Optimization Methods in Look-Up Table Rings

Tsutomu SASAO Masaki KUSANO Munehiro MATSUURA

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

April 27, 2004

Abstract

A Look-Up Table (LUT) ring consists of memories, programmable interconnections and a control circuit. It sequentially emulates an LUT cascade representing a multiple-output logic function. In this paper, we consider the realization of multi-output functions with LUT rings using large memories. In contrast to previous approaches where the number of inputs to each LUT cell is fixed, we allow the number of inputs to be different for each cell. With this new approach, we can reduce the number of levels and the total amount of memory by selecting the optimal size for each cell and by packing the memory. We have developed an optimization system for LUT ring designs using dynamic programming. In trials, our system was able to reduce the amount of memory required for some designs by as much as 60%.

1 Introduction

This paper considers a realization of a multiple-output function by using a large memory. Such realizations are useful for reconfigurable applications. Several methods exist to implement logic functions by using large memories.

1) **Direct Method**. This method directly implements logic functions by a memory. To implement an n-input m-output function, we need a memory with $m2^n$ bits, which is impractical when n is large.
2) **Memory and Microprocessor**. This method uses a general-purpose microprocessor and a memory. First, it represents the given logic functions by a netlist of random logic circuit of gates. Then, it uses an existing logic simulator to evaluate the function. Both the data for the netlist and the simulation program are stored in the memory. With this method, the cost for the development is low, but power dissipation is high relative to the performance.
3) **Branching Program Machine** [2, 3]. This method uses a dedicated circuit to evaluate logic functions instead of a general purpose microprocessor. First, it represents a given logic function by a decision diagram (DD). Then, it evaluates the function by traversing the DD using a dedicated circuit. This method stores only the data for the DD in the large memory. Since it has no instruction fetch, it is faster and dissipates less power than the method using a general microprocessor. To evaluate an n-variable function, this method requires $O(n)$ memory references.
4) **Murgai-Hirose-Fujita's Method** [4]. This method uses a dedicated event-driven logic emulator. Instead of a DD, this method uses the netlist of a multi-level random logic network of large look-up tables (LUTs) to represent the logic function. This method stores the LUT data in a large memory. It also uses another memory to store the netlist of the LUT network. The computation time is proportional to the number of LUTs in the network.
5) **Look-up Table Ring** [6]. This method first represents the logic function by a BDD, then transforms it into an LUT cascade. And, finally it emulates the cascade by an LUT ring. This method stores the LUT data in a large memory. In this method, the structure of the circuit is a cascade rather than random logic, so the control part is simpler and faster than Murgai-Hirose-Fujita's method. Also, logic synthesis is simpler. This method is faster than the branching program machine, since there are fewer memory references.

Table 1 compares the various methods of implementing logic functions with large memories.

In this paper, we consider optimization techniques for LUT rings. The new techniques are as follows:

1) To find better LUT cascade, we use a new decomposition method for multiple-output functions [7]. We use a BDD for characteristic function to find the decomposition with intermediate outputs.

$X_2=(x_3,x_4)$ \ $X_1=(x_1,x_2)$	0 0	0 1	1 0	1 1
0 0	0	1	1	0
0 1	1	1	1	1
1 0	0	1	1	0
1 1	0	0	0	0

Figure 1: Decomposition table for four-variable function.

2) To increase the speed and to reduce the size of memory, we use LUTs with different number of inputs. We use a dynamic programming approach to find the optimal solutions.
3) To reduce the size of memory, we use memory packing.

An LUT ring was called an LUT cascade in [6]. However, in this paper, the sequential circuit that emulates an LUT cascade will be called an **LUT ring**.

2 Definitions and Basic Properties

This section introduces terminology used in the paper.

