Design Methods for Multiple-Valued Input Address Generators

Tsutomu Sasao
Department of Computer Science and Electronics,
Kyushu Institute of Technology,
Iizuka 820-8502, Japan

Abstract

A multiple-valued input address generator produces a unique address given a multiple-valued input data vector. This paper presents methods to realize multiple-valued input address generators by multi-level networks of p-input q-output memories. It shows a method to simplify the address generators using an auxiliary memory.

1 Introduction

A multiple-valued input address generator produces an address given a multiple-valued input vector. Multiple-valued input address generators are used in databases, dictionaries, and password lists. Two-valued input address generators are used in address lists of the Internet, and memory patch circuits. In most cases, the number of registered vectors (stored addresses) is much smaller than the total number of possible input vectors.

Address generators can be directly implemented by Programmable Logic Arrays (PLAs)[1] or Content Addressable Memories (CAMs)[8]. However, these methods require special logic elements.

Address generators can also be implemented using ordinary logic elements. Fig. 5.1 shows an address generator implemented by registers, gates and a priority encoder. The demerit of this circuit is that the interconnections becomes complex as the number of registered vectors increases.

This paper present a new design method to implement multiple-valued input address generators by multi-level networks of p-input q-output memories.

2 Applications of Address Generators

In this part, we present various applications of address generators.

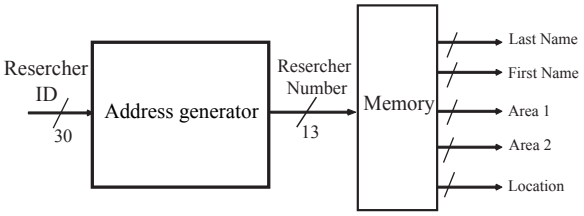

Figure 2.1. Database for multiple-valued logic researchers.

2.1 Database for Multiple-Valued Logic Researchers

Consider Table 2.1, which shows a part of a database for multiple-valued logic researchers [5], and Fig. 2.1, which shows an implementation. This stores, *Researcher number* (integer), *Researcher ID* (English alphabet), *Last name*, *First name*, *Research area 1*, *Research area 2* and *Location*. We assume that the number of researchers in the database is at most 8000. The database consists of two circuits:

1. A circuit to produce the *Researcher number* from the *Researcher ID*.
2. A circuit to produce *Last name*, *First name*, *Research areas*, and *Location* from *Researcher number*.

The first circuit is implemented by an *address generator*, and the second circuit is implemented by an ordinary memory. *Researcher ID* consists of 6 letters from the 26-letter English alphabet or special symbols (the underscore and the blank). To represent 6 characters, we need $5 \times 6 = 30$ bits, since each character requires 5 bits. On the other hand, to represent a *Researcher number*, we need only 13 bits, since the total number of researchers is at most 8000. Note that, in this case, the number of possible input combinations is 2^{30}, while the registered input combinations is at most 8000.

Table 2.1. Database for multiple-valued logic researchers.

Number	ID	Last Name	First Name	Area 1	Area 2	Location
1	vranes	Vranesic	Zvonko	circuit	logic design	Toronto
2	moraga	Moraga	Claudio	spectral method	fuzzy logic	Dortmund
3	smith	Smith	Kenneth	circuit		Toronto
4	muzio	Muzio	Jon	spectral method	test	Victoria
5	miller	Miller	Michael	spectral method	logic design	Victoria
6	rosenb	Rosenberg	Ivo	clone theory		Montreal
7	higuch	Higuchi	Tatsuo	circuit	signal processing	Sendai
8	kameya	Kameyama	Michitaka	circuit	logic design	Sendai
9	ishizu	Ishizuka	Okihiko	circuit	logic design	Miyazaki
10	sasao	Sasao	Tsutomu	logic design	decision diagram	Iizuka
11	butler	Butler	Jon	logic design	decision diagram	Monterey
12	mukaid	Mukaidono	Masao	fuzzy logic	logic design	Tokyo
13	simovi	Simovic	Dan	algebra	database	Boston
14	perkow	Perkowski	Marek	logic design	decision diagram	Portland
15	hanyu	Hanyu	Takahiro	circuit	logic design	Sendai
16	falkow	Falkowski	Bogdan	spectral method	logic design	Singapore
17	aoki	Aoki	Takafumi	circuit	arithmetic	Sendai
18	hata	Hata	Yutaka	fuzzy logic	image processing	Himeji
19	dubrov	Dubrova	Elena	logic design	test	Stockholm
20	dueck	Dueck	Gerhard	logic design	reversible	Fredericton
21	thornt	Thornton	Mitch	spectral method	decision diagram	Dallas
22	drechs	Drechsler	Rolf	decision diagram	verification	Bremen

2.2 Address Table in the Internet

IP addresses of the internet are often represented by 32 bits. An *address table* for a router stores IP addresses and corresponding indeces for the memory that stores the details of the addresses. We assume that the number of addresses in the table is at most 40,000. Thus, the number of inputs is 32 and the number of outputs is 16. Note that the address table must be updated frequently.

