

A Method to Evaluate Logic Functions In the Presence of Unknown Inputs Using LUT Cascades

Yukihiro Iguchi¹ Tsutomu Sasao^{2,3} Munehiro Matsuura²

¹ Department of Computer Science, Meiji University

² Department of Computer Science, Kyushu Institute of Technology

³ Center for Microelectronic Systems, Kyushu Institute of Technology

Abstract

In logic simulation, we often need to evaluate two-valued logic functions in the presence of unknown inputs. However, a naive method often produces imprecise results. In these cases, we can obtain precise values by evaluating the regular ternary logic function for the given two-valued logic function. This paper shows a hardware realization of regular ternary logic functions. We use look-up table (LUT) cascades to implement double-rail logic representation. The evaluation time for n -input logic function is $O(n)$. They require much smaller amount of hardware than naive memory implementations.

1 Introduction

In the design of digital systems, design verification using logic simulation is one of the most time consuming steps [2, 12, 15]. In logic simulation, we often want to keep a selected subset of inputs as unknown, instead of definite values 0 or 1 [1, 3]. For example, if some inputs do not affect the outputs, we want to retain the inputs as unknown. We also need to treat unknown inputs for the initialization of logic networks [4]. For such cases, conventional logic simulators take too much time. Also, they often produce imprecise results for the networks with reconvergence. To evaluate the output values precisely, we need to evaluate regular ternary logic function (RT function) for the given two-valued logic function [16].

Cycle-based logic simulators based on binary decision diagrams (BDDs) are faster than conventional ones. However, to evaluate a logic function in the presence of unknown inputs, we have to evaluate the functions where the unknown values u are replaced by 0 and 1, in all possible combinations. Thus, the simulation time increases exponentially with the number of unknown inputs u . Another method is to use a Kleene TDD (Kleene ternary decision diagram) [10]. It produces precise values in $O(n)$ computation time. Unfortunately, a Kleene TDD requires $O(3^n/n)$ memory storage to represent an n -variable function. To make logic simulation faster, we can use special hardware.

In this paper, we show a fast method to evaluate RT functions by look-up table (LUT) cascades and LUT rings. This paper is organized as follows: Section 2 introduces methods to evaluate logic functions in the presence of unknown inputs. Section 3 shows methods to evaluate RT functions by using decision diagrams. Section 4 shows methods to represent RT functions by using double-rail logic. Section 5 shows LUT cascades to realize double-rail logic for RT functions. Section 6 shows experimental results. Section 7 summarizes the paper.

2 Evaluation of Logic Functions in the Presence of Unknown Inputs

Let $B = \{0, 1\}$. An n -variable two-valued logic function f is a mapping $f : B^n \rightarrow B$. Let $\vec{a} = (a_1, a_2, \dots, a_n)$ be a binary vector, where $a_i \in B$. We often need to evaluate the value $f(\vec{a})$ for \vec{a} , where some values of a_i are unknown [1].

In this section, we review some methods to evaluate f in the presence of unknown inputs. In the following, n denotes the number of input variables for the given two-valued logic function f .

Let $T = \{0, 1, u\}$, where u denotes an unknown input value. Let $\vec{\alpha} = (\alpha_1, \alpha_2, \dots, \alpha_n)$ be a ternary vector, where $\alpha_i \in T$. If α_i is either 0 or 1 for all i , then $\vec{\alpha} \in B^n$. In this case, $f(\vec{\alpha})$ is either 0 or 1. However, if $\alpha_i = u$ for some i , then $\vec{\alpha} \in T^n - B^n$. In this case, $f(\vec{\alpha})$ can be either 0 or 1 for some $\vec{\alpha}$, or can be unknown for other $\vec{\alpha}$. To obtain the precise values, we will introduce the regular ternary logic function.

Definition 2.1 Let $\vec{\alpha} \in T^n$. $A(\vec{\alpha})$ denotes the set of all binary vectors that are obtained by replacing all u with 0 or 1.

Let s be the number of u 's in $\vec{\alpha}$, then the set $A(\vec{\alpha})$ consists of 2^s binary vectors.

Definition 2.2 Let f be a two-valued logic function, and $\vec{\alpha} \in T^n$. **The regular ternary logic function [16] (RT**

$x \backslash y$	0	1	u
0	0	0	0
1	0	1	u
u	0	u	u

AND $x \cdot y$

$x \backslash y$	0	1	u
0	0	1	u
1	1	1	1
u	u	1	u

OR $x \vee y$

$x \backslash y$	0	1	u
0	0	1	u
1	1	0	u
u	u	u	u

EXOR $x \oplus y$

x	0	1	u
\bar{x}	1	0	u

NOT \bar{x}

Figure 2.1: Ternary AND, OR, EXOR, and NOT Operations.