2.1 Functional Decomposition

Let $X=(x_1,x_2,\ldots,x_n)$ be an ordered set of the input variables. Let $\{X_1\}$ be an unordered set of the variables in X. $(X_1,X_2,\ldots,X_s)$ is a **partition of** X if $\{X_1\}\cup\{X_2\}\cup\cdots\cup\{X_s\}=\{X\}$ and $\{X_1\}\cap\{X_2\}\cap\cdots\cap\{X_s\}=\phi$. A partition is a bipartition if $s=2$. The number of variables in X is denoted by $|X|$.

Given a logic function $f(X)$ and a bipartition (X_1,X_2) of X, consider the table with $2^{|X_1|}$ columns and $2^{|X_2|}$ rows. For each column and each row, assign a distinct binary number as a label, and let the value of the corresponding element be the value of f. Such a table is a **decomposition chart**. The number of the different column patterns in the decomposition chart is the **column multiplicity**, denoted by μ. Fig. 1 shows an example of a decomposition chart for a four-variable function. In this example, $\mu=2$. Let $(X_1,X_2,\ldots,X_s)$ be a partition of X. Then, $f(X)$ can be represented as

$$f(X)=g(h_1(X_1),h_2(X_1),\ldots,h_u(X_1),X_2). \qquad (1)$$

The representation of form (1) is called a **decomposition** of f. In this case, $f(X)$ can be realized by the network shown in Fig. 2, where $u=\lceil\log_2\mu\rceil$. In the decomposition, X_1 and X_2 are called **bound variables** and **free variables**, respectively.

2.2 Characteristic Function

Let $X=(x_1,x_2,\ldots,x_n)$ be input variables. Let $F=(f_1(X),f_2(X),\ldots,f_m(X))$ be a multiple-output function. A

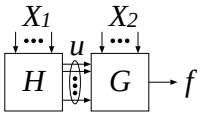

Figure 2: Realization of function f by decomposition.

characteristic function of a multiple-output function is

$$\chi(X,Y)=\bigwedge_{i=1}^{m}(y_i\equiv f_i(X)), \qquad (2)$$

where y_i $(i=1,2,\ldots,m)$ are variables that denote outputs. The characteristic function for an n-input m-output function is a two-valued logic function with $(n+m)$ variables. In this case, in addition to the input variables x_i $(i=1,2,\ldots,n)$, we use output variables y_j for f_j $(j=1,2,\ldots,m)$. Let $B=\{0,1\}$, $\vec{a}\in B^n$, $\vec{b}\in B^m$, and $F(\vec{a})=(f_1(\vec{a}),f_2(\vec{a}),\ldots,f_m(\vec{a}))\in B^m$. Then, we have

$$\begin{aligned}\chi(\vec{a},\vec{b}) &= 1 \quad (\text{If } \vec{b}=F(\vec{a}))\\ &= 0 \quad (\text{Otherwise}).\end{aligned}$$

2.3 BDD and Functional Decomposition [7]

A binary decision diagram for characteristic function (BDD-for-CF) of a multiple-output function $F=(f_1,f_2,\ldots,f_m)$ is the BDD representing the characteristic function χ for F. We assume that in the BDD, variable y_i appears in a position lower than the variables that influence f_i.

When a logic function is represented by a BDD, the number of different nodes for the free variables that are directly connected from the nodes of bound variables is equal to the column multiplicity. Next, we consider the functional decomposition of a BDD-for-CF. Let (X_1,X_2) be the set of input variables, and let (Y_1,Y_2) be the set of output variables. Let (X_1,Y_1,X_2,Y_2) be the ordering of the BDD-for-CF. Let μ be the column multiplicity for the BDD-for-CF of the decomposition (X_A,X_b), where $X_A=(X_1,Y_1)$ and $X_B=(X_2,Y_2)$. In this case, to compute μ by the BDD, we ignore the edges between constant 0 nodes and output nodes. When we realize a multiple-output function by the network shown in Fig. 3, the necessary and sufficient number of lines between two blocks is $\lceil\log_2\mu\rceil$. The outputs of H that are connected to G are **intermediate variables**.