2.3 Memory Patch Circuit

The firmware of an embedded systems is usually implemented by Read-only memories (ROMs). After shipping the product, it is often necessary to modify a part of the ROM to upgrade to a later version. To convert the address of the ROM to the address of the auxiliary memory, we use the address generator shown in Fig. 2.2 [9, 3, 4].

2.4 Properties of Address Generators

The previous three examples of address generators have common properties:

1. The values of the non-zero outputs are distinct.
2. The typical number of non-zero output values is much smaller than the maximum possible.

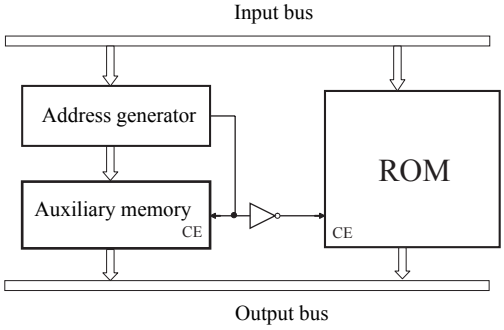

Figure 2.2. Memory patch circuit.

3. High-speed circuits are required.
4. Data must be updated.

The last condition requires address generators to be programmable.

3 Definition and Basic Properties

Definition 3.1 *A mapping* $F(\vec{X}) : M^N \rightarrow \{0,1,\ldots,k\}$, *where* $M = \{0,1,\ldots,m-1\}$ *is an* m**-valued input integer function**. *If* $F(\vec{a}_i) \neq 0$ $(i = 1,2,\ldots,k)$ *for* k *different input vectors, and* $F = 0$ *for other* $(m^n - k)$ *input vectors, then the the* **weight** *of the function is* k.

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

Figure 3.1. Decomposition chart for F.

Definition 3.2 *A mapping $F(\vec{X}) : M^N \rightarrow \{0,1,\ldots,k\}$, where $M = \{0,1,\ldots,m-1\}$ is an m-valued input address generation function if $F(\vec{a}_i) = i$ $(i = 1,2,\ldots,k)$ for k different m-valued registered vectors $\vec{a}_i \in M^N$ $(i = 1,2,\ldots,k)$, and $F = 0$ for other $(m^n - k)$ input vectors. In other words, an m-valued input address generation function produces addresses ranging from 1 to k for k distinct m-valued vectors and produces 0 (special address) for other vectors. The multiple-output logic function that represents the output values in binary numbers is the m-valued input address generation logic function: it is denoted by $\vec{F}$.*

Definition 3.3 *Consider a function $F(\vec{X})$: $M^N \rightarrow \{0,1,\ldots k\}$, where $M = \{0,1,\ldots,m-1\}$ and $\vec{X} = (x_1,x_2,\ldots,x_N)$. Let $(\vec{X}_1,\vec{X}_2)$ be a partition of $\vec{X}$. A* **decomposition chart** *of F is the two-dimensional matrix, where each column label has a distinct assignment of elements in M to $\vec{X}_1$, and each row label has distinct assignment of elements in M to $\vec{X}_2$, and the corresponding matrix value is $F(\vec{X}_1,\vec{X}_2)$. The number of different column patterns in the decomposition chart is the* **column multiplicity**. *$\vec{X}_1$ denotes* **bound variables**, *and $\vec{X}_2$ denotes* **free variable**.

Example 3.1 *Consider the decomposition chart in Fig.3.1, which shows a two-valued input address generation function $F(\vec{X})$ with weight 7. $\vec{X}_1 = (x_1,x_2,x_3,x_4)$ denotes bound variables, and $\vec{X}_2 = (x_5)$ denotes the free variable. Note that the column multiplicity of this decomposition chart is 7. (End of Example)*

Lemma 3.1 *The column multiplicity of the decomposition chart for an address generation function with weight k is at most $k+1$.*

(Proof) Since the number of non-zero outputs is k, the column multiplicity never exceeds $k+1$. (Q.E.D.)

Lemma 3.2 *Let F be an address generation function with weight k. Then, there exists a functional decomposition*

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

where G and H are address generation functions, and the weight of G is k, and the weight of H is at most k, and H takes at most $(k+1)$-values.

Table 3.1. Truth table for $\vec{H}$.