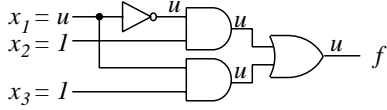


Figure 2.2: AND-OR Network for Expression in Example 2.1.

function \mathcal{F} for f is the mapping $T^n \rightarrow T$, where

$$f(A(\vec{\alpha})) = \{f(\vec{a}) \mid \vec{a} \in A(\vec{\alpha})\}.$$

$$\mathcal{F}(\vec{\alpha}) = \begin{cases} 0 & \text{if } f(A(\vec{\alpha})) = \{0\}. \\ 1 & \text{if } f(A(\vec{\alpha})) = \{1\}. \\ u & \text{if } f(A(\vec{\alpha})) = \{0, 1\}. \end{cases}$$

Note that \mathcal{F} is unique for the given f .

Example 2.1 Consider the logic function $f(x_1, x_2, x_3) = \bar{x}_1 x_2 \vee x_1 x_3$. Let $\vec{\alpha}_1 = (0, 0, u)$, $\vec{\alpha}_2 = (u, 1, 1)$, and $\vec{\alpha}_3 = (u, 1, u)$. Then, $\mathcal{F}(\vec{\alpha}_1)$, $\mathcal{F}(\vec{\alpha}_2)$, and $\mathcal{F}(\vec{\alpha}_3)$ are derived as follows:

$$\begin{aligned} f(A(\vec{\alpha}_1)) &= \{f(0, 0, 0), f(0, 0, 1)\} = \{0\}, \\ f(A(\vec{\alpha}_2)) &= \{f(0, 1, 1), f(1, 1, 1)\} = \{1\}, \\ f(A(\vec{\alpha}_3)) &= \{f(0, 1, 0), f(0, 1, 1), f(1, 1, 0), \\ &\quad f(1, 1, 1)\} = \{0, 1\}. \end{aligned}$$

By Definition 2.2, we have $\mathcal{F}(\vec{\alpha}_1) = 0$, $\mathcal{F}(\vec{\alpha}_2) = 1$, and $\mathcal{F}(\vec{\alpha}_3) = u$. (End of Example)

In a gate level logic simulation, we use a ternary logic simulator [1] based on the Kleenean strong ternary logic [11] shown in Fig. 2.1

Example 2.2 Fig. 2.2 shows an AND-OR network corresponding to the logical expression in Example 2.1. When we evaluate the function for the input $\vec{\alpha}_2 = (u, 1, 1)$ by using the method in Fig. 2.1, we have the value u as shown in Fig. 2.2. However, since $\mathcal{F}(\vec{\alpha}_2) = \mathcal{F}(u, 1, 1) = 1$, the real network produces the definite output 1. As shown in this example, the method using Fig. 2.1 often produces imprecise results. (End of Example)

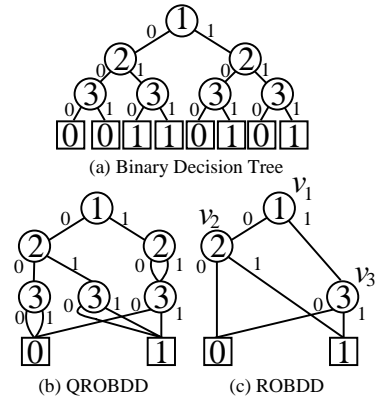


Figure 3.1: Example of BDDs.

Various methods exist to evaluate logic functions in the presence of unknown inputs: [6, 7] use BDDs, [10, 8] use Kleene TDDs, and [9] uses a special hardware implemented by FPGAs.

3 Evaluation of RT Functions Using Decision Diagrams

In this section, we illustrate some methods to represent RT functions by using decision diagrams. As for the terminology of decision diagrams, refer [5, 14, 17].

Example 3.1 Fig. 3.1(a) shows the binary decision tree for the function f in Example 2.1. The number 0 attached to each edge incident to v denotes $low(v)$, and 1 attached to each edge incident to v denotes $high(v)$. By merging equivalent sub-graphs in Fig. 3.1(a), we have a QROBDD (Quasi-Reduced Ordered BDD). Furthermore, by deleting redundant nodes, we have an ROBDD (Reduced Ordered BDD). (End of Example)

The next example illustrates an operation of a BDD-based logic simulator.