By applying the decompositions $(s-1)$ times, we have an LUT cascade shown in Fig. 4. Let k_i be the number of inputs to the i-th cell. Then, we have $k_i=|X_i|+u_{i-1}$.

Table 1: Comparison of various methods.

Method	Data Structure	Evaluation Method	Evaluation Time
Direct	Truth table	Decoder	Constant
Memory and MPU	Random logic of gates	Software simulator	O(# Gates)
Branching Program Machine	Decision diagrams	Special hardware	O(n)
Murgai-Hirose-Fujita's	Random logic of LUTs	Special hardware	O(# LUTs)
LUT Ring	Cascade of LUTs	Special hardware	O(n)

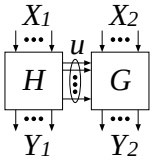

Figure 3: Decomposition with intermediate outputs.

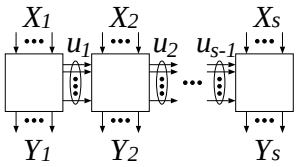

Figure 4: LUT cascade.

3 LUT Cascades and LUT Ring

3.1 LUT Cascade

An **LUT cascade** is shown in Fig. 4, where multiple-output LUTs (cells) are connected in series to realize a multiple-output logic function. The wires connecting adjacent cells are called **rails**. Let k_i be the number of inputs to the i-th cell, and let u_i be the number of **rail outputs** of the i-th cell, i.e., the number of the rails between i-th cell and $(i+1)$-th cell. Let $|Y_i|$ be the number of the **external outputs** of the i-th cell, i.e., the outputs that are connected to the primary output terminals. Let s be the number of cells in a cascade. The **size** of the i-th cell is $2^{k_i} \cdot u_i$. The total amount of memory necessary to implement the cascade is

$$L(X_1, Y_1, X_2, Y_2, \ldots, X_s, Y_s) = \sum_{i=1}^{s} 2^{k_i} \cdot (u_i + |Y_i|). \qquad (3)$$

The LUT cascade is simple and fast, but the restricted nature of its interconnections means it is not so flexible. Once the numbers of rails, inputs and outputs of cells, and the number of the cells are fixed, the number of functions realizable in the cascade is limited.

3.2 LUT Ring

In Fig. 4, by adding feedback lines between Y_s and X_1, we have an LUT ring. An **LUT Ring with a single unit** is shown in Fig. 5. It sequentially emulates an LUT cascade. Although it is slower than the LUT cascade, it has much more flexibility. In the LUT ring, the numbers of rails, inputs and outputs of cells, and the number of cells are flexible. We can consider an LUT ring with multiple units. However, for simplicity, in this paper, we will consider only the LUT ring with a single unit.

In the LUT ring, all the data for the cells are stored in a memory. The **Input Register** stores the values of the primary inputs; the **MAR** (Memory Address Register) stores the address of the memory; the **MBR** (Memory Buffer Register) stores the values of the outputs of the memory; the **Memory for Logic** stores the content of cells in the cascades; the **Programmable interconnection** connects between the Input register and the MAR, and also between the MBR and the MAR; the **Memory for Interconnection** stores method for interconnections; and the **Control** obtains functional values by sequentially accessing the memory.

We can formulate the design problem for an LUT ring as follows:

Problem 1 *Given a multiple-output function $F = (f_1(X), f_2(X), \ldots, f_m(X))$ and the ordering of the input variables X, obtain the partition of X that satisfies the following conditions:*

1. *The total amount of memory is at most L_0.*
2. *The number of cells of the cascade is the minimum subject to condition 1.*
3. *The total amount of memory is the minimum subject to condition 2.*

In an LUT ring, all the least significant k bits of the starting address for k-input cells should be zeros. For example, the starting address of a 10-input cell in a 32-kilo-word memory should have the form xxxxx0000000000. Thus, the amount of memory actually needed to implement the LUT ring may be larger than the value obtained by equation (3).

