IEICE TRANS. FUNDAMENTALS, VOL.E88–A, NO.12 DECEMBER 2005

PAPER Special Section on VLSI Design and CAD Algorithms

A Design Algorithm for Sequential Circuits Using LUT Rings

Hiroki NAKAHARA[†a)], ***Student Member***, **Tsutomu SASAO**[†b)], ***and*** **Munehiro MATSUURA**[†c)], ***Members***

SUMMARY This paper shows a design method for a sequential circuit by using a Look-Up Table (LUT) ring. The method consists of two steps: The first step partitions the outputs into groups. The second step realizes them by LUT cascades, and allocates the cells of the cascades into the memory. The system automatically finds a fast implementation by maximally utilizing available memory. With the presented algorithm, we can easily design sequential circuits satisfying given specifications. The paper also compares the LUT ring with logic simulator to realize sequential circuits: the LUT ring is 25 to 237 times faster than a logic simulator that uses the same amount of memory.

key words: *reconfigurable architecture, LUT cascade, BDD_for_CF, functional decomposition*

1. Introduction

The design of an LSI requires long time, since the number of transistors in an LSI is often greater than 10^7. In addition, the LSI design have deep-submicron (DSM) effects, such as cross-talk noise and inductive effects that require electro-magnetic design. To solve these problems, regular and reconfigurable architectures have been considered. Regular architectures have repeated structures, hence the overall structure at the global level is uniform. Such a structure is more predictable in its delays. A repeated pattern can be hand-designed and extensively analyzed to avoid internal DSM problems, since its scale is relatively small and needs to be designed only once [3]. Reconfigurable architectures is rewritable, and can reduce the hardware development time drastically.

Memory is the most important device that is regular and reconfigurable. We present a Look-Up Table (LUT) Ring that consists of memories are reconfigurable. An LUT ring consists of memories, a control circuit, registers and a programmable interconnection network. It sequentially emulates an LUT cascade that represents the state transition functions and the output functions. The LUT ring is faster than the logic simulator, since the number of memory references can be reduced.

The outline of the design method for a sequential circuit by using the LUT ring is as follows:

1. Represent the state transition functions and the output functions by multiple Binary Decision Diagrams [4] (BDDs).
2. Transform these BDDs into a set of LUT cascades.
3. Allocate the LUT data into the memory of the LUT ring.

We show a memory packing method that efficiently allocates the LUT data into the memory of the LUT ring. Also, we show a method to represent BDDs by partitioning the outputs into several groups. The design of a fast LUT ring for a given size of memory is quite complex. It is similar to the case of an ordinary LSI design, where logic design and physical design interact. In our design method, the parameters given by the user are the total amount of memory and the number of memory inputs and outputs: the optimization for speed is automatically done. It derives a high-speed sequential circuit by maximally utilizing memories on the LUT ring.

We also compare the LUT ring with other methods to realize sequential circuits. The rest of the paper is organized as follows: Sect. 2 introduces representation of logic functions. Section 3 shows the architectures of LUT cascades and LUT rings. Section 4 shows design algorithms for an LUT ring. Section 5 shows experimental results. Finally, Sect. 6 concludes the paper.

This paper builds on the previous publications [7], [10], [12].

2. Representation of Logic Functions

Various methods exist to represent multiple-output logic functions. Among them, MTBDD [5] (multi-terminal BDD), BDD_for_ECFN [9] (BDD for encoded characteristic function for non-zero outputs), and BDD_for_CF [2] (BDD for characteristic function) are suitable for the design of LUT cascades and LUT rings. In [8], the evaluation time and the amount of memory of these decision diagrams are analyzed. They have the following features:

- MTBDD: The width tends to be too large to realize LUT cascades.
- BDD_for_ECFN: The width is smaller than the corresponding MTBDD, but its evaluation time is proportional to the number of outputs.
- BDD_for_CF: The width is smaller than the corresponding MTBDD, and many outputs can be evaluated efficiently.

Manuscript received March 11, 2005.
Manuscript revised June 3, 2005.
Final manuscript received August 11, 2005.

[†]The authors are with the Department of Computer Science and Electronics, Kyushu Institute of Technology, Iizuka-shi, 820-8502 Japan.
a) E-mail: nakahara@aries01.cse.kyutech.ac.jp
b) E-mail: sasao@cse.kyutech.ac.jp
c) E-mail: matsuura@cse.kyutech.ac.jp

DOI: 10.1093/ietfec/e88–a.12.3342

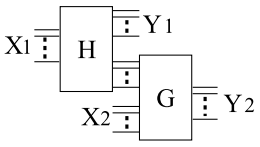


Fig. 1 Functional decomposition with intermediate outputs.

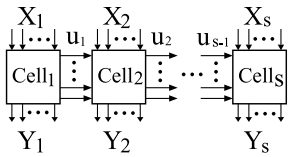


Fig. 2 LUT cascade.

Therefore, in this paper, we use a BDD_for_CF to represent a multiple-output logic function.

Definition 2.1: Let $\vec{X} = (x_1, x_2, \ldots, x_n)$ be the input variables, $\vec{Y} = (y_1, y_2, \ldots, y_m)$ the output variables, and $\vec{f} = (f_1(\vec{X}), f_2(\vec{X}), \ldots, f_m(\vec{X}))$ be a multiple-output function. **The characteristic function of a multiple-output function** is $\vec{\chi}(\vec{X}, \vec{Y}) = \bigwedge_{i=1}^{m} (y_i \equiv f_i(\vec{X}))$.