Example 3.2 By using the ROBDD in Fig. 3.1(c), we obtain the value $f(x_1, x_2, x_3) = f(0, 1, 1)$ for the network in Example 2.2. First, the label of the root node v_1 is 1, which denotes variable x_1 . Since the value of x_1 is 0, we go along the edge 0 to arrive node v_2 . The label of node v_2 is 2, which denotes variable x_2 . Since the value of x_2 is 1, we go along the edge 1 to arrive the terminal node $\boxed{1}$. Thus, we have $f(0, 1, 1) = 1$. (End of Example)

As shown in the above example, the BDD-based logic simulator evaluates the values of functions by tracing edges from the root node to a terminal node according to the values of each input vector. Practical simulators often generate the code shown in Fig. 3.2 from the decision diagram in Fig. 3.1(c). Then, they compile the code to execute it. Some

```

int f(int x1,x2,x3){
  if(x1) goto v3;
  else goto v2;
v2:  if(x2) return(1);
     else return(0);
v3:  if(x3) return(1);
     else return(0);
}

```

Figure 3.2: Example of Code Generated from BDD.

x	0	1	u
0	0	u	u
1	u	1	u
u	u	u	u

Figure 3.3: Alignment Operation $x \odot y$.

logic simulators generate native codes directly. The evaluation of an n -input logic function for an input vector requires traversing a BDD from the root node to a constant node. Thus, the evaluation time for an input vector is proportional to n . Some logic simulators use MDDs (Multi-valued Decision Diagrams) instead of BDDs to reduce evaluation time.

However, these methods can treat only two values: 0 and 1. To treat u , we have to evaluate function f many times after assigning 0 and 1 for each u , as shown in Definition 2.2.

We introduce the **Alignment operation** as follows:

Definition 3.1 Let $x, y \in T$.

$$x \odot y = \begin{cases} x & \text{if } x = y, \\ u & \text{otherwise.} \end{cases}$$

The Alignment operation is also denoted by the table shown in Fig. 3.3.

Theorem 3.1 $\mathcal{F}(u, x_2, \dots, x_n) = \mathcal{F}(0, x_2, \dots, x_n) \odot \mathcal{F}(1, x_2, \dots, x_n)$.

Example 3.3 By using the ROBDD shown in Fig. 3.1(c), obtain the value $\mathcal{F}(x_1, x_2, x_3) = \mathcal{F}(u, 1, 1)$ for the network in Example 2.2. The value of $\mathcal{F}(u, 1, 1)$ is obtained as $\mathcal{F}(0, 1, 1) \odot \mathcal{F}(1, 1, 1)$ by Theorem 3.1. As shown in Example 3.2, we have $\mathcal{F}(0, 1, 1) = 1$. Similarly, we have $\mathcal{F}(1, 1, 1) = 1$. Therefore, we have $\mathcal{F}(u, 1, 1) = \mathcal{F}(0, 1, 1) \odot \mathcal{F}(1, 1, 1) = 1 \odot 1 = 1$. (End of Example)

Theorem 3.1 implies that we have to traverse both *low* and *high* edges at the nodes whose input variables are u . In the worst case, this method requires $O(2^s \cdot n)$ time, where s is the number of u 's in the input vector. If we use a Kleene TDD, the evaluation time will be $O(n)$. However, the number of nodes increases with $O(3^n/n)$, and so it is impractical when n is large [8].

4 Double-Rail Representations of RT Functions

In this section, we will show a method to represent an RT function by a pair of two-valued logic functions.

4.1 Double-Rail Representation

Definition 4.1 Let f be an n -variable two-valued logic function. Let $\mathcal{F}: T^n \rightarrow T$ be the n -variable RT function for f . We represent three values of $T = \{0, 1, u\}$ by a pair of two-valued variable (x_L, x_H) as follows:

$$\begin{aligned} (x_L, x_H) &= (1, 0) \text{ denotes } x = 0, \\ (x_L, x_H) &= (0, 1) \text{ denotes } x = 1, \text{ and} \\ (x_L, x_H) &= (1, 1) \text{ denotes } x = u. \end{aligned}$$

$(x_L, x_H) = (0, 0)$ is not used. We represent an RT function by a pair of two-valued logic functions (f_L, f_H) . It is a **double-rail representation of an RT function for f** .

