Realization of Regular Ternary Logic Functions using Double-Rail Logic

Yukihiro IGUCHI[†]
Munehiro MATSUURA[‡]

[†]Dept. of Computer Science
Meiji University
Kawasaki 214-8571, JAPAN
e-mail: {iguchi, iseno}@cs.meiji.ac.jp

Tsutomu SASAO[‡]
Atsumu ISENO[†]

[‡]Dept. of Electronics and Computer Science
Kyushu Institute of Technology
Iizuka 820-8502, JAPAN
e-mail: {sasao, matsuura}@cse.kyutech.ac.jp

Abstract— In logic simulation, we often have to evaluate logic functions in the presence of unknown inputs. However, the naive method often produces incorrect values. In these cases, we can produce correct values by evaluating regular ternary logic functions instead of switching functions. This paper proposes a realization of regular ternary logic functions by using double-rail logic. This implementation requires $O(2^n/n)$ logic cells, and $O(n)$ time to simulate an n-variable logic function. We showed an FPGA realization that is about 100 times faster than software simulation.

I. Introduction

In logic simulation, we often have to evaluate logic functions in the presence of unknown inputs. For example, if some inputs do not affect the outputs, we want to retain the inputs as unknown not 0's or 1's as is common. In such a case, a naive method often produces incorrect values, or simulation takes too long for large networks. It is known that correct values are obtained by evaluating the regular ternary functions (RT functions) derived from the given two-valued logic function[3].

BDD(binary decision diagram)-based simulators are faster than conventional ones[1, 2]. However, the evaluation time increases exponentially when we evaluate the functions in the presence of unknown inputs[4]!%Kleene_TDDs presented by Jennings[6] evaluate logic functions in the presence of unknown inputs. A Kleene_TDD-based simulator produces correct results very quickly. However, a Kleene_TDD requires a large amount of memory.

This paper is organized as follows: Section 2 reviews a proposed method to evaluate logic functions in the presence of unknown inputs. Section 3 presents a hardware realization of RT functions using double-rail logic. We show an FPGA realization that is about 100 times faster than software simulation. Section 4 shows preliminary experimental results.

II. Evaluation of Logic Functions in the Presence of Unknown Inputs

Let $B = \{0, 1\}$. An n-variable switching function f represents the mapping: $f : B^n \rightarrow B$.
Let $\vec{a} = (a_1, a_2, \ldots, a_n)$ be a binary vector, where $a_i \in B$. We often have to evaluate the value $f(\vec{a})$ for $\vec{a}$, where some a_i are unknown. In this section, we will review the method to evaluate f in the presence of unknown inputs.

Let $T = \{0, 1, u\}$, where u denotes the truth value showing an unknown input. Let $\vec{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ 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. If $\alpha_i = u$ for some i, then $\vec{\alpha} \in T^n - B^n$. In this case, for some $\vec{\alpha}$, $f(\vec{\alpha})$ is either 0 or 1, but for some other $\vec{\alpha}$, $f(\vec{\alpha})$ is undetermined. Therefore, it is convenient to introduce a three-valued logic function, $\mathcal{F} : T^n \rightarrow T$, which is derived from f.

Definition 2.1 *Let $\vec{\alpha} \in T^n$. $A(\vec{\alpha})$ denotes the set of all the 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 $\vec{\alpha} \in T^n$.*

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

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

Then, $\mathcal{F}$ is the regular ternary logic function[3] of f(hereafter, we will call it an RT function).

Example 2.1 *Consider the 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:*

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

By Definition 2.2, we have

$$\mathcal{F}(\vec{\alpha}_1) = 0, \quad \mathcal{F}(\vec{\alpha}_2) = 1, \quad \textit{and} \quad \mathcal{F}(\vec{\alpha}_3) = u. \qquad \blacksquare$$

In a gate-level logic simulation, we extend binary logic to ternary logic as shown in Figure 2.1. This is the Kleenean strong ternary logic[5].

Example 2.2 *Figure 2.2 shows an AND–OR network that realizes the function in Example 2.1. When we evaluate the output f for the input $\vec{\alpha}_2 = (u, 1, 1)$ by using a naive method, we evaluate signals from the primary inputs to the primary outputs. In this case, we have the output u as shown in Figure 2.2. However, Example 2.1 shows that*

$$\mathcal{F}(\vec{\alpha}_2) = \mathcal{F}(u, 1, 1) = 1.$$

Thus, a naive method produces an incorrect value for this input. $\blacksquare$

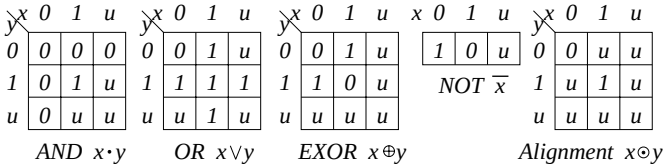


Figure 2.1: Ternary AND, OR, NOT, EXOR, and Alignment operations

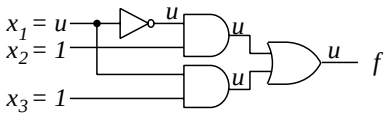


Figure 2.2: Realization of function in Example 2.1.