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

(Proof) Consider a decomposition chart, in which $\vec{X}_1$ denotes the bound variables, and $\vec{X}_2$ denotes the free variables. Let $\vec{X}_1 = (x_1,x_2,\ldots,x_p)$, where $p > \lceil \log_2(k+1) \rceil$. Let H be a function where the input variables are X_1, and the output values are defined as follows: Consider the decomposition chart, where assignments of values to X_1 label columns (i.e., bound variables). For the assignments to X_1 corresponding to columns with only zero elements, $H = 0$. For other inputs, the outputs are distinct integers from 1 to w_h, where w_h denotes the number of columns that have non-zero element(s). Since $w_h \leq k$, the weight of H is at most k, and the number of output values of H is at most $k+1$. On the other hand, the function G is obtained from F by reducing some columns that have all zero outputs in the decomposition chart. Thus, the number of non-zero outputs in G is equal to the number of non-zero outputs in F. Thus, G is also an address generation function with the weight k. (Q.E.D.)

Example 3.2 *Consider the decomposition chart in Fig.3.1, 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 $\vec{X}_2 = (x_5)$. Table 3.1 shows the function $\vec{H}$. It is a 4-input 3-output address generation logic function with weight 6. The decomposition chart for the function G is shown in Fig. 3.2. As shown in this example, the functions obtained by decomposing address generation function F are also address generation functions, and the weights of F and G are both 7. (End of Example)*

In general, a multiple-valued variable can be represented by using two-valued variables. Thus, an arbitrary multiple-valued input multiple-valued output function can be represented as a two-valued input two-valued output network. To

x_1	0	0	0	0	1	1	1	1
x_2	0	0	1	1	0	0	1	1
x_3	0	1	0	1	0	1	0	1
$x_5 = 0$		1		3	4	5		
$x_5 = 1$			2			6	7	

Figure 3.2. Decomposition chart for G.

implement the network by using two-valued logic elements, in this paper, we assume that all the multiple-valued variables are represented by two-valued variables.

Let N be the number of m-valued variables, and let n be the number of two-valued variables. Then, we have the following relation: $n = \lceil \log_2 m \rceil N$.

4 Synthesis of Address Generators

Definition 4.1 *A pq-element implements an arbitrary p-input q-output logic function. Its* **memory size** *is $q2^p$.*

Theorem 4.1 *An arbitrary two-valued input n-variable address generator with weight k can be realized as a multi-level network of pq-elements. The number of such elements is at most $\lceil \frac{n-q}{p-q} \rceil$, where $p > q$ and $q = \lceil \log_2(k+1) \rceil$.*

(Proof) An address generation logic function $\vec{F}$ with weight k can be decomposed as

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

where $X_1 = (x_1, x_2, \ldots, x_p)$. In this case, by Lemma 3.2, $\vec{G}(\vec{X}_1', \vec{X}_2)$ is also an address generation logic function with weight k. Note that the number of input variables for $\vec{G}$ is reduced to $n-(p-q)$, since the number of output variables of $\vec{H}$ is $q = \lceil \log_2(k+1) \rceil$. By iterating this operations $\lceil \frac{n-p}{p-q} \rceil$ times, we can reduce the number of variables at most to p. Thus, the address generator can be implemented by using only pq-elements. The number of elements is at most $\lceil \frac{n-p}{p-q} \rceil + 1 = \lceil \frac{n-q}{p-q} \rceil$. (Q.E.D.)

Example 4.1 *The number of non-zero outputs in the 5-variable address generation function $F(\vec{X})$ shown in Fig.3.1 is $k = 7$. Since $q = \lceil \log_2(k+1) \rceil = \lceil \log_2(7+1) \rceil = 3$, the address generator can be realized by 4-input 3-output elements as shown in Fig. 4.1. (End of Example)*

When realizing an address generator by pq-elements, increasing p decreases the number of pq-elements, but increases the total amount of memory. On the other hand, decreasing p increases the number of pq-element, but decreases the total amount of memory. The next theorem shows a strategy to design address generators using pq-elements. It finds a value of p that minimizes the least upper bound on the total amount of memory without increasing the number of elements.

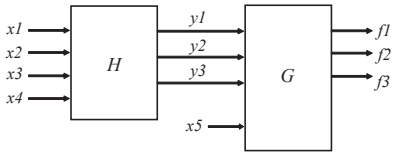

Figure 4.1. Realization of address generation function F.

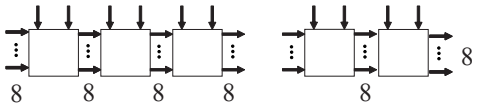

Figure 4.2. Cascade realization of address generator.

Theorem 4.2 *When an address generator is implemented as a multi-level network of pq-elements, the least upper bound on the total amount of memory is minimized when $p-q = 1$ or $p-q = 2$.*

(Proof) When an address generation function is decomposed into pq-elements, for each decomposition, we can reduce the number of input variables by $r = p-q$. To reduce n inputs into q, we need $s = \lceil \frac{n-q}{r} \rceil$ functional decompositions. To realize the address generator, we need s pq-elements. Thus, the total amount of memory necessary to implement the address generator is $MEM = s \cdot 2^p q$. When n is sufficiently large, MEM can be approximated by $(\frac{2^r}{r}) \cdot (n-q) \cdot 2^q q$. Since n and q are fixed for a given problem, only r can be changed. Note that $\frac{2^r}{r}$ takes its minimum when $r = 1$ or $r = 2$. Hence we have the theorem. (Q.E.D.)