The characteristic function of an n-input m-output function is a two-valued logic function with $(n + m)$ inputs. It has input variables x_i $(i = 1, 2, \ldots, n)$, and output variables y_j for output f_j. Let $B = \{0, 1\}$, $\vec{a} \in B^n$, $\vec{F} = (f_1(\vec{a}), f_2(\vec{a}), \ldots, f_m(\vec{a})) \in B^m$, and $\vec{b} \in B^m$. Then, the characteristic function satisfies the relation

$$\vec{\chi}(\vec{a}, \vec{b}) = \begin{cases} 1 & (when\ \vec{b} = \vec{F}(\vec{a})) \\ 0 & (otherwise) \end{cases}$$

Definition 2.2: A support variable of a function f is a variable on which f actually depends.

Definition 2.3: [2] **The BDD_for_CF** of a multiple-output function $\vec{f} = (f_1, f_2, \ldots, f_m)$ is the ROBDD for the characteristic function $\vec{\chi}$. In this case, we assume that the root node is in the top of the BDD, and the variable y_i is below the support variable of f_i, where y_i is the variable representing f_i.

Definition 2.4: The width of the BDD_for_CF at height k is the number of edges crossing the section of the graph between x_k and x_{k+1}, where the edges incident to the same nodes are counted as one. Also, in counting the width of the BDD_for_CF, we ignore the edges which indicate the constant 0 node.

Let X_1, and X_2 be sets of input variables, Y_1, and Y_2 be sets of output variables, (X_1, Y_1, X_2, Y_2) be the variable ordering of a BDD_for_CF for the multiple-output function $\vec{f} = (f_1, f_2, \ldots, f_m)$, and W be the width of the BDD_for_CF at the height (X_1, Y_1). By applying **functional decomposition** to $\vec{f}$, we can obtain the network in Fig. 1, where the number of lines connecting two blocks is $t = \lceil \log_2 W \rceil$ [11].

Theorem 2.1: [8] Let μ_{max} be the maximum width of the BDD_for_CF that represents an n-input logic function $\vec{f}$. If $u = \lceil \log_2 \mu_{max} \rceil \leq k - 1$, then $\vec{f}$ can be realized by a cascade of k-input LUTs as shown in Fig. 2. By applying functional decompositions $s - 1$ times, we have the circuit having the structure of Fig. 2.

3. LUT Cascade and LUT Ring

3.1 LUT Cascade

An **LUT Cascade** is shown in Fig. 2, where multiple-output LUTs (cells) are connected in series to realize a multiple-output function. The wires connecting adjacent cells are called **rails**. The wiring delay is small since the wires between cells are limited to between the adjacent cells. Also, each cells may have external outputs in addition to the rail outputs. In this paper, k_i denotes the number of inputs to the i-th cell; u_i denotes *the number of rail outputs* of the i-th cell, i.e., the number of signal lines between i-th cell and $(i+1)$-th cell; $|Y_i|$ denotes *the number of the external outputs* of the i-th cell, i.e., the outputs that are connected to the primary output terminals; s denotes the number of levels; and r denotes the number of LUT cascades. Note that, the number of levels is equal to the number of cells in a cascade.

3.2 LUT Ring for Sequential Circuit

Although the LUT cascade has excellent features, it can only realize limited combinational circuits once the parameters, such as the number of cell inputs, the number of cells, and the number of rail outputs are fixed.

In this paper, we present an **LUT Ring** in Fig. 3 that realizes a wide range of sequential circuits. It sequentially emulates an LUT cascade that realizes the combinational part of the sequential circuit, and produces state variables, and external outputs. 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. When the LUT ring has enough memory, it can emulates many cascades by storing multiple LUT cascades. Also, the LUT ring has a faster realization by maximally utilizing the available memory. To make the circuit faster, 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 for a sequential circuit, all the data for the cells are stored in the **Memory for Logic**. Figure 4 illustrates that the LUT cascade with three cells is emulated by the LUT ring.

The **Input Register** stores the values of the primary inputs; the **Feedback Register** stores the values of the state variables; the **Output Register** stores the values of external outputs; the **MAR** (Memory Address Register) stores the address of the memory; the **MBR** (Memory Buffer Regis-

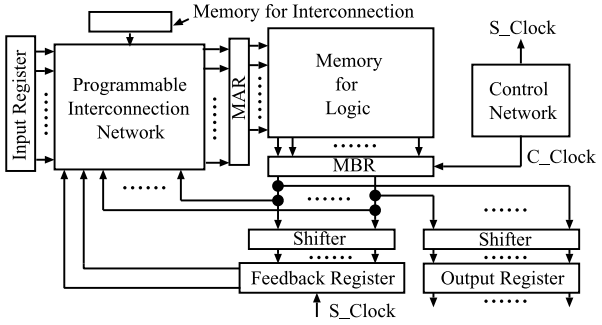
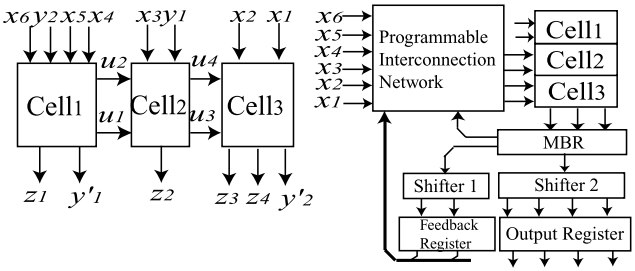


Fig. 3 LUT ring for sequential circuit.



(a) LUT cascade. (b) LUT ring.

Fig. 4 LUT ring which realizes the LUT cascade (a).

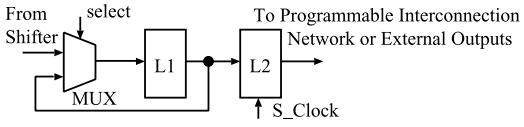


Fig. 5 Double-Rank Flip-Flop.

ter) stores the outputs of the memory; the **Programmable Interconnection Network** connects the Input register, the Feedback register, and the MAR, also it connects the MBR and the MAR; the **Memory for Interconnection** stores data for the interconnections; and the **Control Network** generates necessary signals to obtain functional values.