(f_L, f_H) can be obtained by Procedure 4.1 using sum-of-products expressions (SOPs) as follows:

Procedure 4.1 Let $f(X)$ be a given logic function, where $X = (x_1, x_2, \dots, x_n)$. In an SOP for \bar{f} , replace un-complemented literals x_i with x_{iH} , and replace complemented literals \bar{x}_i with x_{iL} , to obtain the SOP for f_L . In an SOP for f , replace un-complemented literals x_i with x_{iH} , and replace complemented literals \bar{x}_i with x_{iL} , to obtain the SOP for f_H .

Theorem 4.1 The pair of functions (f_L, f_H) obtained by Procedure 4.1 is a double-rail representation of the RT function for f .

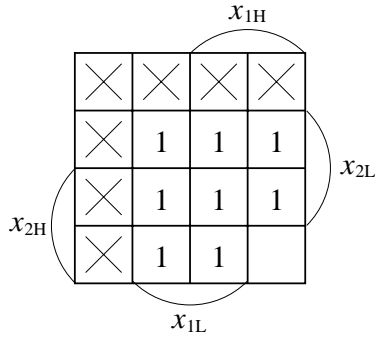
(Proof) By the construction of f_L and f_H , the following are clear: For the input combinations such that $f = 0$, we have $f_L = 1$, and for the input combinations such that $f = 1$, we have $f_H = 1$.

On the other hand, no valid combination makes $(f_L, f_H) = (0, 0)$. For the combinations such that $(f_L, f_H) = (1, 0)$, the output values are $f = 0$ (definite). For the combinations such that $(f_L, f_H) = (0, 1)$, the output values are $f = 1$ (definite). For the combinations such that $(f_L, f_H) = (1, 1)$, the output values are unknown. Thus, (f_L, f_H) obtained by Procedure 4.1 represents the RT function. (Q.E.D.)

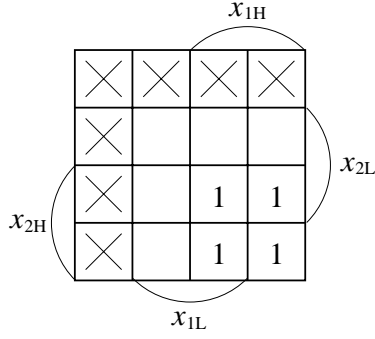
As will be shown, f_L and f_H are incompletely specified functions, so their representations are not unique.

4.2 Don't Cares

Let $f: B^n \rightarrow B$ be an n -variable logic function. Let (f_L, f_H) be a double-rail representation for the RT function $\mathcal{F}: T^n \rightarrow T$. In this case, f_L and f_H represent mappings $B^{2n} \rightarrow B$. Note that these functions are undefined for the inputs $(x_L, x_H) = (0, 0)$. Thus, these functions are defined



(a) Map for f_L .



(b) Map for f_H .

Figure 4.1: Simplification of SOPs for f_L and f_H .

only for 3^n out of 4^n possible input combinations. For the remaining $4^n - 3^n$ combinations, values of the functions are undefined. The fractions of *don't cares* in $2n$ -variable functions f_L and f_H are

$$\frac{4^n - 3^n}{4^n} = 1 - \left(\frac{3}{4}\right)^n.$$

For the functions with $n = 10$, more than 94% of the input combinations are *don't cares*. By using these *don't cares*, we can simplify the representations for f_L and f_H drastically. Especially, we can often reduce the number of variables. The next example illustrates this.

Example 4.1 Consider the two-variable function $f = x_1x_2$. The canonical SOP for \bar{f} is given by $\bar{f} = \bar{x}_1\bar{x}_2 \vee \bar{x}_1x_2 \vee x_1\bar{x}_2$. By Procedure 4.1, we can obtain the SOPs for f_L and f_H as follows: $f_H = x_{1H}x_{2H}$, and $f_L = x_{1L}x_{2L} \vee x_{1L}x_{2H} \vee x_{1H}x_{2L}$. The Karnaugh map in Fig. 4.1 shows that the input combinations for $(x_L, x_H) = (0, 0)$ are *don't cares*. By using these *don't cares*, the SOP for f_L is simplified to $f_L = x_{1L} \vee x_{2L}$. Note that the original SOP depends on four variables, while the simplified SOP depends on only two variables. (End of Example)