Several methods exist to evaluate output values according to Definition 2.2, i.e., to evaluate correct value by computing RT functions: [7] use BDDs[8], and [6, 4] use Kleene_TDDs.

III. A Hardware Realization of RT Functions

In this section, we propose a realization of RT functions using double-rail logic. We will review the evaluation method for f in the presence of unknown inputs.

A. DD-based Evaluation for Logic Functions

Definition 3.1 *[9] A BDD is a* Quasi Reduced Ordered Binary Decision Diagram *(QROBDD) if it does not contain distinct nodes v_1 and v_2 such that the subgraph rooted by v_1 and v_2 are equivalent, and if every path from the root node to the terminal nodes involves all the variables.*

Definition 3.2 *[9] A BDD is a* Reduced Ordered Binary Decision Diagram *(ROBDD) if it contains no node v with $low(v) = high(v)$, and if it does not contain distinct nodes v_1 and v_2 such that the subgraph rooted by v_1 and v_2 are equivalent.*

Example 3.1 *Figure 3.1(a) shows the binary decision tree for the function in Example 2.1. The number 0(1) attached to each edge incident to v denotes $low(v)$ ($high(v)$). Figure 3.1(b) and (c) show the QROBDD and ROBDD for the function in Example 2.1, respectively.* $\blacksquare$

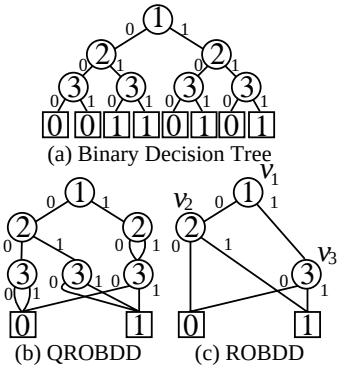


Figure 3.1: Example of BDDs

The next example explains the behavior of BDD-based logic simulator.

Example 3.2 *By tracing the edges of the ROBDD in Figure 3.1(c), we can evaluate $f(x_1, x_2, x_3) = f(0, 1, 1)$ for the function in Example 2.1. First, the label 1 of the node v_1 denotes the variable x_1. As the value of x_1 is 0, we trace the edge 0, then reach the node v_2. The label 2 of the node v_2 denotes the variable x_2. As the value of x_2 is 1, we trace the edge 1, then reach the terminal node $\boxed{1}$. Thus, we have $f(0, 1, 1) = 1$.* $\blacksquare$

In this way, the BDD-based simulator evaluates functions by tracing edges from the root node to a terminal node according to the value of each input variable.

In a practical system, the BDD-based simulator generates the code. For example, the code shown in Figure 3.2 is generated from Figure 3.1(c). Then, it is compiled, and executed. Some simulators generate native codes directly. Note that the simulators trace n edges for evaluating an n-variable logic function. In the cycle-based simulator using 2^p-valued decision diagrams, only n/p edges need to be traced[2]. However, these methods can treat only two values: 0 and 1. To treat u, we have to split u into both 0 and 1, as in Definition 2.2. We can formalize this by introducing the Alignment operation.

Definition 3.3 *Figure 2.1 shows the ternary operation Alignment. Let $a, b \in T$.*

$$a \odot b = \begin{cases} a & \textit{if} \quad a = b. \\ u & \textit{otherwise.} \end{cases}$$