Since networks with fewer levels are desirable, we often select $r = p-q = 2$ to design the address generator.

Theorem 4.1 shows that we can design an address generator as a multi-level network of pq-elements by iterations of functional decompositions.

The next Example 4.2 shows that we can generate various multi-level logic networks, including cascades.

Example 4.2 *Let us design address generators, where the number of inputs is $n = 48$ and the weight is $k = 255$. Since $q = \lceil \log_2(255+1) \rceil = 8$, when $p = 10$, the total amount of memory is minimized, and also the number of levels is minimized. For each pq-element, we can reduce the number of input lines by two. So, by using 20 pq-elements, we can reduce the number of inputs into 8. For example, we have the LUT cascade as shown in Fig.4.2. Or, we have the multi-level logic network shown in Fig. 4.3, where the number*

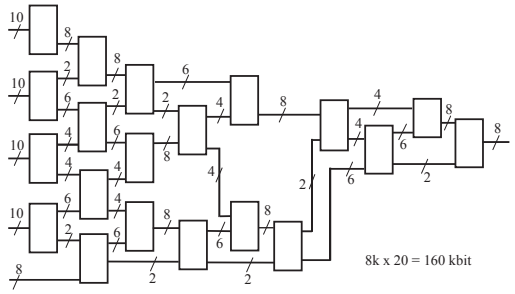

Figure 4.3. Address generator using pq-elements ($p = 10$).

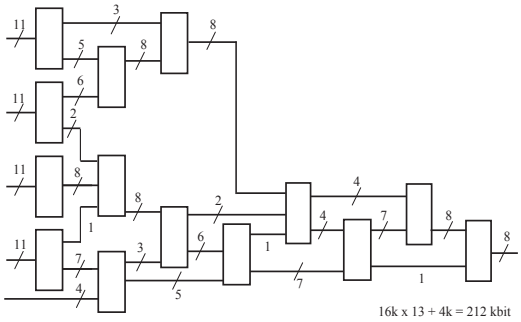

Figure 4.4. Address generator using pq-elements ($p = 11$).

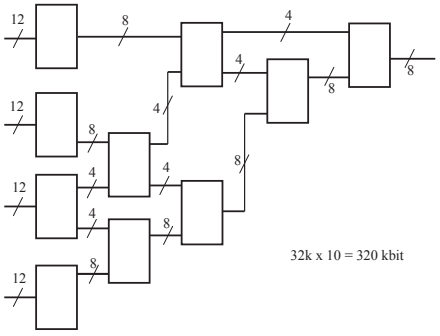

Figure 4.5. Address generator using pq-elements ($p = 12$).

of levels is 10*. In this case, the variables are permutated during functional decompositions. Note that both structures require the same amount of memory:* 160*k bits. We can further reduce the number of levels by using elements with more inputs. Fig.4.4 shows an example with* $p = 11$ *and* $q = 8$*. In this case, the number of elements is* $(48-8)/(11-8) = 14$*, the number of levels is* 8*, and the total amount of memory is* 212 *k bits. Fig. 4.5 show an example with* $p = 12$ *and* $q = 8$*. In this case, the number of elements is* $(48-8)/(12-8) = 10$*, the number of levels is* 5*, and the total amount of memory is* 320 *k bits. (End of Example)*

Table 4.1. Decomposition chart for address generation function F.

$x_3x_4x_5$ \ x_1x_2	00	01	10	11
000	0	0	0	0
001	1	0	0	0
010	2	0	0	0
011	3	0	0	0
100	4	0	0	0
101	5	0	0	0
110	6	0	0	0
111	7	0	0	0

Table 4.2. Truth table for H.

x_1x_2	y_1
00	1
01	0
10	0
11	0

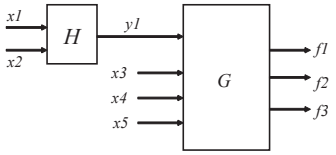

Figure 4.6. Address generator for Table 4.1.

Theorem 4.2 shows the strategy for general address generators. It minimize the least upper bound on the total amount of memory. For a particular address generator, the total amount of memory can be minimum for the cases other than $p - q = 2$. The next example illustrates this.

Example 4.3 *Consider the* 5*-variable address generation function* $F(\vec{X})$ *shown in Table 4.1. 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* $\vec{X}_1 = (x_1, x_2)$*, and* $\vec{X}_2 = (x_3, x_4, x_5)$*. The column multiplicity of the decomposition chart in Table 4.1 is* 2*.*

Table 4.2 is the truth table of H, and Table 4.3 is the truth table of G. This address generator can be implemented as Fig.4.6. In this case, the weight of the function is $k = 7$*, but H is realized by a* 2*-input* 1*-output element. (End of Example)*

5 Comparison with Other Realizations

Another method to realize an address generator is shown in Fig. 5.1. It consist of registers, gates and an encoder. This method requires one register for each registered vector.