The control network produces signal to move data between the memory, programmable interconnection networks, and registers. The memory for interconnection stores the information on connections among cells. After evaluating each cascades, the outputs for external outputs are sent to the output register, and the outputs for state variables are sent to the feedback register. When the evaluation of all the cascades are finished, the values of the feedback register are sent to the programmable interconnection network, and the values of the output register are sent to the primary outputs.

To emulate the sequential circuit, the LUT ring uses two types of clock pulses: C_Clock to evaluate each cell of the LUT cascade, and S_Clock for state transitions. These clocks are produced by the control network. Figure 5 shows the **Double-Rank Filp-Flop** for the feedback register and the output register. Note that, L_1 and L_2 are D-latches. Set the select signals to high when all the cells in a cascade are evaluated, and send the values into L_1 latches. When all the cascades are evaluated, the values of the state variables are sent to L_2 latches. This can be done by adding a pulse to S_Clock. When the memory for logic and the memory for interconnection are implemented by rewritable memories, the LUT ring is reconfigurable. Also, we can reduce the amount of memory by **memory-packing**. Although **Mapping Memory** which stores the mapping information for the programmable interconnection network is needed for memory packing, we can ignore the cost of it, since the area is small compared with the memory for logic.

4. Design of a Sequential Circuit by an LUT Ring

In this section, we present an efficient method to map a sequential circuit into the memory of the LUT ring. We assume that the given sequential circuit consists of the combinational part and feedback flip-flops.

4.1 Synthesis Flow

Figure 6 illustrates the synthesis flow for a sequential circuit on the LUT ring.

First, partition the outputs of the combinational part into groups, and realize them by a set of LUT cascades. Since the combinational part of a sequential circuit usually has many inputs and outputs, a direct implementation by a single memory is usually impractical. In Sect. 4.2, we will adopt a strategy to realize many short cascades rather than to realize a single long cascade.

Second, reduce the number of levels by using cells with more inputs, when an enough memory is available. In Sect. 4.3, we will show an algorithm to reduce the number of levels using available memory.

4.2 Partition of the Outputs

Partitioning the outputs into groups after constructing a large single BDD_for_CF is inefficient, since the number of nodes of the BDD_for_CF is so large that the optimization of the BDD_for_CF is very time consuming. When the number of outputs is large, we partition the outputs into groups, and realize a cascade for each of them. The BDD_for_CF for all the outputs can be too large to be stored in the memory. Even if the BDD_for_CF is stored in a memory of the computer, it can be too large to be realized by an LUT cascade. Also, constructing a single BDD_for_CF for all the outputs is inefficient, since the optimization of a larger BDD_for_CF is time consuming.

In this paper, we use the strategy to start with many groups, and increase the number of outputs in each group. We partition the outputs into groups that minimizes the total number of cells in the cascades. For many practical functions, with the increase of the number of outputs for a BDD_for_CF, the width and the number of nodes increase rapidly. Thus, we partition the outputs into groups of equal number of outputs. In this case, we have to determine the

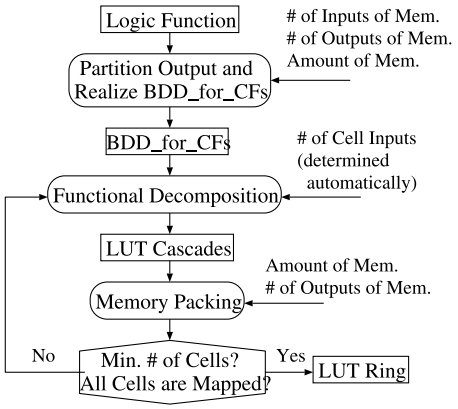


Fig. 6 Synthesis flow.

number of outputs for each groups. Trying all the possibility is impractical due to the excessive computation time. Therefore, in this paper, we find a reasonable number of outputs in each group by a heuristic method. We start with the partition, where each group consists of only one output, and we search for the optimal partition the outputs by increasing the number of outputs in groups. The amount of computation for variable ordering to optimize each BDD_for_CF is small, since the number of outputs for each BDD_for_CF is small.

We will define **Cost Function for the Total Number of Cells** of all the cascades, which will be used for partitioning the outputs.

Definition 4.1: (Cost Function for the Total Number of Cells)
Suppose that the output functions are partitioned into r groups, and each group is represented by a BDD_for_CF. Let w be the number of outputs in the memory for logic, n_i be the number of inputs for the i-th BDD_for_CF, and Sum_u_i be the sum of widths of all the levels in i-th BDD_for_CFs. Define EC that estimates the total number of cells in cascades as follows:

$$EC = \frac{\sum_{i=1}^{r} n_i - \bar{u} \cdot r}{(w+1) - \bar{u}}, \tag{1}$$

where $\bar{u} = \dfrac{\sum_{i=1}^{r} \lceil \log_2 Sum\_u_i \rceil}{\sum_{i=1}^{r} n_i}$ denotes the average of the logarithms of the sum of widths of all the levels in BDD_for_CFs.

