registered vectors. The first term corresponds to the estimated number of steps in Algorithm 4.1-1, and the second term corresponds to the estimated number of steps in Algorithm 4.1-2. Similarly, the number of steps to delete one registered vector from the LUT cascade is estimated as follows:

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

The first term corresponds to the estimated number of steps in Algorithm 4.2-1, and the second term corresponds to the estimated number of steps in Algorithm 4.2-2.

4.2 Modification of CAMs using 4-LUT Method and BRAM+CLB Method

To reduce the time to modify the address generator using 4-LUT method (Fig. 5) and BRAM+CLB method (Fig. 6, 7), we use an **index table** (controlled by software) that maintains indices[1]. Fig. 13 shows an example of an index table, where the address corresponds to the index of the vector, and the value of the table shows whether the index is used or not. When we add a registered vector, we first check the index table. If the index is unused, then we write 1 to the index table and write the vector to the CAM. When we delete a registered vector, we first use the CAM to check whether the vector is registered or not. If the vector is registered, we reset the value of the index table to zero, and delete the registered vector from the CAM.

We show two algorithms for addition and deletion of a vector in CAMs using 4-LUT method and BRAM+CLB method.

Algorithm 4.3 *(Addition of a registered vector)*

1 Check whether the vector is registered or not by using the CAM. If the output is non-zero, then go to 4, since the vector is already registered.

2 Search the index table, and find an unused index.

3 Enter 1 in the index table. Also, write the registered vector to the CAM.

4 Terminate.

Algorithm 4.4 *(Deletion of a registered vector)*

1 Check whether the vector is registered or not by using the CAM. If the output is zero, then go to 3, since the vector is un-registered.

2 Delete the registered vector from the CAM. Also, reset the index table corresponding the vector to zero.

3 Terminate.

The number of steps to add one registered vector to a CAM is estimated as follows:

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

where *Op.CAM* is the number of steps to operate a CAM. The first term corresponds to the estimated number of steps in Algorithm 4.3-1, and the second term corresponds to the estimated number of steps in Algorithm 4.3-2. Also, the number of steps to delete one registered vector from the CAM is estimated as follows:

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

The first term corresponds to the estimated number of steps in Algorithm 4.4-1, and the second term corresponds to the estimated number of steps in Algorithm 4.4-2.

From equations (1), (2), (3), and (4), when *Op.CAM*, *Op.Cas*, and *Acc.Mem* have the same values, the number of steps for the LUT cascade is about s times of 4-LUT method or BRAM+CLB method.

[1] In a CAM produced by the Xilinx Core Generator [16], users must maintains the index. By attaching the priority encoder that produces unused indices to the CAM, we can implement by hardware. However, since many applications uses both CAM and software, in this paper, we maintain the references by software.

Table 5. FPGA and Design Tool.

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 Embedded Multipliers :	12
Design Tool: Xilinx, ISE Web Pack 7.2i	

Table 6. Code sizes for maintaining CAM.

Library and StandAlone OS	11806 [Byte]
Code to modify the 4-LUT (BRAM)	892 [Byte]
Code to modify the LUT cascade	2293 [Byte]

5 Experimental Results

5.1 Size and Speed

For different values of k and n, we implemented address generators on an Xilinx FPGA. Table 5 shows the FPGA device and design tool used in the experiment. Table 7 compares the amount of hardware and the speed of implemented address generators. To generate CAMs using BRAM+CLB method, we used the Xilinx Core Generator. In this case, we did not implement the pipeline in the BRAM+CLB method. CAMs using 4-LUT method also can be generated by the Core Generator. However, we developed our own CAMs using the 4-LUT method to implement a pipeline structure. Rewriting a word for a CAM using 4-LUT method requires 16 clocks, while rewriting a word to a CAM using BRAM+CLB method requires only two clocks. On the other hand, rewriting a word of a BRAM for the LUT cascade requires only one clock. We also implemented rewriting circuits by hardware. Note that, each method requires many fewer clocks to rewrite a word to a CAM than to generate the rewritten data by MicroBlaze. (This will be discussed later). In Table 7, *4-LUT* denotes the amount of hardware the using 4-LUT method (Fig. 5); *BRAM+CLB* denotes the amount of hardware using BRAM+CLB method (Fig. 7); and *Cascade* denotes the amount of hardware using the LUT cascade (Fig. 9). To compare the total area, we assume that one 4-input LUT corresponds to 96 bits of a BRAM [14]. We derived the **normalized area** as follows:

$$\begin{aligned} \text{normalized area} &= (\#\text{ of Slice} \times 2) \\ &\quad +(\#\text{ of BRAM} \times 192). \end{aligned} \quad (5)$$

LUT cascades vs. 4-LUT method The area for the LUT cascade is 16% to 27% of that for 4-LUT method. The operating frequency for the LUT cascade is about three times higher than that for 4-LUT method. One reason for this is that the LUT cascade realization is smaller than the 4-LUT method, so the LUT cascade is easier to place and route than the 4-LUT method. Other reason is that the LUT cascade has the shorter critical path than 4-LUT method, since the LUT cascade realization requires no encoder, while a CAM using the 4-LUT method requires an encoder. Note that the delay time of the encoder is rather high.