Example 4.2 Here, we obtain the double-rail representation (f_L, f_H) from the canonical form of a three-variable function f :

$$\begin{aligned} f_L &= \bar{f}_{000}X_{1L}X_{2L}X_{3L} \vee \bar{f}_{001}X_{1L}X_{2L}X_{3H} \\ &\vee \bar{f}_{010}X_{1L}X_{2H}X_{3L} \vee \bar{f}_{011}X_{1L}X_{2H}X_{3H} \\ &\vee \bar{f}_{100}X_{1H}X_{2L}X_{3L} \vee \bar{f}_{101}X_{1H}X_{2L}X_{3H} \\ &\vee \bar{f}_{110}X_{1H}X_{2H}X_{3L} \vee \bar{f}_{111}X_{1H}X_{2H}X_{3H}. \\ f_H &= f_{000}X_{1L}X_{2L}X_{3L} \vee f_{001}X_{1L}X_{2L}X_{3H} \\ &\vee f_{010}X_{1L}X_{2H}X_{3L} \vee f_{011}X_{1L}X_{2H}X_{3H} \\ &\vee f_{100}X_{1H}X_{2L}X_{3L} \vee f_{101}X_{1H}X_{2L}X_{3H} \\ &\vee f_{110}X_{1H}X_{2H}X_{3L} \vee f_{111}X_{1H}X_{2H}X_{3H}. \end{aligned}$$

(End of Example)

As shown in Example 4.1, we can often simplify the SOPs for f_L and f_H by using *don't cares*. However, in general, SOPs for f_L and f_H require $O(2^n)$ product terms.

4.3 Unate Functions

Definition 4.2 A function f is **monotone increasing** if f is a constant, or can be represented by an SOP without using complemented literals. A function f is **monotone decreasing** if \bar{f} is monotone increasing.

Definition 4.3 If a logic function f can be represented by an SOP without using the literal x_i , then f is **monotone increasing** with respect to x_i . If a logic function f can be represented by an SOP using only one of two literals of a variable x_i , then f is **unate** in x_i . If any SOP of a logic function f have both literals of a variable x_i , then f is **bi-nate** in x_i .

Definition 4.4 A logic function f is **unate** if f is a constant, or can be represented by an SOP using only one of two literals for each variable. Monotone increasing functions and monotone decreasing functions form special classes of unate functions.

Theorem 4.2 [13] The minimum sum-of-products expression for a unate function is unique.

Lemma 4.1 Let $f(x_1, x_2, \dots, x_n)$ be an n -variable monotone increasing function. Then, the double-rail representation of the RT function is given as (f_L, f_H) , where

$$\begin{aligned} f_L &= \bar{f}(x_{1L}, x_{2L}, \dots, x_{nL}), \text{ and} \\ f_H &= f(x_{1H}, x_{2H}, \dots, x_{nH}). \end{aligned}$$

(Proof) Since f is monotone increasing, f can be represented by an SOP using only un-complemented literals. Also, \bar{f} can be represented by an SOP using only complemented literals. To these SOPs, apply Procedure 4.1, and

we have

$$\begin{aligned} f_L &= \bar{f}(x_{1L}, x_{2L}, \dots, x_{nL}), \text{ and} \\ f_H &= f(x_{1H}, x_{2H}, \dots, x_{nH}). \end{aligned} \quad (Q.E.D.)$$

In general, f_L and f_H are $2n$ -variable functions. However, if f is monotone increasing, f_L and f_H can be represented as n -variable functions.

Theorem 4.3 *Let f be a monotone increasing function of n variables. Double-rail representation of the RT function (f_L, f_H) can be represented by a shared binary decision diagram (SBDD) with $O(2^n/n)$ nodes.*

(Proof) From Lemma 4.1, f_L can be represented by an SOP that depends on only x_{iL} ($i = 1, 2, \dots, n$), and f_H can be represented by an SOP that depends on only x_{iH} ($i = 1, 2, \dots, n$). Thus, both f_L and f_H can be represented as n -variable functions. In general, an n -variable function can be represented by a BDD with $O(2^n/n)$ nodes [21]. Also, the variables in f_L and f_H are completely disjoint. Thus, the SBDD for (f_L, f_H) can be represented with $O(2^n/n)$ nodes. (Q.E.D.)

Corollary 4.1 *Even if some of the inputs are unknown, a unate function of n variables can be evaluated in $O(n)$ time by using a BDD with $O(2^n/n)$ nodes.*

(Proof) For simplicity, assume that f is a monotone increasing function. By Lemma 4.1, f_L can be represented by variables x_{iL} , and f_H can be represented by variables x_{iH} . Thus, both f_L and f_H can be represented by a BDD with $O(2^n/n)$ nodes.

In the order of the variables, assume that the variables x_{iL} appear first, and the variables x_{iH} appear the next. Then, the SBDD for f_L and f_H has $O(2^n/n)$ nodes. This is because f_L depends only on x_{iL} , and f_H depends only on x_{iH} . Since this SBDD is a double-rail representation of the RT function, we can evaluate f in $O(n)$ time. When f is a unate function, we can prove in a similar way. (Q.E.D.)