Let n be the number of input variables, and k be the number of registered vectors. To detect each address, we need a n-bit register, n copies of coincidence circuits (EXNORs), and an n-input AND gate. To generate the output address, we

Table 4.3. Truth table for G.

y_1	x_3	x_4	x_5	f_1	f_2	f_3
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	0	0	0	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	0	0	0
0	1	1	1	0	0	0
1	0	0	0	0	0	0
1	0	0	1	0	0	1
1	0	1	0	0	1	0
1	0	1	1	0	1	1
1	1	0	0	1	0	0
1	1	0	1	1	0	1
1	1	1	0	1	1	0
1	1	1	1	1	1	1

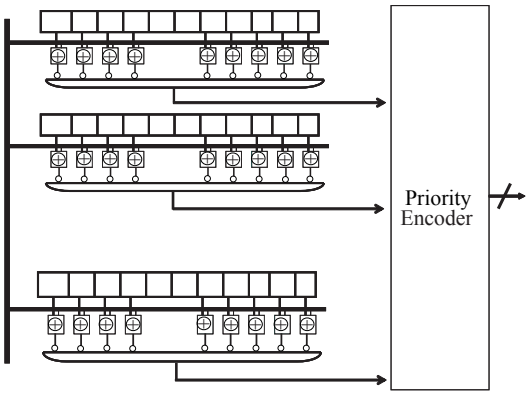

Figure 5.1. Address generator implemented by registers, gates and a priority encoder.

need a k-input $\lceil \log_2(k+1) \rceil$-output priority encoder. Thus, in total we need, nk flip-flops, nk coincidence circuits, k n-input AND gates, and a k-input $\lceil \log_2(k+1) \rceil$-output priority encoder.

Note that this network can realize only the address generation logic functions. On the other hand, the multi-level network of pq-elements can realize wider class of functions.

6 Realization of Logic Functions with Weight k

Up to now, we have considered the realization of address generation functions. Next, we consider the realization of general logic functions.

Theorem 6.1 *An arbitrary two-valued n-input u-output function with weight k is realized as a multi-level network of pq-elements. The number of elements needed is at most* $\lceil \frac{n-q}{p-q} \rceil + \lceil \frac{u}{q} \rceil$, *where* $p > q$ *and* $q = \lceil \log_2(k+1) \rceil$.

(Proof) An arbitrary logic function with weight k can be realized as a cascade of an address generator and a decoder, where the address generator produces unique indices for k input combinations, and the decoder converts each index into corresponding outputs. The number of inputs of the decoder is at most $\lceil \log_2(k+1) \rceil$. By Theorem 4.1, the address generator can be realized with $s_1 = \lceil \frac{n-q}{p-q} \rceil$ elements. Also, note that the decoder can be realized by s_2 copies of q-input q-output elements, where s_2 is given by $s_2 = \lceil \frac{u}{q} \rceil$. Thus, total number of elements is $s_1 + s_2 = \lceil \frac{n-q}{p-q} \rceil + \lceil \frac{u}{q} \rceil$. (Q.E.D.)

Corollary 6.1 *An arbitrary two-valued n-variable single-output logic function with weight k is realized as a multi-level network of pq-elements. The number of elements needed is at most* $\lceil \frac{n-q}{p-q} \rceil$, *where* $p > q$ *and* $q = \lceil \log_2(k+1) \rceil$.

7 Address Generators using Auxiliary Memory

7.1 Don't Cares Generated by Multiple-Valued Input Variables

When an m-valued variable is represented by $\beta = \lceil \log_2 m \rceil$ two-valued variables, if m is not a power of two, then *don't cares* will occur. In other words, for each m-valued variable, we use only m valid combinations out of 2^β possible combinations of two-valued variables. Let N be the number of m-valued variables. The number of possible combinations of two-valued input variables is $2^{\beta N}$. On the other hand, the number of valid combinations for multiple-valued variables is only m^N. Thus, the fraction of *don't cares* is $\frac{2^{\beta N} - m^N}{2^{\beta N}} = 1 - (\frac{m}{2^\beta})^N$. For example, when $N = 6, m = 28$, we have $\beta = 5$. The fraction of *don't cares* is $1 - (\frac{28}{32})^6 \approx 0.55$. We can use these don't cares to simplify the address generators.

7.2 Design of Address Generators

In a typical m-valued input address generator, the number of non-zero outputs is much smaller than m^n, the total number of input combinations. Thus, we have the following:

Assumption 7.1 *Let N be the number of input variables, and k be the number of registered m-valued vectors. Then, we have the relation,* $k << m^N$.

