A PC-Based Logic Simulator Using a Look-Up Table Cascade Emulator

Hiroki NAKAHARA (a), Student Member, Tsutomu SASAO (b), and Munehiro MATSUURA (c), Members

SUMMARY This paper represents a cycle-based logic simulation method using an LUT cascade emulator, where an LUT cascade consists of multiple-output LUTs (cells) connected in series. The LUT cascade emulator is an architecture that emulates LUT cascades. It has a control part, a memory for logic, and registers. It connects the memory to registers through a programmable interconnection circuit, and evaluates the given circuit stored in the memory. The LUT cascade emulator runs on an ordinary PC. This paper also compares the method with a Levelized Compiled Code (LCC) simulator and a simulator using a Quasi-Reduced Multi-valued Decision Diagram (QRMDD). Our simulator is 3.5 to 10.6 times faster than the LCC, and 1.1 to 3.9 times faster than the one using a QRMDD. The simulation setup time is 2.0 to 9.8 times shorter than the LCC. The necessary amount of memory is 1/1.8 to 1/5.5 of the one using a QRMDD.

key words: LUT cascade, bdd for j, functional decomposition

1. Introduction

With the increase of the integration of LSIs, the time for the verification of the design increases. Thus, high-speed logic simulators are needed.

Logic simulators can be roughly divided into two types: event-driven simulators and cycle-based simulators. In an event-driven simulator, only the outputs of the gates whose input signals change are evaluated. On the other hand, in a cycle-based logic simulator, the operation order of gates are determined statically beforehand, and all the outputs of the gates are evaluated for each clock cycle. Although the cycle-based logic simulator does not perform the timing verification, it is often faster than the event-driven simulator.

An LCC [1] is a kind of a cycle-based logic simulator using a general-purpose CPU. An LCC generates a program code for each gate of a logic circuit, and evaluates the circuit in a topological order from the inputs towards the outputs. In this paper, we will present a cycle-based logic simulator using an LUT cascade emulator. An LUT cascade emulator [2] consists of a control part, memories, and registers. Each register is connected to a programmable interconnection circuit, and the LUT cascade emulator evaluates the logic circuit stored in the memory. Murgai-Hirose-Fujita [10] also developed a logic simulator using large memories. Their method first converts a given circuit into a random logic network of single-output LUTs, then stores them in the memory, and finally evaluates the circuit by an event-driven logic simulator implemented by a hardware accelerator. In our method, we first convert the given circuit into a cascade rather than random logic, so the control part is simpler than Murgai-Hirose-Fujita’s method. Also, our method uses multiple-output LUTs rather than single-output LUTs. In this paper, we consider a software-based logic simulation system where the LUT cascade emulator is simulated on a PC. Compared with the hardware-based logic emulator, a logic simulator using a standard PC is much cheaper, and can be enhanced with the improvement of the performance of PCs. Our simulator outperforms commercial logic simulators [13].

This paper is an extended version of [13].

2. LUT Cascade Emulator

Figure 1 shows a model of a sequential circuit, where \( X \) denotes inputs, \( Z \) denotes outputs, \( Y \) denotes the inputs to flip-flops, \( Y' \) denotes the outputs of flip-flops, and \([Y]\) denotes the number of state variables. We first introduce an LUT cascade [3] that realizes the combinational part of a sequential circuit, then introduce the LUT cascade emulator that emulates the LUT cascade.

2.1 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. Also, each cells may have external outputs in addition to the rail outputs. In this paper, \( X_i \) denotes the external inputs to the \( i \)-th cell; \( Y'_i \) denotes the state inputs to the \( i \)-th cell; \( Z_i \) denotes the external outputs of the \( i \)-th cell;

![Fig. 1 A model for a sequential circuit.](image-url)
for the multiple-output function $y$ 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_CFR, we ignore the edges that incident to 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_CFR for the multiple-output function $\vec{f} = (f_1, f_2, \ldots, f_m)$, and $W$ be the width of the BDD_for_CFR at the height $(X_1, Y_1)$ in Fig. 3. By applying functional decompositions to $\vec{f}$, we obtain the network in Fig. 4, where the number of lines connecting two blocks is $t = \lceil \log_2 W \rceil$ [4].

**Theorem 2.1:** [5] Let $\mu_{\text{max}}$ be the maximum width of the BDD_for_CFR that represents an $n$-input logic function $\vec{f}$. If $u = \lceil \log_2 \mu_{\text{max}} \rceil \leq k - 1$, then $\vec{f}$ can be realized by a circuit shown in Fig. 4, where $|X_1| = k$. By applying functional decompositions $s - 1$ times, we have the cascade having the structure of Fig. 2.

### 2.2 LUT Cascade Emulator

Figure 5 shows an LUT cascade emulator for a sequential circuit.

An LUT cascade emulator stores the cell data of an LUT cascade in the Memory for Logic. The address of cell data is calculated from inputs, state variables, and rail outputs of the preceding cell. The LUT cascade emulator reads the cell outputs from the memory for logic, and send them to the State Register and the Output Register. The Input Register stores the values of the primary inputs; the MAR (Memory Address Register) stores the address of the memory; the MBR (Memory Buffer Register) stores the outputs of the memory; the Programmable Interconnection Network connects the input register, the state register, and the MBR to the MAR; the Memory for Interconnection stores data for the interconnections; Memory for Page Address stores data for the page address; and the Control Net-
work generates necessary signals to obtain functional values.

To emulate a sequential circuit, the LUT cascade emulator stores state variables and output variables in the registers. Figure 6 shows the Double-Rank Flip-Flop for the state register and the output register, where $L_1$ 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.