Definition 4.5 *Let $f(X_1, X_2)$ be binate with the variables X_1 , and unate with the variables X_2 . Then, f is **partially unate**.*

Theorem 4.4 *Let $f(X_1, X_2)$ be binate with X_1 and unate with X_2 . Let (f_L, f_H) be a double-rail representation of the RT function. Then, f_L and f_H can be represented as $(2n_1 + n_2)$ -variable functions, where n_1 and n_2 denote numbers of variables in X_1 , and X_2 , respectively.*

(Proof) For simplicity, assume that f is positive with respect to the variables in X_2 . First, obtain minimum SOPs for f and \bar{f} . Then, obtain SOPs for f_L and f_H by using Procedure 4.1. Note that f_L depends on X_{1L} , X_{1H} , and X_{2L} , while f_H depends on X_{1L} , X_{1H} , and X_{2H} . Thus, we have the theorem. When f is unate with respect to the variables in X_2 , we can prove similarly. (Q.E.D.)

5 Hardware for Double-Rail Realizations of RT Functions

Although BDDs and Kleene TDDs can be implemented easily by software, they are time consuming for logic simulation with many input vectors. In this section, we consider various hardware for RT functions.

5.1 Comparison of Various Methods

A method using RAM

We can directly implement the double-rail representation (f_L, f_H) by a RAM. Note that f_L and f_H have $2n$ inputs. Thus, this method requires a memory with $2n$ -bit address. Although the evaluation time is $O(1)$, the size of memory is 2×4^n bits, which is too large for practical applications.

For a partially unate function, we can reduce the required memory as follows: Let n_1 be the number of binate variables, and n_2 be the number of unate variables. By Theorem 4.4, f_L and f_H can be represented as $(2n_1 + n_2)$ -variable functions. If f_L and f_H are realized independently by memory, they require $2 \times 2^{2n_1+n_2} = 2^{2n_1+n_2+1}$ bits of memory.

A method using LUT cascades

An LUT cascade [19] shown in Fig. 5.1 is obtained from a BDD by iterative functional decompositions. It reduces the necessary amount of memory to represent f by detecting repeated patterns, and by utilizing *don't cares* in the function. The wires that connect adjacent cells are called **rails**. In the design of LUT cascade, the reduction of the number of rails is very important. Reduction of the number of rails directly reduces the size of the subsequent LUT. The number of rails is related to the width of the BDD. Functions having small BDD widths have efficient LUT cascade realizations, while functions having large BDD widths do not have efficient LUT cascade realizations.

Recall that the double-rail representation of the RT function denotes a mapping $(f_L, f_H) : B^{2n} \rightarrow B^2$. Since the input combinations $(X_L, X_H) = (0, 0)$ are not used, the function is defined only for 3^n out of 4^n input combinations. LUT cascades are suitable for the realization of functions with a large amount of *don't care*.

Also, in functions for control circuits, many input variables are unate. As shown in Theorem 4.3, functions with many unate variables require smaller number of nodes in BDDs. This implies that such functions can be efficiently realized by LUT cascades.

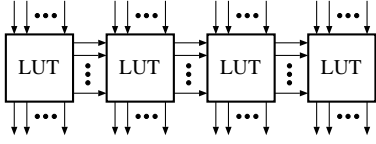


Figure 5.1: LUT Cascade.

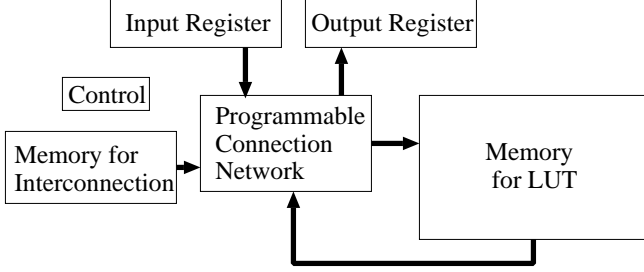


Figure 5.2: LUT Ring.

5.2 Realization Using LUT Rings

An LUT cascade realizes a given function f by the structure shown in Fig. 5.1. However, once the numbers of inputs and outputs for each LUT are fixed, the LUT cascade can realize only a limited class of functions. Thus, we use an **LUT ring** having the architecture shown in Fig. 5.2. It emulates an LUT cascade. It consists of a large memory that stores LUTs, and a control part that implements interconnections. Note that the necessary amount of memory is at least the sum of memories for LUTs.