For example, consider the the case of $N = 32$, $m = 2$ and $k = 1000$ vectors. The fraction of non-zero outputs over all input combinations is $\frac{1000}{2^{32}} = 2.3 \times 10^{-7}$. From Lemma 3.2, we can see that the column multiplicity of the decomposition chart is at most $k+1$. In fact, in many cases the column multiplicity is exactly $k+1$. Thus, to realize the address generator by a multi-level network of memories, we need many cells with $\lceil \log_2(k+1) \rceil + 1$ inputs and $\lceil \log_2(k+1) \rceil$ outputs.

We now show a method to reduce the amount of hardware by using an auxiliary memory. Fig. 7.1 illustrates the idea of the method.

Algorithm 7.1 *(Simplification of Address Generators).*

1. *Let F be an address generation function. Let* **Network** *1 be a circuit that realizes F. Let G be the function where the output values for the non-registered inputs in F are replaced by don't cares.*
2. *Construct the binary decision diagram for characteristic function (BDD_for_CF)[15] that represents the characteristic function for G. Simplify the BDD by using don't cares.*
3. *From the simplified BDD, produce the memory network (call it* **Network 2***). In general, Network 2 is simpler than Network 1 that realizes F.*
4. *When the query data is equal to the registered data, Network 2 produces correct outputs. When the query data does not match to any registered data, Network 2 may produce wrong output values.*
5. *To detect the correct outputs, we use an* **auxiliary memory** *with $\alpha = \lceil \log_2(k+1) \rceil$ inputs and n outputs. This auxiliary memory stores corresponding registered data for each address.*
6. *Apply the output address of Network 2 to the auxiliary memory, and read out the registered data in the auxiliary memory. If the output data of the auxiliary memory is equal to the input data, Network 2 produces the correct output value. If the output data of the auxiliary memory is different from the input data, then the input data is not registered. Thus, the circuit produces a special address (0).*
7. *Since the size of the auxiliary memory is $n2^{\alpha}$, the cost of memory is smaller compared with the cost of Network 2.*

An ordinary logic circuit can be simplified by *don't cares* [11, 14]. The present method has the following features:

- The number of non-zero outputs (k) of the address generator function is much smaller than the total number of input combinations m^n. In F, the outputs for the non-registered inputs are set to don't cares to produce G. By simplifying the network for G, we can greatly reduce the network.

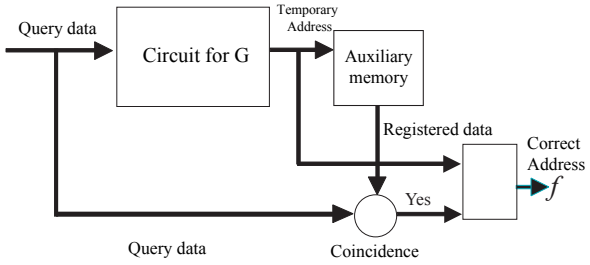

Figure 7.1. Address generator using auxiliary memory.

- To verify the correctness of the output of Network 2, we use an auxiliary memory.
- The size of the auxiliary memory is smaller than that of Network 2.

The total amount of hardware in the system is smaller than that of Network 1. In the logic synthesis using memories, reduction of support variables is important. In the address generation functions, the fraction of *don't cares* is very large, and we can often reduce the number of support variables.

Example 7.1 *Table 7.1 shows the registered vector table of an* 11*-input address generation function. Since the number of the registered vectors is* 15*, the address generator has* 11 *inputs and* 4 *outputs.*

Table 7.1. Registered vector table.

Address	*Registerd Vector* $x_1, x_2 \cdots x_{11}$
1	00100101100
2	00111110101
3	00111110110
4	01001100101
5	01001101111
6	10000100100
7	10001100101
8	10001101001
9	10001101111
10	10100001001
11	10100100100
12	11000110101
13	11001001111
14	11010000000
15	11010001001

Let the output values for the non-registered input vectors be don't care. Realize an LUT cascade.

Table 7.2. Truth table for auxiliary memory.

z_3	z_2	z_1	z_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}	y_{11}
0	0	0	1	0	0	1	0	0	1	0	1	1	0	0
0	0	0	0	0	0	1	1	1	1	1	0	1	0	1
0	0	1	1	0	0	1	1	1	1	1	0	1	1	0
0	1	1	0	0	1	0	0	1	1	0	0	1	0	1
0	1	0	1	0	1	0	0	1	1	0	1	1	1	1
0	1	1	0	1	0	0	0	0	1	0	0	1	0	0
0	1	1	1	1	0	0	0	1	1	0	0	1	0	1
1	0	0	0	1	0	0	0	1	1	0	1	0	0	1
1	0	0	1	1	0	0	0	1	1	0	1	1	1	1
1	0	1	0	1	0	1	0	0	0	0	1	0	0	1
1	0	1	1	1	0	1	0	0	1	0	0	1	0	0
1	1	0	0	1	1	0	0	0	1	1	0	1	0	1
1	1	0	1	1	1	0	0	1	0	0	1	1	1	1
1	1	1	0	1	1	0	1	0	0	0	0	0	0	0
1	1	1	1	1	1	0	1	0	0	0	1	0	0	1