Theorem 3.1

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

This means that we have to trace both *low* and *high* edges at the node where the input variable is equal to u. In

```
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 the code generated from the BDD.

the worst case, this method requires $O(2^s n)$ time, where s is the number of u's in the input vector. A software implementation on a uniprocessor will require exponential time with s. Kleene_TDD-based simulation requires only time $O(n)$. However, the Kleene_TDD requires $O(3^n/n)$ nodes, while the BDD requires $O(2^n/n)$ nodes.

In the next part, we will propose a hardware realization of RT functions. The propagation time of the network is $O(n)$, and the amount of hardware is $O(2^n/n)$. We also show a speedup method by pipelining.

B. Hardware Realization of RT Functions

When we trace the edges of BDDs, if an input variable corresponding to a node label is 0(1), then we go to the *low*(*high*) edge. If an input variable corresponding to the node label is u, then we need to evaluate both the *low* and the *high* edges, and need to perform alignment operation.

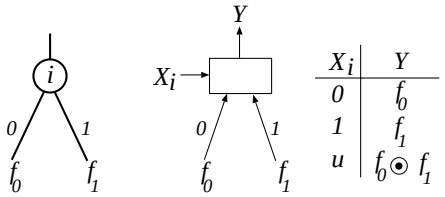
Table 3.1 shows an RT module implementing the operation in Figure 3.3(b). This module corresponds to a node of BDD (Figure 3.3(a)). Note that in Table 3.1, $(1,0)$, $(0,1)$, and $(1,1)$ represent 0, 1, and u, respectively. Figure 3.4(a) is the AND-OR double-rail logic network for Figure 3.3(b).

Algorithm 3.1 *A hardware realization of RT function: Given a BDD for a function f.*

1. *Replace each node of the BDD(Figure 3.3(a)) with an RT module (Figure 3.4(a)). Interconnect the RT modules by wires. For example, derive networks in Figure 3.5(a) and (b) from BDDs in Figure 3.1(a) and (b), respectively.*
2. *Assign $(f_L, f_H) = (1,0)$ to the terminal node $\boxed{0}$, and $(f_L, f_H) = (0,1)$ to the terminal node $\boxed{1}$.*
3. *Assign input variable X_i, $(X_{iL}, X_{iH}) = (1,0)$, $(0,1)$, and $(1,1)$ to 0, 1, and u, respectively.*
4. *Assign the output (Y_L, Y_H) of RT function, $(Y_L, Y_H) = (1,0), (0,1)$, and $(1,1)$ to 0, 1, and u, respectively.*

Definition 3.4 *Let $N(BDD : f)$ be the number of non-terminal nodes in the BDD for f.*

Theorem 3.2 *An RT function derived from n-input two-valued logic function f can be implemented by using $N(BDD, f)$ copies of RT modules.*



(a) A node of BDD. (b) RT module.

Figure 3.3: RT module.

Table 3.1: Truth table for an RT module.

X_{iL}	X_{iH}	f_{0L}	f_{0H}	f_{1L}	f_{1H}	Y_L	Y_H
1	0	1	0	-	-	1	0
1	0	0	1	-	-	0	1
1	0	1	1	-	-	1	1
0	1	-	-	1	0	1	0
0	1	-	-	0	1	0	1
0	1	-	-	1	1	1	1
1	1	1	0	1	0	1	0
1	1	0	1	0	1	0	1
1	1	otherwise				1	1
otherwise						-	-

- : don't care

Theorem 3.3 *Let $\tau(MSOP, f)$ be the number of products in MSOP(minimum sum-of-products expression) for f. $\tau(MSOP, Y_L) + \tau(MSOP, Y_H) = 2^n$.*

Theorem 3.4 *By assigning $(\bar{x}_i, x_i)$ to the variable (X_{iL}, X_{iH}) in Figure 3.5(a)(b), we have $Y_L = \bar{f}$ and $Y_H = f$.*

When RT functions are implemented by using sum-of-product expressions. Theorem 3.3 shows that the total number of products is 2^n. However, RT functions can be implemented with $O(2^n/n)$ cells of LUT (look-up-table) type FPGAs.

Xilinx FPGAs[10] include three major configurable elements: configurable logic blocks (CLBs), input/output blocks(IOBs), and interconnections. An XC4000 CLB contains two four-input lookup tables, a three-input lookup table, and two D-type flip-flops. Since outputs of an RT module, Y_L and Y_H, can be expressed in four-input logic functions, a CLB can implement an RT module. In this case, placing and routing are necessary but logic synthesis is trivial, since the network is directly mapped to the BDD.

C. Speedup by Pipelining