Theorem 5.1 *If a unate function $f(x_1, x_2, \dots, x_n)$ can be realized by an LUT cascade with r rails, then functions f_L and f_H are also realized by LUT cascades with r rails.*

(Proof) For simplicity, assume that f is monotone increasing. By Lemma 4.1, f_L and f_H can be represented as $f_L = \bar{f}(x_{L1}, x_{L2}, \dots, x_{Ln})$ and $f_H = f(x_{H1}, x_{H2}, \dots, x_{Hn})$, respectively. By attaching an inverter to the output of the cascade for f , we obtain the cascade for \bar{f} . Thus, if f is realized by a cascade with r rails, then \bar{f} is also realized by a cascade with r rails. These two functions can be realized by cascades shown in Fig. 5.1. In the case where f is unate, we can prove the theorem in a similarly manner. (Q.E.D.)

Realization of Partially Unate Functions

Most of control circuits realize partially unate functions. Let f be partially unate with respect to X_2 . We can convert f into monotone increasing with respect to X_2 , by complementing some of variables in X_2 appropriately. Given the double-rail representation of the RT function (f_L, f_H) , realize a pair of cascades as shown in Fig. 5.3. One cascade realizes f_L , and other realizes f_H . The cascade for f_L can

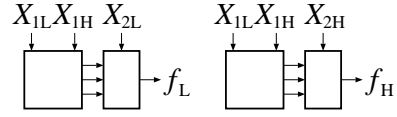


Figure 5.3: Realization of Partially Unate Function.

be decomposed into two parts: one depends on X_{L1} and X_{H1} , and other depends on X_{L2} . Similarly, the cascade for f_H can be decomposed into two parts: one depends on X_{L1} and X_{H1} , and other depends on X_{H2} .

In general, control circuits have many unate input variables. If f is unate in x_i , then f_L and f_H depend on either x_{iL} or x_{iH} . Thus, separate realization of f_L and f_H does not waste hardware.

6 Experimental Results

For a multiple-output function $F(f_0, f_1, \dots, f_{m-1})$, we generated a pair of multiple-output functions $F_L(f_{0L}, f_{1L}, \dots, f_{(m-1)L})$ and $F_H(f_{0H}, f_{1H}, \dots, f_{(m-1)H})$. Then, we implemented them by separate LUT rings. To design LUT cascades with intermediate outputs, we used BDDs for characteristic functions [20]. We assumed that each LUT has at most 15 inputs, and at most 16 outputs.

Table 6.1 shows the experimental results. In the table, *Name* denotes the name of benchmark function; *In* denotes the number of inputs; *Out* denotes the number of outputs; *Depend* denotes the number of variables that actually influence the outputs; *Unate* denotes the number of unate variables; RT_{in} denotes the number of variables that actually influence f_L and f_H ; *LUT* denotes the total number of outputs used in the LUTs; *Level* denotes the maximum number of levels; *Cas* denotes the number of cascades; and *Mem* denotes the total amount of bits to represent LUTs.

For example, *apex2* is a 39-input 3-output function. Among 39 inputs, the function does not depend on 9 variables. It has 26 unate variables, and four binate variables. Also, one of the outputs is the constant 0. By Theorem 4.4, f_L and f_H can be represented as 34-variable functions. Both of f_L and f_H can be realized by cascades with three levels. Thus, the evaluation time for (f_L, f_H) is equal to the time for three memory references. Also, the amount of memory to represent f_L and f_H are about 200 kbits and 231kbits, respectively.

In the case of *apex1*, we need two cascades to realize f_H .

Direct realizations of f_L and f_H require large memories, whose total size is $2^{RT_{in}+1}$. Table 6.1 shows that the direct realization by memory is unrealistic.

7 Concluding Remarks

In this paper, we considered a method to evaluate logic functions in the presence of unknown inputs. To obtain precise values, we evaluate RT functions. We also showed

Table 6.1: Cascade Realizations of RT Functions.

Name	In	Out	Depend	Unate	RT _{in}	f_L				f_H			
						LUT	Level	Cas	Mem	LUT	Level	Cas	Mem
accpla	50	69	50	11	89	439	15	3	3,178,496	1001	16	7	1,351,680
apex1	45	45	45	4	86	183	16	1	5,177,424	515	21	2	7,706,624
apex2	39	3	30	26	34	9	3	1	199,680	9	3	1	231,424
C432	36	7	36	1	71	338	27	1	10,874,880	174	16	1	5,554,176
exep	30	63	28	12	44	128	10	1	1,716,992	271	16	2	983,104
misj	35	14	35	31	39	23	4	1	688,144	18	3	1	491,520
rckl	32	7	32	1	63	43	7	1	1,247,744	40	7	1	1,212,608
signet	39	8	39	20	58	160	14	1	5,242,880	170	15	1	5,443,584
xparc	41	73	39	3	75	180	15	1	3,577,859	455	16	3	2,179,072

double-rail realizations of RT functions by using LUT cascades. By experiment, we showed that the evaluation time is $O(n)$, and they are much smaller than the straightforward RAM realizations.