We can reduce the number of levels of the cascade and/or the total amount of memory by using cells with different numbers of inputs and/or by **memory packing**.

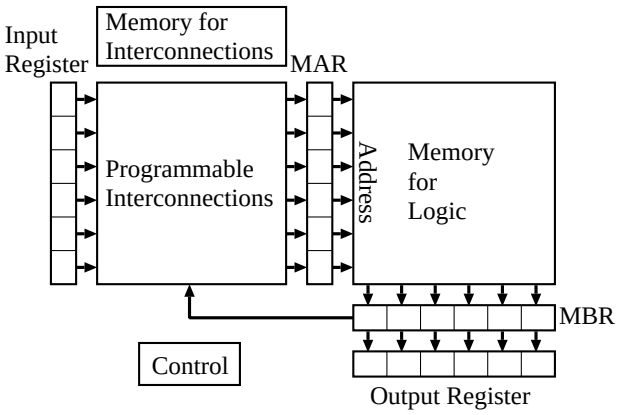

Figure 5: LUT Ring with a single unit.

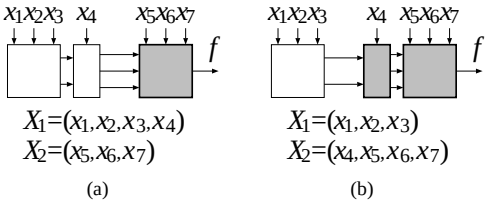

Figure 6: Two different LUT cascades.

4 Reduction using Cells with Different Number of Inputs

4.1 Principle of Reduction

A straightforward method to design an LUT ring is to use cells with the same numbers of inputs. However, we can often reduce the total amount of memory by using cells with different numbers of inputs. In fact, we can often reduce the number of inputs to a cell without increasing the number of inputs of other cells.

Example 1 *Consider the LUT cascade shown in Fig. 6. Suppose that the input variables are partitioned into* (X_1, X_2)*, where* $X_1 = (x_1, x_2, x_3, x_4)$ *and* $X_2 = (x_5, x_6, x_7)$*, as shown in Fig. 6(a). In this case, the first block has 4 inputs and 3 outputs, and the second block has 6 inputs and 1 output.*

Next consider the case, where the input variables are partitioned into (X_1, X_2)*, where* $X_1 = (x_1, x_2, x_3)$ *and* $X_2 = (x_4, x_5, x_6, x_7)$ *as shown in Fig. 6(b). In this case, the first block has 3 inputs and 2 outputs, and the second block has 6 inputs and 1 output. Therefore, the second realization of this function requires less memory than the first realization.* (*End of Example*)

The next theorem generalizes the above example.

Theorem 1 *Let* $(X_1, X_2, \ldots, X_s)$ *be a partition of the variables* X*, and let* u_i *be the number of the rail outputs of the* i*-th cell. Suppose that one variable* x_l *is moved from* X_i *to* X_{i+1}*. If* u_i *is reduced by one, then we can reduce the total amount of memory for the cascade.*

(Proof) By the hypothesis of the theorem, the numbers of inputs and outputs for the i-th cell are reduced by one by the move of the variable. Let k_i be the number of inputs for the i-th cell. Then, the amount of memory for the i-th cell is reduced from $2^{k_i} \cdot u_i$ to $2^{k_i - 1} \cdot (u_i - 1)$. On the other hand, the size of the $(i+1)$-th cell remain unchanged. This is because the number of the rail outputs of the i-th cell is reduce by one, but one external variable x_l is appended to the inputs of the $(i+1)$-th cell. Also, the sizes of other cells do not change. Hence, we have the theorem. (Q.E.D.)

By using this theorem as well as other technique, we can reduce the total amount of memory and the number of levels by changing the partition X. In the next section, we will show an algorithm that solves Problem 1 with a dynamic programming approach.