LUT cascades vs. BRAM+CLB method The area for the LUT cascade is 8% to 11% of that for BRAM+CLB. The operation frequency for the LUT cascade is about four to five times higher than that for the BRAM+CLB method. The reason for this is the same as the case of the 4-LUT method. Note that, the LUT cascade and 4-LUT method use a pipeline architecture, while the BRAM+CLB method does not[2]. Thus, extra experiment using pipelined BRAM+CLB method is necessary.

5.2 Sizes of Executable Codes

We implemented the embedded processor MicroBlaze on Xilinx FPGA for the minimum configurations with I/O ports and an interrupt timer. The size of the available memory is 8 Kbytes. Also, we used StandAlone OS for a single application. Then, we implemented modification programs for registered vectors described in Chapter 4 with the C language. Table 6 shows the sizes of the modification programs. Note that, since we assigned the unused memories to the work area, Table 6 does not include the work area. Table 6 shows that the size of the modification code for the LUT cascade is larger than that for 4-LUT(BRAM+CLB) methods. However, since the size of the library and OS code are much larger, the size of the modification code for the LUT cascade is negligible.

5.3 Modification Time

To measure the modification time, we attached a clock counter to MicroBlaze, and counted the number of clocks necessary to add and delete vectors for each method. Table 8 shows the results, for $s = 8$, $k = 255$, and $n = 32$. We executed the modification program, and obtained the number of clocks to modify 255 vectors. From Table 8, the number of clocks to add vectors to the LUT cascade is about s times larger than that of 4-LUT (BRAM+CLB) method, where s is the number of cells in the cascade. This result agrees with the observations in Chapter 4. On the other hand, the number of clocks to delete vectors from the LUT cascade is 5.39 times larger than that of 4-LUT (BRAM+CLB) method. This result is smaller than the estimated value in Chapter 4. The reason for this is observed as follows: Since the LUT cascade shares the rail vectors

[2]In this experiment, we used the Xilinx IP Core Generator. So, we could not modify the circuit. Thus, we implemented the circuit without a pipeline.

Table 7. Amount of Hardware and Speed for Three Methods.

	$k = 255, n = 32$			$k = 255, n = 40$			$k = 511, n = 32$		
	4-LUT	BRAM +CLB	Cascade	4-LUT	BRAM +CLB	Cascade	4-LUT	BRAM +CLB	Cascade
# of Slice (# of 4-LUTs)	2495 (4226)	1026 (1894)	70 (0)	2990 (5127)	1204 (2298)	128 (0)	4984 (8456)	2014 (3688)	110 (0)
# of Block RAM (# of cells)	— (—)	32 (—)	4 (8)	— (—)	40 (—)	6 (11)	— (—)	64 (—)	6 (12)
normalized area	4226	8038	908	5127	12807	1408	8456	15976	1372
max. operation freq [MHz] (delay time [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)

Table 8. Modification time (# of clocks).

	Add	Delete
4-LUT (BRAM+CLB)	1590	1184
LUT cascade	12698	6393
Ratio(4-LUT (BRAM+CLB)/ LUT cascade)	7.98	5.39

in each cell, the reference table is rather sparse. Thus, the search time for reference tables is shorter than the worst case. On the other hand, since the 4-LUT (BRAM+CLB) method used only one index table, the index table is rather dense.

6 Conclusion

In this paper, we presented a realization method for address generation functions using LUT cascades on an FPGA. Cells of the LUT cascade are implemented by embedded memories (BRAMs) on an FPGA. The embedded processor adds and deletes registered vectors. Compared with a CAM produced by the Xilinx Core Generator, our method produces much smaller circuits (1/5 or smaller in this experiment). Also, since our circuit is smaller than the Xilinx CAM, it is four to five times faster. The demerit is that our method requires longer time to modify registered vectors. In this experiment, when the number of cells is 8, the modification time is 5 to 8 times longer. The time to add and delete registered vectors in the LUT cascade is proportional to the number of cells. In this paper, we considered FPGA realizations of the circuit.

7 Acknowledgements

This research is partly supported by Japan Society for the Promotion of Science (JSPS), MEXT, and Kitakyushu Innovative Cluster. Discussion with Prof. Jon T. Butler improved English presentation.

References

[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.