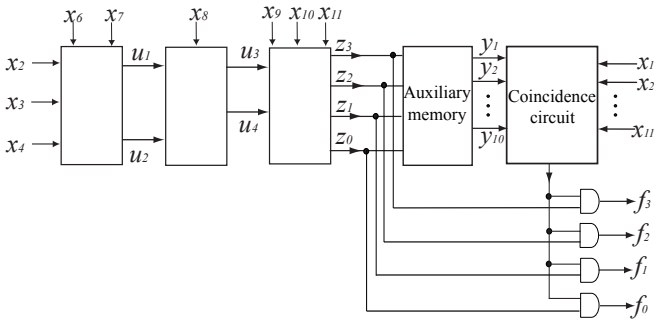

Figure 7.2. Address generator using auxiliary memory.

Fig. 7.2 shows the address generator using an auxiliary memory. Note that in Fig. 7.2, variables x_1 and x_5 are not used as inputs. Table 7.2 shows the content of the auxiliary memory. The third cell generates the temporary address (z_3, z_2, z_1, z_0).

For this address, we read the auxiliary memory, and compare the content $(y_1, y_2, \ldots, y_{11})$ with the input value $(x_1, x_2, \ldots, x_{11})$. If they agree, the temporary address is correct, and we produce $(f_3, f_2, f_1, f_0) = (z_3, z_2, z_1, z_0)$ as the output. Otherwise, the input query data is not in the auxiliary memory, and we produce $(f_3, f_2, f_1, f_0) = (0,0,0,0)$ as the output.

We designed an LUT cascade with the following conditions: Each cell has at most five inputs and at most four outputs, and the number of levels is three. Fig. 7.2 shows the cascade. Total amount of memory is 208 bits. Tables 7.3, 7.5, and 7.4 are truth tables for the cells. (End of Example)

Table 7.3. Truth table of the 1st cell.

$x_2x_3x_4x_6x_7$	u_1u_2
00000	00
00001	01
00010	10
00011	11
00100	01
00101	00
00110	00
00111	01
01000	01
01001	00
01010	00
01011	01
01100	00
01101	01
01110	10
01111	11
10000	01
10001	00
10010	00
10011	01
10100	00
10101	01
10110	10
10111	11
11000	00
11001	01
11010	10
11011	11
11100	01
11101	00
11110	00
11111	01

Table 7.4. Truth table of the 3rd cell.

$u_3u_4x_9x_{10}x_{11}$	$z_3z_2z_1z_0$
00000	0111
00001	0101
00010	0111
00011	0101
00100	1101
00101	0010
00110	1100
00111	1011
01000	0111
01001	1111
01010	0111
01011	0101
01100	1000
01101	0011
01110	0111
01111	1010
10000	0111
10001	0001
10010	0111
10011	0101
10100	0110
10101	1110
10110	0111
10111	1001
11000	0111
11001	0101
11010	0111
11011	0101
11100	0111
11101	0100
11110	1100
11111	1011

Table 7.5. Truth table of the 2nd cell.

$u_1u_2x_8$	u_3u_4
000	00
001	01
010	01
011	00
100	10
101	10
110	11
111	00

8 An Address Generator for an English-Japanese Dictionary

For simple English communication, we prepare a dictionary consisting of 1500 English words. To make a list of 1500 English words using a single memory or a single circuit is unrealistic. Therefore, we partition the list into three

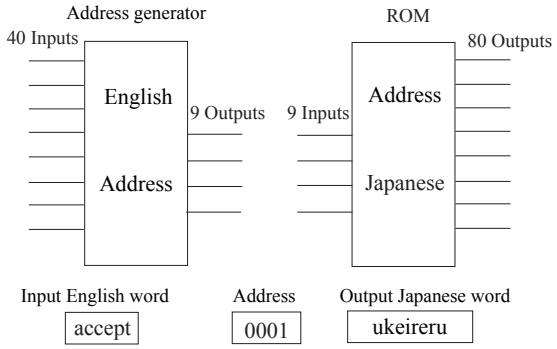

Figure 8.1. Implementation of English-Japanese dictionary.

groups, so that each list contains at most 500 words. Let the names of three lists be *Word list A*, *Word list B*, and *Word list C*. The maximum number of letters in the word lists is 13, but we only consider the first 8 letters. For English words consisting of fewer than 8 letters, we append blanks to make the length of words 8. We represent each alphabetic character by 5 bits. So, all the English words are represented by 40 bits. We assume that each group has at most 500 English words, and each word has unique address from 1 to 500. The address is represented by 9 bit.