4.2 Algorithm to Find a Partition of X That Minimizes the Number of Levels

Consider a BDD-for-CF for an n-input m-output function. Let $Z = (z_1, z_2, \ldots, z_{n+m})$ be the set of input and output variables. Let the height of the root node be $n+m$, and let the height of the constant node be 0. Let μ_{z_i} be the column multiplicity with respect the partition (Z_A, Z_B), where $Z_A = (z_1, z_2, \ldots, z_{i-1})$ and $Z_B = (z_i, z_{i+1}, \ldots, z_{n+m})$. Let $u_{z_i} = \lceil \log_2 \mu_{z_i} \rceil$. Let $mem(z_i)$ be the amount of memory for the cells of the cascade from z_1 up to the variable z_i. Let $s(z_i)$ be the number of cells in the cascade from z_1 up to the variable z_i. Let $mem\_opt(z_i)$ and $s\_opt(z_i)$ be the minimum memory and minimum cells of the optimal solutions found so far, respectively. Let k be the maximum number of inputs of cells.

Algorithm 1 *Fig. 7 shows the pseudo-code to find an LUT ring with the minimum number of cells by dynamic programming. In this algorithm, the maximum number of inputs of cells are restricted to k.*

The 7th and 8th lines of Algorithm 1 generate the partition. The 11th line checks if the number of inputs is equal to or less than the maximum number allowed. The lines after 13 check if the partition is optimum or not.

Algorithm 1 constructs optimum LUT cascades with two cells, three cells, and so on, sequentially, and finally, it finds the optimum LUT cascade with s cells. It finds the cascade with the minimum number of levels and the minimum amount of memory by dynamic programming.

```
1  dyna_cascade(BDD,n,m,k) {
2    for(i ← n+m; i > 0; i ← i−1){
       (Compute the number of the rail outputs)
3      Compute μ_{z_i} and u_{z_i} ← ⌈log_2 μ_{z_i}⌉;
4      mem(i) ← 0;
5    }
6    mem(n+m+1) ← 0, s(n+m+1) ← 0, u_{n+m+1} ← 0;
7    for(i ← n+m; i > 1; i ← i−1){
8      for(j ← i−1; j > 0; j ← j−1){
9        p ←(The number of input variables in (z_i, z_{i−1}, ..., z_j))
10       q ←(The number of output variables in (z_i, z_{i−1}, ..., z_j))
11       if(p + u_{z_{i+1}} ≤ k){
12         mem(j) ← mem(i) + (u_{z_j} + q)2^{p+u_{z_{i+1}}}, s(j) ← s(i) + 1;
13         if(s(j) < s_opt(j)){(update the optimum level)
14           s_opt(j) ← s(j);
15           mem_opt(j) ← mem(j);
16         }
17         else if(s(j) = s_opt(j) & mem(j) < mem_opt(j)){
             (update the optimum memory size)
18           mem_opt(j) ← mem(j);
19         }
20       }
21     }
22   }
23 return mem_opt(1), s_opt(1)
24 }
```

Figure 7: Algorithm 1: Pseudo-code to find an LUT ring with the minimum number of cells.

5 Reduction by Memory Packing

In an LUT ring, the data of the cells is stored entirely in the memory for logic. In this case, we can reduce the necessary amount of memory by **memory packing**.

5.1 Principle of Memory Packing

We will illustrate the idea of memory packing by an example.

Example 2 *Fig. 8 shows an LUT cascade for an 11-input 3-output function, where 4-input cells are used. Fig. 9 shows the memory mapping of cell data, where the memory has 6-bit address inputs, and each word consists of four bits.* $(A_0, A_1, \ldots, A_5)$ *in Fig. 9 denotes the address, where* (A_0, A_1) *denotes the page number.* (D_0, D_1, D_2, D_3) *denotes the outputs of the memory. The dark areas in the figure are unused. In Fig. 9, only data for a single cell is stored in each page. Note that half of the memory area in Fig. 9 is unused. By moving the cell data in page 3 to the* D_3 *part of page 1, and the cell data for page 2 to the* D_3 *and* D_2 *parts of page 0, we can reduce the necessary amount of memory by half.* (*End of Example*)

In an LUT ring, the data of a cell must be read simultaneously. Thus, the data for each cell must be stored in the same page of the memory. However, if there is any vacancy, the data for multiple cells can be stored in the same page. By using this property, we can reduce the required amount of memory. This is called **memory packing**. To implement memory packing, we need a **shifter** that shifts the bits between MBR and MAR and a mapping memory.