Theorem 4.1: EC approximates a lower bound on the total number of cells in the LUT cascades that realize given multiple-output function.
(Proof) Partition the outputs into r groups, and represent them by a set of BDD_for_CFs. Let k be the maximum number of inputs for a cell, n_i be the number of variables in the i-th BDD_for_CF, s_i be the number of cells for the i-th group, $\hat{u}_i$ be the average number of rails for the i-th group, and w be the number of memory outputs. From the method of the cascade realization, we have:

$$n_i + \hat{u}_i(s_i - 1) \le s_i k \ (i = 1, 2, \ldots, r) \tag{2}$$

Since, we cannot obtain $\hat{u}_i$ directly, we approximate it by $\bar{u}$, the average value of the logarithms for the sum of widths in all the levels of all the BDD_for_CFs. Then, by summing (2) from 1 to r, we have the following relation:

$$\sum_{i=1}^{r} n_i + \bar{u} \sum_{i=1}^{r} (s_i - 1) \le \sum_{i=1}^{r} s_i \cdot k \tag{3}$$

The conditions that the given multi-output function is realized by an LUT cascade are:
($\lceil \log_2 width\ of\ BDD \rceil$) $\le$ (the number of inputs for a cell -1)
and ($\lceil \log_2 width\ of\ BDD \rceil$) $\le$ (the number ofmemory outputs)
By setting $k = w + 1$, and $Cs = \sum_{i=1}^{r} s_i$ to (3), we have $EC \le Cs$. (Q.E.D.)

In order to construct as small BDD_for_CFs as possible, we partition the outputs so that each group has a small number of support variables.

Definition 4.2: Let $F = \{f_1, f_2, \ldots, f_m\}$ be the set of the outputs functions, $G \subseteq F$, and $f_i \in F - G$. Then, the **similarity** of the output f_i with G is defined as follows:

$$Similarity(i, G, F) = |Sup(f_i) \cap Sup(G)|, \tag{4}$$

where $Sup(F)$ denotes the set of support variables of F.

Algorithm 4.1: (Partition the Outputs and Construct BDD_for_CFs)

Figure 7 shows the pseudo-code to partition outputs and to construct BDD_for_CFs.

The 14th to 23th lines of Algorithm 4.1 increase the number of outputs g in groups G, and generate the BDD_for_CF that represents G. The 16th to 20th lines make the G from the outputs functions F. The 8th line checks whether it keeps partition or not.

4.3 Realization of Cascades and Memory Packing

By Algorithm 4.1, partition the given multiple-output function into groups, and represented them by BDD_for_CFs.

First, we realize the LUT cascades for the given BDD_for_CFs with the cells having a specified number of inputs. Next, we will allocate the LUT data into the memory of the LUT ring. In the previous method, to find the faster implementation, the user has to design repeatedly by changing parameters. This is because, the functional decomposition and memory packing interact, and the estimation of the necessary amount of memory is difficult.

In the proposed method in this paper, to find the fastest implementation, the user need not specify the number of inputs to cells: the system finds it.

```
1 : Procedure Construct BDD_for_CFs(F);
2 : input  : F /*a set of functions ={f1, f2, ... , fn} */
3 : output : BDDcf /*a set of BDD_for_CFs */
4 :          :r /* denotes the number of BDD_for_CFs */
5 : EC ←∞ ; /* denotes the Cost Function */
6 : THEC ←∞ ; /* denotes the threshold of EC */
7 : g ← 0;/* denotes the maximum number of outputs in a group */
8 : while(THEC >= EC & g ≠ |F|){
9 :    H ← F; /* save F */
10:   BDDcf ← φ;
11:   r ← 0;
12:   THEC ← EC ;
13:   g ← g + 1;
14:  do {
15:     G ← φ; /* denotes the subsets of output functions*/
16:     do {
17:        (select fi  that has the maximum Similarity(i,G,F));
18:        F ← F - {fi};
19:        G ← G ⋃ {fi};
20:     }whlie (|G| < g & F ≠ φ);
21:     BDDcf ← BDDcf ⋃ (the BDD_for_CF that represents G);
22:     r ← r + 1;
23:  }whlie (F ≠ φ);
24:  ( calculate EC for BDDcf );
25:  F ← H;
26: }
27: return BDDcf, r;
```

Fig. 7 Pseudo-code for Algorithm 4.1.

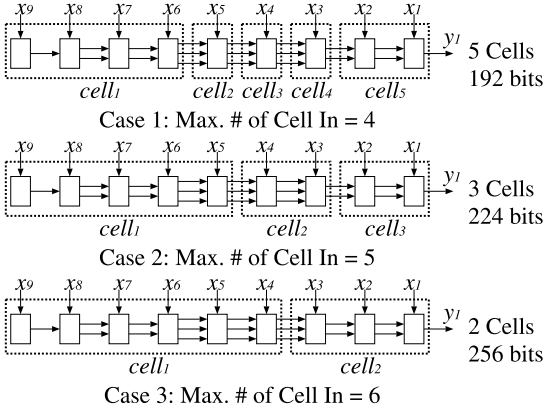


Fig. 8 Three different LUT cascades.

4.3.1 Principle to Reduce the Number of Cells

The next example illustrates the idea to reduce the number of cells by increasing the number of inputs to cells.

Example 4.1: Figure 8 shows three different LUT cascades for MCNC'89 benchmark function Life (9-input 1-output). It also shows the numbers of cells, and the amount of memory. For simplicity, in this example, we assume that the numbers of inputs of cells in each cascade are the same. We assume that the maximal amount of memory available is 256 bits. When the number of inputs of cell is four, we have the LUT cascade consisting of five cells (Case 1); when the number of inputs of cells is five, we have the LUT cascade consisting of three cells (Case 2); and, when the number of input of cells is six, we have the LUT cascade consisting of two cells (Case 3). (End of Example)

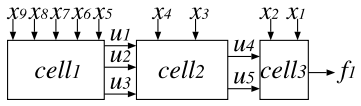


Fig. 9 Example of LUT cascade.

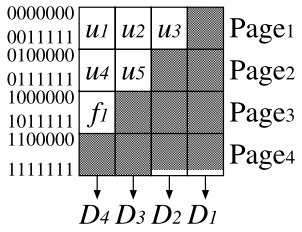


Fig. 10 Memory map of cell data.