We often need simulators that evaluate the outputs not for a single stimulus but for a group of stimuli. By the proposed method, we can achieve a speedup with pipelined RT modules as shown in Figure 3.4(b). The pipelined RT module consists of one RT module and two D-FFs. In the pipelining network, the number of levels from the root node to terminal nodes must be same. Thus, we may construct the networks from a QROBDD. Figure 3.5(c) illustrates the realization of RT function with pipelined RT modules from Figure 3.1(b). A pipelined RT module can be also implemented by one CLB.

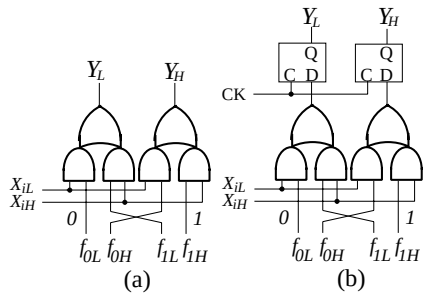


Figure 3.4: Double-rail realization for an RT module and a pipelined RT module.

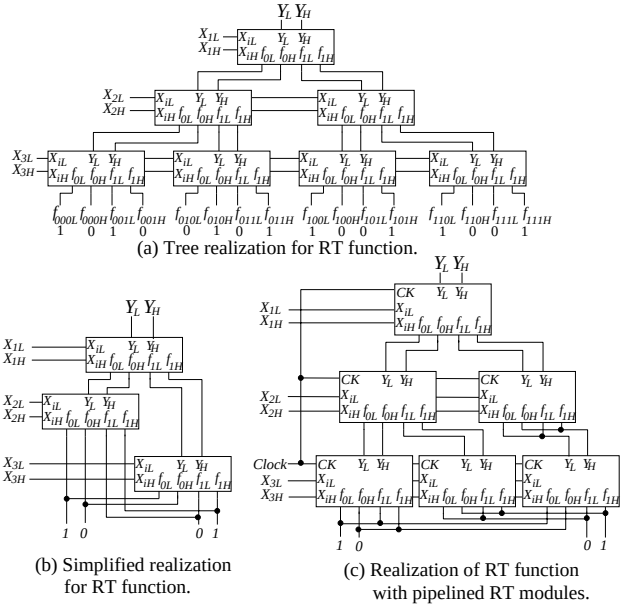


Figure 3.5: Realization of RT function.

IV. Experimental Results

We presented benchmark functions by shared ROBDDs. The ordering of the inputs variables for BDDs are obtained by heuristic algorithms[11]. We assumed XILINX FPGAs, and mapped RT functions to them by using Foundation Series 1.4 (XILINX tool). Table 4.1 compares the number of non-terminal nodes in the shared ROBDD, the number of CLBs, and simulation time[nsec/vector].

Table 4.1: Experimental results

function	in	out	nodes	CLBs	time
misj[1]	35	14	41	48	28
rckl[1]	32	7	190	178	95
b4[1]	33	23	205	149	45
x6dn[1]	39	5	219	328	84
jbp[2]	36	57	413	347	57
ti[2]	47	72	665	556	67
apex1[3]	45	45	1275	1057	147
apex7[3]	49	37	262	252	73
exep[3]	30	63	601	383	69
accpla[4]	50	69	1587	1290	169
mish[4]	95	43	103	130	22
signet[5]	39	8	1440	1295	216
c432[5]	36	7	1848	1777	340

[1]XC4010E-1, [2]XC4036XL-1, [3]XC4052XL-1, [4]XC4062XL-09, [5]XC40125XV-1

Simulation time was obtained by a static timing analyzer and a timing simulator (XILINX tool). XC40125XV, which is the biggest FPGA in this experiments, has $4624(68 \times 68)$ CLBs. Since the number of CLBs for each benchmark function is less than 4624, each function is mapped to single FPGA. We acheieved a speed up of 10 over a software simulator that cannot treat unknown inputs[2], and of 100 over a software simulator that can treat unknown inputs[4]. The use of the pipelining technique will improve the performance.

V. Conclusion

This paper showed a method to evaluate logic functions in the presence of unknown inputs. To produce correct values, we computed regular ternary(RT) functions derived from given switching functions. We proposed a realization of RT functions by using double-rail logic. This implementation requires $O(2^n/n)$ logic cells, and $O(n)$ time. We showed FPGA realizations are about 100 times faster than software simulation.

Acknowledgments

This work was supported in part by a Grant in Aid for Scientific Research of the Ministry of Education, Science, Culture and Sports of Japan. Prof. Jon T. Butler's comments were useful to improve the presentation.

References

[1] P. Ashar, and S. Malik, "Fast functional simulation using branching programs," *ICCAD'95*, pp.408–412, Nov. 1995.

[2] 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.

[3] 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.

[4] Y. Iguchi, T. Sasao, and M. Matsuura, "On properties of Kleene TDDs," *Asia and South Pacific Design Automation Conference, Proc. ASP-DAC'97*, pp.473–476. Jan. 1997.

[5] S. C. Kleene, *Introduction to Metamathematics*, Wolters-Noordhoff, North-Holland Publishing, 1952.

[6] 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.

[7] R. E. Bryant, "Symbolic simulation – techniques and applications," *Proc. of 27th Design Automation Conf.*, pp.517–521, June 1990.

[8] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. Comput.*, Vol. C-35, no. 8, pp. 677–691, Aug. 1986.

[9] T. Sasao, and M. Fujita, (ed.): "*Representations of Discrete Functions*," Kluwer Academic Publishers, 1996.

[10] "XC4000E and XC4000X Series FPGAs," http://www.xilinx.com/partinfo/databook.htm# xc4000, Nov. 1997.

[11] N. Ishiura, H. Sawada, and S. Yajima, "Minimization of binary decision diagrams based on exchanges of variables," *ICCAD'91*, pp. 472–475, 1991.