Algorithm 1 produces the LUT cascade with the minimum number of cells. Note that the numbers of inputs for cells can be different. Cells with different numbers of inputs require different numbers of address lines in an LUT ring. In Fig. 10, data for two cells are stored in the same page. In this case, only three bits are necessary to specify the address. We need a circuit to supply constants to the MAR. In the following, we present a heuristic algorithm to reduce the necessary amount of memory by packing.

5.2 Algorithm for Memory Packing

Algorithm 2 *Let* w_i *be the number of outputs of the i-th cell, where* $i = 1, 2, \ldots, s$. *Assume that the word length of the memory is at least* $w = \max_i\{w_i\}$. *Recall that the i-th cell has* u_i *rail outputs and* $|Y_i|$ *external outputs. So,* $w_i = u_i + |Y_i|$.

1. *Reorder the cells in descending order of the numbers of inputs. For the cells with the same numbers of inputs, reorder them in descending order of the numbers of outputs. Let* $v_1, v_2, \ldots, v_s$ *be the numbers of outputs of the cells.*
2. $i \leftarrow 1$.
3. *If* $(i = s)$ *then stop the algorithm else* $j \leftarrow i+1$.
4. *Check if* v_j *outputs of j-th cell can be moved to the i-th page. If possible, move them and go to step 5, otherwise, go to step 6.*
5. *If unused area remains in the i-th cell then go to step 7 else go to step 8.*
6. *If* $j < s$ *then go to step 7 else go to step 8.*
7. $j \leftarrow j+1$ *and go to step 4.*
8. $i \leftarrow i+1$ *and go to step 3.*

Example 3 *Fig. 11 illustrates memory packing. First, the cells are reordered in descending order of the number of the inputs (Fig. 11(b)). Then, they are reordered in descending order of the number of the outputs (Fig. 11(c)). Then, the cell for* g_1 *is moved to the first page (Fig. 11(d)). Finally, the cells for* g_{10}, g_{11} *and* f *are moved into the second page (Fig. 11(e)).* (*End of Example*)

6 Experimental Results

We implemented the algorithms in C, and applied them to MCNC benchmark functions. First, we obtained the partitions where all the cells have the same numbers of inputs

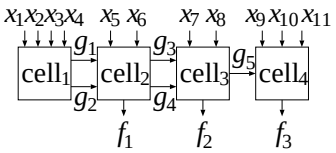

Figure 8: Example of LUT cascade.

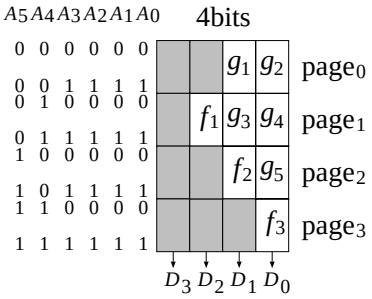

Figure 9: Memory mapping of LUT data.

(*Case 1*). Second, we obtained the partitions by Algorithm 1, where cells can have different numbers of inputs to reduce the total amount of memory (*Case 2*). Third, we obtained the partitions where cells can have different numbers of inputs to make the numbers of levels minimum (*Case 3*). We implemented the cascade so that the data may fit into a memory with one mega bits (i.e., 64 kilo words $\times$ 16 bits).