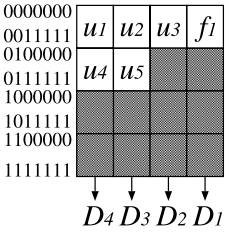


Fig. 11 Principle of memory packing.

By using this idea, we can find a set of LUT cascades with the minimum number of cells. Actually, we reduce the number of cells by increasing the number of inputs to cells while the memory is available. Note that the reduction of cells will reduce the evaluation time.

Next, we will introduce the memory packing that efficiently pack the LUT data into the memory for logic.

4.3.2 Memory Packing

In an LUT ring, all the data of the cells is stored in the memory for logic. In this case, we can reduce the necessary amount of memory by memory packing. The next example illustrates the idea.

Example 4.2: Figure 9 shows the LUT cascade for Life (9-input 1-output) that appeared in Example 4.1, where 5-input cells are used. Figure 10 shows the memory map of cell data, where the memory has 7-bit address inputs, and each word consists of four bits. (D_4, D_3, D_2, D_1) denotes the outputs of the memory. The dark parts in the figure are unused. In Fig. 10, only data for a single cell is stored in each page. By moving the cell data in Page 3 to the D_1 part of Page 1, we can reduce the necessary amount of memory by half (Fig. 11). (End of Example)

4.3.3 Lower Bound on the Size of Memory

The upper bound on the number of cells of LUT cascades (Theorem 4.4) is obtained from the available amount

of memory and BDD_for_CF produced in Algorithm 4.1. This bound limits the number of LUT cascades to consider. To derive an upper bound on the number of cells in LUT cascades, we first need to derive a lower bound on the number of cells of LUT cascades.

Theorem 4.2: (Lower Bound on the Number of Cells of an LUT Cascade) [8]

Suppose that an n-variable logic function is realized by the cascade shown in Fig. 2. Let s be the number of cells in the cascade, and let k be the maximum number of inputs of cells in the cascade. Then, we have the following relation:

$$\left\lceil \frac{n+u-2}{k-1} \right\rceil \leq s, \tag{5}$$

where $u = \max\{u_i\}$, and $u_i \leq \lceil \log_2 \mu_{max} \rceil$. When $n \leq k$, $\vec{f}$ can be realized by one cell.

(Proof) From the method of cascade realization, we have the relation:

$$n + \sum_{i=1}^{s-1} u_i \leq sk. \tag{6}$$

Since $1 \leq u_i \leq u$, we have $n + s - 2 + u \leq sk$. Thus, as for s, we have $\lceil \frac{n-2+u}{k-1} \rceil \leq s$. (Q.E.D.)

Suppose that $\vec{f}$ cannot be represented as $\vec{f}(\vec{X}_1, \vec{X}_2) = g(h(\vec{X}_1), \vec{X}_2)$, then $2 \leq u_i$, because the column multiplicities of non-trivial decomposition of $\vec{f}$ are greater than two. Therefore, we have the relation:

$$\sum_{i=1}^{s-1} u_i \geq 2(s-1). \tag{7}$$

From (6) and (7), we have the following:

Theorem 4.3: Let $\vec{f}(\vec{X})$ be an n-input function, and cannot be represented as $\vec{f} = \vec{g}(\vec{h}(\vec{X}_1), \vec{X}_2)$, where $(\vec{X}_1, \vec{X}_2)$ is a partition of X. Let s be the number of k-input cells to realize $\vec{f}(\vec{X})$. Then, we have the following relation:

$$\left\lceil \frac{n-2}{k-2} \right\rceil \leq s. \tag{8}$$

We can estimate the number of cells of the LUT cascade by using (8). From this theorem, we can derive an upper bound on the number of cells of LUT cascades.

Theorem 4.4: (Upper Bound on the Number of Inputs of a Cell)

Let k be the maximum number of inputs of cells in a cascade. Let n be the number of inputs of the BDD, L [bit] be the total amount of memory available, and w be the number output bits of the memory. Then, we have

$$\frac{2^k}{k-2} \leq \frac{L}{w(n-2)}, \tag{9}$$

where $n \geq k$.

(Proof) The upper bound on the size of memory L is given by $L = ws \cdot 2^k$, where k denotes the maximum number of the inputs to a cell, and s denotes the number of cells. By applying the above equation to expression (8), we have $\frac{2^k}{k-2} \leq \frac{L}{w(n-2)}$. (Q.E.D.)

From (9), we can obtain an upper bound of the number of inputs of a cell.

4.4 Cascade Realization with the Minimum Number of Cells under the Limitation of Memory

The evaluation time of an LUT ring is proportional to the total number of cells in the cascades. Thus, to minimize the evaluation time, we can formulate the design problem of an LUT ring as follows:

Problem 4.1: Suppose that the given multiple-output function $\vec{f} = (f_1(\vec{X}), f_2(\vec{X}), \ldots, f_m(\vec{X}))$ is represented by r BDD_for_CFs $(bddcf_1, bddcf_2, \ldots, bddcf_r)$. Let k_i be the number of inputs of cells in the cascade for $bddcf_i$, and let $Ncell(i, k_i)$ be the number of cells obtained by cascade realization of $bddcf_i$. When the cascade realization of $bddcf_i$ is impossible, $Ncell(i, k_i) = \infty$. Then, find the number of inputs of cells of each LUT cascade that satisfies the following conditions:

1. $\sum_{i=1}^{r} Ncell(i, k_i)$ is the minimum.
2. (The amount of memory obtained by memory packing) $\leq$ (The amount of available memory for an LUT ring)

To solve this problem, we use the dynamic programming. Dynamic programming is a technique to find an optimum solution, and finds the optimum solution of the partial problem step by step based on the optimal solutions obtained in the previous step. A partial problem is to obtain the minimum $\sum_{j=1}^{i} Ncell(j, k_j)$ of LUT cascades that realizes $bddcf_j$ $(j = 1, 2, \ldots, i)$. We solve the optimal solution for a partial problem by using evaluation function $\eta_i(k_i)$. It shows the necessary number of cells to map the cascades for $(bddcf_1, bddcf_2, \ldots, bddcf_i)$ by using cells with at most k_i inputs.