Fig. 8.1 shows the English-Japanese dictionary consisting of the address generator and a ROM. In this dictionary, the address generator finds the address of the English word, and the ROM produces the Japanese translation. Note that in Japanese, 80 outputs are needed to represent the Chinese characters and *KANA* characters.

The size of the auxiliary memory is $n2^{\alpha}$, where $\alpha = \lceil \log_2(k+1) \rceil$, $k+1 = 501$ (number of words +1), and $n = 40$ (number of bits to represent an English word). Thus, we need $40 \times 2^9 = 20$ k bits.

Tables 8.1$\sim$ 8.3 compare the size of memories, where address generators are implemented either by a simple LUT cascade, or by a cascade and an auxiliary memory. As shown in these tables, address generators using auxiliary memories are about one fourth the size of simple cascade realizations. In this case, only 500 combinations out of 2^{40} input combinations of address generators are specified, and other combinations are all *don't cares*.

9 Conclusion

In this paper, we presented design methods for multiple-valued input address generators using memories. An address generator with k registered vectors can be implemented by memories with $\lceil \log_2(k+1) \rceil + 1$ inputs and $\lceil \log_2(k+1) \rceil$ outputs.

Table 8.1. Amount of memory (Word list A).

	Simple cascade realization	Simplified with DCs
Maximum number of rails	9	8
Number of cells	9	5
Total number of cell outputs	77	28
Size of auxiliary memory	0	20480
Total amount of memory	315392	91136

Table 8.2. Amount of memory (Word list B).

	Simple cascade realization	Simplified with DCs
Maximum number of rails	9	8
Number of cells	10	5
Total number of cell outputs	87	30
Size of auxiliary memory	0	20480
Total amount of memory	325384	89088

Table 8.3. Amount of memory (Word list C).

	Simple cascade realization	Simplified with DCs
Maximum number of rails	9	6
Number of cells	10	4
Total number of cell outputs	87	23
Size of auxiliary memory	0	20480
Total amount of memory	325384	83968

We also presented a method to simplify the address generator using an auxiliary memory. The outline of this method is

1. The output values for non-registered inputs are set to *don't cares*. Realize the incompletely specified logic function.
2. Verify the output of the address generator by an auxiliary memory. If the output is correct, produce it as it is, otherwise the outputs is set to zero.

Address generators can be efficiently implemented by memories. For this application, we can also use CAMs. However CAMs dissipate more powers than ordinary SRAMs[12].

Acknowledgments

This research is supported in part by the Grants in Aid for Scientific Research of JSPS, and the grant of Kitakyushu Innovative Cluster Project. Discussion with Prof. Jon T. Butler improved English presentation. Mr. Matsuura did experiments.

References

[1] C. Akrout, et.al, "Reprogrammable logic fuse based on a 6-device SRAM cell for logic arrays," US Patent 5063537.

[2] L. Chisvin and R. J. Duckworth, "Content-addressable and associative memory: Alternatives to the ubiquitous RAM, " *IEEE Computer*, Vol. 22, pp. 51-64, July 1989.

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

[4] C. H. Divine and J. C. Moran, "Memory patching circuit with repatching capability," US Patent 4028684.

[5] L. Fujino, "Digital Archive 2004 of the IEEE International Symposium on Multiple-Valued Logic," (DVD-ROM), *IEEE Computer Society, Technical Committee on Multiple-Valued Logic*, May 2004.

[6] 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.

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

[8] P-F. Lin and J. B. Kuo, "A 1-V 128-kb four-way set-associative CMOS cache memory using wordline-oriented tag-compare (WLOTC) structure with the content-addressable-memory (CAM) 10-transistor tag cell," *IEEE Journal of Solid-State Circuits*, Vol. 36, pp. 666 - 675, April 2001.

[9] J. C. Moran, "Memory patching circuit," US Patent 4028678.

[10] M. Motomura et al., "A 1.2-million-transistor, 33-MHz, 20-b dictionary search processor (DISP) ULSI with a 160-Kbyte CAM, " *IEEE J. Solid-State Circuits*, Vol. 25, No. 5, Oct. 1990, p. 1158- 1165.

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

[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 and T. Sasao, "Design of address generators using multiple LUT cascade on FPGA," SASIMI 2006, April 3-4, 2006 (to be published).

[14] T. Sasao, *Switching Theory for Logic Synthesis*, Kluwer Academic Publishers, 1999.

[15] T. Sasao and M. Matsuura, "BDD representation for incompletely specified multiple-output logic functions and its applications to functional decomposition, " *Design Automation Conference*, June 2005, pp.373-378.

[16] T. Sasao and J. T. Butler, "Implementation of multiple-valued CAM functions by using LUT cascade, " *International Symposium on Multi-Valued Logic*, May 17-20, 2006, (to be published).

[17] F. Zane, G. Narlikar, and A. Basu, "CoolCAM: Power-efficient TCAMs for forwarding engines", *Proceeding of IEEE INFOCOM '03*, April, 2003.