For each case, we packed memory by Algorithm 2. Table 2 compares three cases with and without memory packing. In the table, *Name* denotes the name of the function; *In* denotes the number of inputs; *Out* denotes the number of outputs; *k* denotes maximum number of inputs of cells; *s* denotes the number of cells in the cascade; *Memory* denotes the amount of memory (mega bits); *non-pack* denotes the case without memory packing; and *pack* denotes the case with memory packing. Experiments were done in the following environment: CPU: Pentium4 Xeon 2.8GHz, L1 Cache: 32KB, L2 Cache: 512KB, Memory: 4GB, OS: RedHad Linux 7.3, Compiler: gcc version 2.96.

In an LUT cascade, we can reduce the number of cells by increasing the total amount of memory. In *Case 1*, we

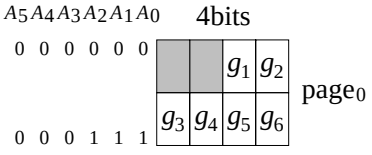

Figure 10: Memory mapping for the cells with different numbers of inputs.

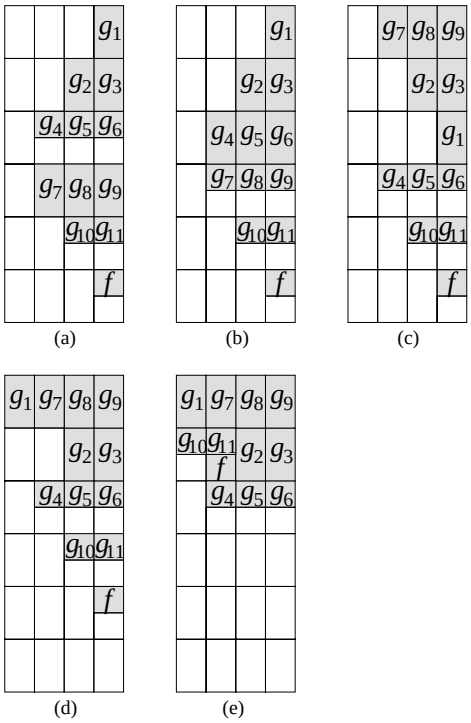

Figure 11: Example of memory packing.

selected the value of *k* that produces a cascade with the minimum number of levels, under the condition that all the cells fit in a 1-Mega-bit RAM. In *Case 2*, we used the same *k* as *Case 1*, and obtained the partitions that minimize the total amount of memory and the number of cells. As shown in Table 2, for most functions, *Case 2* required less memory than *Case 1*. As for the computation time, the most CPU time was spent for the optimization of BDD-for-CFs. In Table 2, the most time-consuming one was *k2*, which took 63.7 seconds.

Fig. 12 shows the cascade for the benchmark function *C432*, where $k = 15$. The number of levels (cells) is four, and the amount of memory after packing is 0.75 Mega bits. By using cells with different numbers of inputs, we could reduce the amount of memory for the LUT ring. Next, we compare *Case 1* with *Case 3*. For some benchmark functions, we could reduce the number of levels by using cells with more inputs than in *Case 1*.

Fig. 13 shows cascades for the benchmark function *misex2*. When $k = 14$, we produced the cascade with four cells, while when $k = 16$, we produced a cascade with only three cells. Note that the cells have different numbers of inputs in both cascades. On the other hand, if we used the cells with the same numbers of inputs $k = 16$, then we could not realize the cascade using a 1-Mega-bit memory.

Table 2: The amount of memory for LUT rings to realize benchmark functions.