Definition 4.3:

$$\eta_i(k_i) = \begin{cases} \min\limits_{k'=3}^{k_{max}} [Ncell(i, k_i) + \eta_{i-1}(k')], \\ \quad \text{for} \quad 2 \leq i \leq r \\ \\ Ncell(i, k_i), \quad \text{for} \quad i = 1 \end{cases} \tag{10}$$

k_{max} is obtained by (9). When the values of evaluation function are the same, we choose k_i that requires the smaller amount of memory by memory packing.

For the given amount of memory, the next algorithm finds the number of inputs of cells of the LUT cascade for $bddcf_i$ that has the minimum number of cells.

```
1 : Procedure Find Optimal Cell Inputs(r, mem_avail, Kopt);
2 : input  : r /* number of bddcf's*/
3 :        :mem_avail /* the amount of available memory */
4 : output : Kopt /*denotes the numbers of inputs of cells in the
5 :  LUT cascades with the minimum number of cells for bddcf_i*/
6 : k_max ← upper bound of inputs of cells by expression (8);
7 : for k ← 3 to k_max do {
8 :   η_1(k) ← Ncell( 1, k);
9 :   K'opt(k) ← φ; /*denotes a set of numbers of inputs of
10:                          cells of each LUT cascade*/
11: }
12: for i ← 2 to r do {
13:   k'_max ← k_max;
14:   k_max ← upper bound of inputs of cells by expression (8);
15:   for k ← 3 to k_max do {
16:     η_i(k)←∞ ;
17:     for k' ← 3 to k'_max do {
18:       mem_pack ←(the amount of memory after memory packing);
19:       if(η_i(k) < Ncell( i, k) + η_i-1(k') & mem_pack <= mem_avail) {
20:         η_i(k) ← Ncell( i, k) + η_i-1(k');
21:         k_opt ←k';
22:       }
23:     }
24:     K'opt(k) ←K'opt(k)∪ {k_opt};
25:   }
26: }
27: (compute k_r to which η_r(k_r) is minimized);
28: Kopt ←K'opt(k_r)∪{k_r};
29: return Kopt;
```

Fig. 12 Pseudo-code for Algorithm 4.2.

Algorithm 4.2: (Find the Optimal Number of Cell Inputs of Each LUT Cascade)
Figure 12 shows the pseudo-code to find the optimal number of cell inputs by dynamic programming.

To obtain the initial solution, the 7th to 11th lines compute the number of cells of first LUT cascade. The 15th to 25th lines compute (10).

Example 4.1: The outputs of benchmark function s27 (8-input 4-output) is partitioned into two groups by Algorithm 4.1, and realized by two LUT cascades ($Cascade_1$, $Cascade_2$) shown in Figs. 13 and 14. We are going to allocate these cell data into the memory ($64 \times 6 = 384$ [bit]).
Step 1-1. From (9), we have $k_{max} = 6$.
Step 1-2. From (10), we have the evaluation function η_1 for $Cascade_1$:

$$\begin{aligned} \eta_1(3) &= Ncell(1,3) = 4 \\ \eta_1(4) &= Ncell(1,4) = 2 \\ \eta_1(5) &= Ncell(1,5) = 2 \\ \eta_1(6) &= Ncell(1,6) = 1 \end{aligned}$$

Step 2-1. From (9), we have $k_{max} = 5$.
Step 2-2. Figure 15 shows the memory map for LUT cascades after memory packing. Note that $Ncell(2,5) = 3$. From (10), we have the evaluation function η_2 of $Cascade_2$:

$$\begin{aligned} \eta_2(5) &= min[Ncell(2,5) + \eta_1(3), Ncell(2,5) + \eta_1(4), \\ &\quad Ncell(2,5) + \eta_1(5), Ncell(2,5) + \eta_1(6)] \\ &= min\{3+4, 3+2, 3+2, \infty\} \\ &= 5 \end{aligned}$$

Step 3. The evaluation function $\eta_2(k_{opt})$ takes its minimum when $k_{opt} = 5$. Note that $\eta_2(5) = 5$ when k_1 is four or five. We select four, since the case of four requires smaller amount of memory. Thus, the number of inputs for cells in the LUT cascades that minimizes the total number of cells is four, and five. (End of Example)

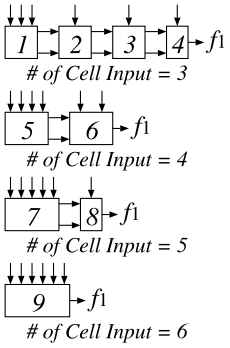


Fig. 13 Cascades for f_1 of s27.

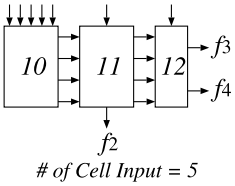


Fig. 14 Cascade for f_2, f_3, f_4 of s27.

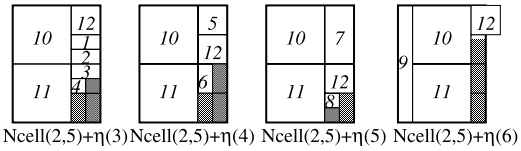


Fig. 15 Memory map for s27.