Acknowledgments

This research is partly supported by JSPS, the Grant in Aid for Scientific Research, and MEXT, the Kitakyushu area innovative cluster project.

References

- [1] M. Abramovici, M. A. Breuer, and A. D. Friedman, "Digital Systems TESTING and Testable DESIGN," pp. 43–46, Computer Science Press, 1990.
- [2] P. Ashar, and S. Malik, "Fast functional simulation using branching programs," *ICCAD'95*, pp. 408–412, Nov. 1995.
- [3] M. A. Breuer, "A note on three-valued logic simulation," *IEEE Trans. on Comput.*, Vol. C-21, No. 4, pp. 399–402, April 1972.
- [4] J. A. Brzozowski and M. Yoeli, "On a ternary model of gate networks," *IEEE Trans. on Comput.*, Vol. C-28, No. 3, pp. 178–184, March 1979.
- [5] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. Comput.*, Vol. C-35, No. 8, pp. 677–691, Aug. 1986.
- [6] R. E. Bryant, "Boolean analysis of MOS circuits," *IEEE Trans. on CAD of Integrated Circuits*, CAD-6(4), pp. 634–649, July 1987.
- [7] R. E. Bryant, "Symbolic simulation – techniques and applications," *Proc. of 27th Design Automation Conf.*, pp. 517–521, June 1990.
- [8] Y. Iguchi, T. Sasao, and M. Matsuura, "On properties of Kleene TDDs," *IEICE Trans. INF. & SYST.*, Vol. E81-D, No. 7, pp. 716–723, July 1998.
- [9] Y. Iguchi, T. Sasao, M. Matsuura, and A. Iseno, "Realization of regular ternary logic functions using double-rail logic," *Asia and South Pacific Design Automation Conference, ASP-DAC'99*, Jan. 1999, pp. 331–334.
- [10] G. Jennings, "Symbolic incompletely specified functions for correct evaluation in the presence of indeterminate input values," *28th Hawaii Int'l Conf. on System Science (Vol. I: Architecture)*, pp. 23–31, Jan. 1995.
- [11] S. C. Kleene, *Introduction to Metamathematics*, Wolters-Noordhoff, North-Holland Publishing, 1952.
- [12] P. C. McGeer, K. L. McMillan, A. Saldanha, A. L. Sangiovanni-Vincentelli, and P. Scaglia, "Fast discrete function evaluation using decision diagrams," *ICCAD'95*, pp. 402–407, Nov. 1995.
- [13] R. McNaughton, "Unate truth functions," *IRE Transactions on Electronic Computers*, Vol. EC-10, No. 1, pp. 1–6, March 1961.
- [14] S. Minato, "Graph-based representation of discrete functions," in [17].
- [15] R. Murgai and M. Fujita, "Some recent advances in software and hardware logic simulation," *10th Int'l Conf. on VLSI Design*, pp. 232–238, Jan. 1997.
- [16] M. Mukaidono, "Regular ternary logic functions – ternary logic functions suitable for treating ambiguity," *IEEE Trans. Comput.*, Vol. C-35, No. 2, pp. 179–183, Feb. 1986.
- [17] T. Sasao, and M. Fujita, (ed.): *Representations of Discrete Functions*, Kluwer Academic Publishers, 1996.
- [18] T. Sasao, *Switching Theory for Logic Synthesis*, Kluwer Academic Publishers, 1999.
- [19] T. Sasao, M. Matsuura, and Y. Iguchi, "A cascade realization of multiple-output function for reconfigurable hardware," *International Workshop on Logic and Synthesis(IWLS01)*, Lake Tahoe, CA, June 12–15, 2001, pp. 225–230.
- [20] T. Sasao and M. Matsuura, "A method to decompose multiple-output logic functions," *Technical Report of IEICE*, VLD2003-108, Kitakyushu, Japan, Nov. 27–28, 2003.
- [21] C. E. Shannon, "The synthesis of two-terminal switching circuits," *Bell Syst. Tech. J.*, 28, 1, pp. 59–98, 1949.