Name	In	Out	Case 1 Cells with the Same numbers of inputs				Case 2 Minimal Memory Different numbers of inputs				Case 3 Minimum Levels Different numbers of inputs			
					Memory				Memory				Memory	
			k	*s*	non-pack	Pack	*k*	*s*	non-pack	Pack	*k*	*s*	non-pack	Pack
C432	36	7	15	4	2.000	1.000	15	4	1.250	0.750	15	4	1.250	0.750
apex1	45	45	13	10	1.250	1.000	13	10	0.721	0.596	13	10	0.721	0.596
apex2	39	3	15	3	1.000	0.500	15	3	1.000	0.500	15	3	1.000	0.500
apex3	54	50	12	16	0.938	0.625	12	16	0.666	0.416	13	13	1.135	0.760
comp	32	3	12	3	0.051	0.016	12	3	0.051	0.016	12	3	0.051	0.016
duke2	22	29	14	3	1.000	0.750	14	3	0.376	0.376	14	3	0.376	0.376
e64	65	65	13	6	0.750	0.750	13	6	0.313	0.313	13	6	0.313	0.313
k2	45	45	13	10	1.125	1.000	13	10	0.721	0.596	13	10	0.721	0.596
misex2	25	18	14	3	0.750	0.500	14	3	0.078	0.068	16	2	0.750	0.750
seq	41	35	13	8	1.000	0.625	13	8	0.626	0.376	15	7	1.113	0.751
vg2	25	8	13	3	0.375	0.250	13	3	0.250	0.125	16	2	1.500	1.000
x6dn	39	5	13	5	0.625	0.250	13	5	0.375	0.188	16	4	1.125	0.625
ratio					1	0.669			0.592	0.398			0.926	0.647

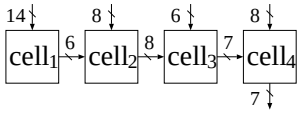

Figure 12: LUT cascade for C432.

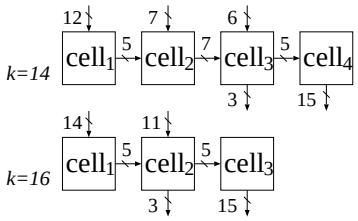

Figure 13: LUT cascade for *misex2*.

7 Conclusion

In this paper, we presented design methods for LUT rings. By using cells with different numbers of inputs, and by packing the memory, we could reduce the amount of memory by 60%, on the average of original memory sizes. To implement an LUT ring, we need a *Memory for Interconnection* that stores the connection information. To do memory packing, we need a shifter and a mapping memory. However, the amount of additional hardware is much smaller than the *Memory for Logic* in Fig. 5. Thus, our method effectively reduces the total chip size and increases the performance.

In Table 2, we showed only the functions, where each of them we could realize by single cascade. For the functions with more inputs and/or more outputs, we have to partition the outputs into groups, and realize them by separate cascades. Such cascades can also be emulated by the architecture shown in Fig. 5. Details of the results will be reported by a separate paper.

8 Acknowledgments

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

References

[1] R. K. Brayton, "The future of logic synthesis and verification," in S. Hassoun and T. Sasao (eds.), *Logic Synthesis and Verification*, Kluwer Academic Publishers, 2002.

[2] Y. Iguchi, T. Sasao, M. Matsuura, and A. Iseno, "A hardware simulation engine based on decision diagrams," *Asia and South Pacific Design Automation Conference (ASP-DAC'2000)*, Jan. 26-28, Yokohama, Japan, pp. 73-76.

[3] Y. Iguchi, T. Sasao, and M. Matsuura, "Implementation of multiple-output functions using PROMDDs," *30th International Symposium on Multiple-Valued Logic*, Portland, Oregon, U.S.A., May 23 - 25, 2000, pp. 199-205.

[4] R. Murgai, F. Hirose, and M. Fujita, "Logic synthesis for a single large look-up table," *Proc. International Conference on Computer Design*, pp. 415-424, Oct. 1995.

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

[6] T. Sasao, M. Matsuura, and Y. Iguchi, "A cascade realization of multiple-output function for reconfigurable hardware," *International Workshop on Logic Synthesis (IWLS-2001)*, Lake Tahoe, CA, June 12-15, 2001, pp. 225-300.

[7] T. Sasao and M. Matsuura, "A method to decompose multiple-output logic functions," *41st Design Automation Conference*, June 2004, (accepted for publication).