5. Experimental Results

5.1 Design of LUT Rings for ISCAS'89 Benchmark Functions

We implemented Algorithms 4.1 and 4.2 in the C programming language, and designed LUT rings for selected ISCAS'89 benchmark functions [14]. Table 1 compares three cases: the memory limitation of 1 Mega bits (**1 Mbit**), the memory limitation of 8 Mega bits (**8 Mbit**), and the memory limitation of 48 Mega bits (**48 Mbit**). *Name* denotes the name of a benchmark function; *In* denotes the number of inputs; *Out* denotes the number of outputs; *FF* denotes the number of flip-flops; *r* denotes the number of LUT cascades; *s* denotes the number of levels; and *Mem* denotes the amount of memory (Mega Bits). We used an IBM PC/AT compatible machine using a Pentium4 Xeon 2.8GHz (L1 Cache:32 kilo Byte, L2 Cache:512 kilo Byte) processor with 4GByte of memory. We used gcc compiler with the optimize option -O2 on a Redhad (Linux 7.3) operating system.

Table 1 Parameters for LUT rings to realize ISCAS'89 benchmark functions.

Name	In	Out	FF	r	1 Mbit (64k×16 bit)		8 Mbit (256k×32 bit)		48 Mbit (1M×48 bit)	
					s	Mem [Mbit]	s	Mem [Mbit]	s	Mem [Mbit]
s420	18	1	16	1	4	0.500	3	2.000	2	48.000
s510	19	7	6	2	3	0.251	3	0.251	3	0.750
s641	36	23	19	3	10	1.000	7	4.750	6	12.000
s713	36	23	19	3	9	0.755	7	4.000	7	6.000
s838	34	1	32	3	11	1.000	10	2.000	7	48.000
s1196	13	13	19	3	7	0.438	5	3.000	4	24.000
s1423	17	5	74	9	52	1.000	34	7.125	28	48.000
s5378	35	49	164	22	—	—	83	7.625	65	48.000
s9234	36	39	211	62	—	—	153	7.968	98	45.750
s13207	62	152	638	79	—	—	283	7.945	208	47.683

Table 2 Comparison of LUT ring with other methods.

Name	In	Out	FF	Direct [Mbit]	Mem+ Mux [Mbit]	LCC		LUT ring	
						Mem [Mbit]	time [ns]	Mem [Mbit]	time [ns]
s344	9	11	15	416.000	208.000	0.015	1180	0.006	46.4
s382	3	6	21	432.000	216.000	0.015	1530	0.008	41.9
s386	7	7	6	0.102	0.025	0.016	1630	0.016	19.4
s400	3	6	21	432.000	216.000	0.015	1540	0.008	41.9
s820	18	19	5	192.000	0.188	0.019	4080	0.015	32.9
s1494	8	19	6	0.391	0.049	0.025	7830	0.025	32.9

In Table 1, the symbol — denotes that realization of the cascades was impossible because of the memory limitation. Our strategy is to construct the cascades with as small number of levels as possible within the given amount of memory for logic. Therefore, with the enough amount of memory, we can reduce the number of levels considerably. Table 1 shows that the number of levels decreases with the increases of memory for almost all functions. Thus, we can realize a high-speed sequential circuit by an LUT ring.

For the benchmark functions s510 and s713, the numbers of levels are the same while sizes of memory are different. This can be explained as follows. Algorithm 4.1 generated different BDD_for_CFs for different sizes of memories. However, Algorithm 4.2 happened to find LUT cascades with the same number of cells. In these cases, since the numbers of rails and numbers of inputs to cells were different, we had cascades with different memory sizes.

5.2 Comparison with Other Methods

Table 2 compares LUT rings with other three methods, where *Name*, *In*, *Out*, and FF denote the same things as Table 1.

The column **Direct** denotes the size of memory for the direct implementation by a single memory. Consider the sequential circuit, whose number of the external input variables is n, the number of the external output variables is m, and the number of the state variables is p. The straightforward implementations of the transition functions and the output functions by memories require $p \cdot 2^{n+p}$ bits and $m \cdot 2^{n+p}$ bits, respectively. This is impractical when n and/or p are large.

The column **Mem+Mux** denotes the amount of memory by using multiplexers [6]. We can often reduce the necessary amount of memory by using properties of the given sequential circuits. In many case, the transition functions and output functions depend on proper subsets of the input variables. Let q be the maximal number of the external input variables on which next states depend. Then, we can use q qualifier variables instead of the external input variables. In this case, we use current state to select the external input variables. Also, we need q copies of n to 1-multiplexers to select the qualifier variables from the external inputs.

In the column **LCC** [1]; *Mem* denotes the size of executive code, and *time* denotes the evaluation time. The LCC is a kind of logic simulator that assigns a fragment of program code to each gate of logic circuits. Then, it evaluates codes from the inputs to the outputs in a topological order. To produce the executable code, we converted a benchmark circuit into the program code, and compiled it by gcc compiler with optimize option -O2. To obtain the evaluation time, we generated one million random test vectors on an IBM PC/AT compatible machine using a Pentium III 800MHz microprocessor with 256 MBytes of memory. We obtained average evaluation time per one vector, and considered it as the LCC evaluation time.

The column **LUT ring** denotes the amount of memory for logic in Fig. 3, and the evaluation time (ns). In the experiment, the memory size limitation of LUT ring is the size for the executive code when the benchmark function is implemented by LCC. Thus, the size of memory for LUT ring does not exceed the size of the executive code for LCC. Also, from the circuit simulation in [13], evaluation time for the LUT ring was estimated as follows:
Evaluation time[ns] = 4.5×Number of levels + 5.9.

Table 2 shows that Mem+Mux require the smaller amount of memory than Direct when the number of state variables is small (s386,s1494), or the number of the dependent variables of state transition functions is small (s820). However, when the number of state variables or the number of the dependent variables of state transition functions is large, the necessary amount of memory becomes too large. LCC and LUT ring can realize the benchmark functions with smaller amount of memory than Direct and Mem+Mux methods. Also, Table 2 shows that the LUT ring is 25 to 237 times faster than the LCC by using the same amount of memory.

6. Conclusion

In this paper, we presented a method to realize sequential circuits by using Look-Up Table (LUT) rings. In this method, the user need only to specify the amount of memory and the number of memory inputs or outputs. It finds high-speed sequential circuits by utilizing maximal amount of memory available on the LUT ring. We also compared the LUT rings with other methods to realize sequential circuits, and found that LUT rings efficiently realize sequential circuits in both the amount of memory and the evaluation time.

Acknowledgments

This research is partly supported by Japan Society for the Promotion of Science (JSPS), MEXT, and Kitakyushu Innovative Cluster. Discussions with Prof. Iguchi were quite useful.

References

[1] M. Abramovici, M.A. Breuer, and A.D. Friedman, Digital Systems Testing and Testable Design, Rev. Print ed., Wiley-IEEE Press, 1994.
[2] P. Ashar and S. Malik, "Fast functional simulation using branching programs," ICCAD'95, pp.408–412, Oct. 1995.
[3] R.K. Brayton, "The future of logic synthesis and verification," in Logic Synthesis and Verification, ed. S. Hassoun and T. Sasao, Kluwer Academic Publishers, 2001.
[4] R.E. Bryant, "Graph-based algorithms for Boolean function manipulation," IEEE Trans. Comput., vol.C-35, no.8, pp.677–691, Aug. 1986.
[5] E.M. Clarke, M. Fujita, and X. Zhao, "Hybrid decision diagrams: Overcoming the limitation of MTBDDs and BMDs," ICCAD'95, pp.159–163, San Jose, CA, Nov. 1995.
[6] D. Green, Modern Logic Design, Addison-Wesley, 1986.
[7] H. Nakahara, T. Sasao, and M. Matsuura, "A design algorithm for sequential circuits using LUT rings," SASIMI2004, pp.430–437, Oct. 2004.
[8] T. Sasao, M. Matsuura, and Y. Iguchi, "A cascade realization of multiple-output function for reconfigurable hardware," IWLS-2001, pp.225–300, Lake Tahoe, CA, June 2001.
[9] T. Sasao, M. Matsuura, Y. Iguchi, and S. Nagayama, "Compact BDD representations for multiple-output functions and their applications to embedded system," IFIP VLSI-SOC'01, pp.406–411, Montpellier, France, Dec. 2001.
[10] T. Sasao, M. Kusano, and M. Matsuura, "Optimization methods in look-up table rings," Proc. International Workshop on Logic and Synthesis (IWLS-2004), pp.431–437, Temecula, CA, June 2004.
[11] T. Sasao and M. Matsuura, "A method to decompose multiple-output logic functions," 41st Design Automation Conference, pp.428–433, San Diego, CA, June 2004.
[12] T. Sasao, H. Nakahara, M. Matsuura, and Y. Iguchi, "Realization of sequential circuits by look-up table ring," MWSCAS2004, pp.I517–I520, Hiroshima, July 2004.
[13] H. Qin, T. Sasao, M. Matsuura, S. Nagayama, K. Nakamura, and Y. Iguchi, "Realization of multiple-output functions by a sequential look-up table cascade," IEICE Trans. Fundamentals, vol.E87-A, no.12, pp.3141–3150, Dec. 2004.
[14] http://www.cbl.ncsu.edu/pub/Benchmark_dirs/ISCAS89/

Hiroki Nakahara was born on September 9, 1980 in Fukuoka, Japan, and received B.E. and M.E. degrees from Kyushu Institute of Technology, Iizuka, Japan in 2003 and 2005, respectively. He is now a doctoral student of Kyushu Institute of Technology. His research interest includes logic synthesis, reconfigurable devices, and high-level synthesis.

Tsutomu Sasao received the B.E., M.E., and Ph.D. degrees in electronics engineering from Osaka University, Osaka, Japan, in 1972, 1974, and 1977, respectively. He has held faculty/research positions at Osaka University, Japan, the IBM T.J. Watson Research Center, Yorktown Heights, New York, and the Naval Postgraduate School, Monterey, California. He is now a Professor of the Department of Computer Science and Electronics at the Kyushu Institute of Technology, Iizuka, Japan. His research areas include logic design and switching theory, representations of logic functions, and multiple-valued logic. He has published more than nine books on logic design, including *Logic Synthesis and Optimization, Representation of Discrete Functions, Switching Theory for Logic Synthesis*, and *Logic Synthesis and Verification*, Kluwer Academic Publishers, 1993, 1996, 1999, and 2001, respectively. He has served as Program Chairman for the IEEE International Symposium on Multiple-Valued Logic (ISMVL) many times. Also, he was the Symposium Chairman of the 28th ISMVL held in Fukuoka, Japan, in 1998. He received the NIWA Memorial Award in 1979, Distinctive Contribution Awards from the IEEE Computer Society MVL-TC for papers presented at ISMVLs in 1986, 1996, 2003 and 2004, and Takeda Techno-Entrepreneurship Award in 2001. He has served as an Associate Editor of the *IEEE Transactions on Computers*. He is a fellow of the IEEE.

Munehiro Matsuura was born on 1965 in Kitakyushu City, Japan. He studied at the Kyushu Institute of Technology from 1983 to 1989. He received the B.E. degree in Natural Sciences from the University of the Air, in Japan, 2003. He has been working as a Technical Assistant at the Kyushu Institute of Technology since 1991. He has implemented several logic design algorithms under the direction of Professor Tsutomu Sasao. His interests include decision diagrams and exclusive-OR based circuit design.