3. Synthesis of the LUT Cascade Emulator

The data for a BDD_for_CF can be too large to be stored in a memory of the computer. 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 large BDD_for_CF is time consuming. In this paper, we first partition the given circuit into groups, and then construct a BDD_for_CF for each group.

Previous approach [13] partitions the output functions into groups so that the total number of cells is minimized. The method [13] uses a simple heuristic method to partition the outputs quickly. However, when the BDD_for_CF representing a single output function is excessively large, this method fails.

In this paper, we partition the circuit rather than the outputs. Although we have to introduce connection signals between groups, we can represent the circuits that are too large for the previous method [13].

3.1 Graph Representation of a Circuit

To partition the circuit, we represent the given circuit by a directed-graph. We replace logic modules with nodes, and interconnections with edges. Also, we divide feedback lines into feedback inputs and feedback outputs, and replace them with edges.

Definition 3.5: A primary input node denotes a primary input or a feedback input. A primary output node denotes a primary output or a feedback output. A logic module node denotes a logic module.

Example 3.1: Figure 8 illustrates a graph representing the circuit in Fig. 7. In Fig. 8, $x_0, x_1, x_2, x_3,$ and $z_1'$ are primary input nodes, and $z_0$ and $z_1$ are primary output nodes.

$a_0, a_1, a_2, a_3,$ and $a_4$ are logic module nodes.

Definition 3.6: Let $A$ be a set of logic module nodes. The input nodes for $A$ are the nodes that have incident edges to $A$, denoted by $In(A)$. Similarly, the output nodes for $A$ are the nodes that have incidented edges from $A$, denoted by $Out(A)$.

3.2 Partition of a Circuit

We formulate the partition problem for the given circuit as follows:

Problem 3.1: Suppose that the given circuit is represent by a graph. Let $A$ be the set of the logic module nodes in the graph. Then, partition the set $A$ into subsets $A_j$ ($j = 1, 2, \ldots, g$) as follows:

1. $A_i \cap A_j = \phi$ ($i \neq j$), $\bigcup_{j=1}^{g} A_j = A$.
2. $A_j$ can be realized by an LUT cascade.
3. $Node(A_j) \leq ThNode$, where $ThNode$ denotes the maximum number of nodes for a group, and $Node(A_j)$ denotes the number of nodes in the BDD_for_CF that represents $A_j$.

Although several partitioning algorithms (e.g. by liner programming, or by dynamic programming) have been reported [18]–[20], they require long computation time. In this paper, we trade the partition time and the quality of the partitioned circuits.

Algorithm 3.1: (Partition the Circuit into Groups and Construct BDD_for_CFs) Let $A$ be the set of the logic module nodes that represent the given circuit, $g$ be the number of block in the partition, $X$ be the set of the primary input
nodes, $A_g$ be the set of nodes under selection, $B$ be the set of nodes after selection, $C$ be the set of candidate nodes, and $ThNode$ be the maximum number of nodes for each BDD for CF.

1: $B \leftarrow X, C \leftarrow Out(X), g \leftarrow 0, A_0 \leftarrow \phi.$
2: while ($C \neq \phi$)
3: Select $a_j \in C$ such that $|In(A_g \cup a_j)| + |Out(A_g \cup a_j)|$
4: Construct the BDD for CF that represents $A_g \cup a_j$
5: if $(Node(A_g \cup a_j) \leq ThNode$ and $(A_g \cup a_j$ can be realized by a cascade))
6: $A_g \leftarrow A_g \cup a_j, C \leftarrow C \cup Out(a_j) - a_j,$
7: else 
8: $B \leftarrow B \cup \{A_g\}$
9: $g \leftarrow g + 1$
10: $A_g \leftarrow a_j, C \leftarrow Out(a_j)$
11: }
12: }
13: Terminate.

Algorithm 3.1 finds a logic module node that minimizes the total number of inputs and outputs nodes (line 3). Then, it constructs the BDD for CF, and checks whether the number of nodes in BDD for CF is less than the $ThNode$ or not (line 5). Also, it checks whether the group can be realized by a cascade or not (line 5). Algorithm 3.1 partitions the given circuit quickly, since it searches for the nodes of the circuit only once.

3.3 Memory Packing

By Algorithm 3.1, we represent a given multiple-output function by a set of BDD for CFs. Then, we construct the LUT cascades for them, and then store the LUT data into the memory of the LUT cascade emulator.

Example 3.2: Figure 9 shows an LUT cascade consisting of 4-input cells. Figure 10(a) illustrates the memory map of cell data, where the memory for logic has 6-bit address inputs, and each word consists of four bits. The dark areas in the figure are unused, and $P_i$ denotes the page number.

In Example 3.2, each cell data is stored in a separate page of the memory. The data of a cell must be stored in the same page, and must be read simultaneously. If there are any extra space in the same page, then multiple cell data can be stored in the same page. This method to reduce the memory area is memory-packing [6].

Example 3.3: In Fig. 10(a), by storing the cell data $r_5$ and $z_1$ to Page 1, we have the memory map in Fig. 10(b), where a half of the memory is enough to store all the data. (End of Example)

4. Logic Simulation on an LUT Cascade Emulator

4.1 Generation of the Execution Code for Simulation

Figure 11 shows the logic simulation system using an LUT cascade emulator. First, it partitions the Verilog-VHDL netlist-code describing the given circuit, and constructs BDD for CFs by Algorithm 3.1. Then, it reduces the number of nodes of BDDs by optimizing variable orders [7]. Next, it generates LUT cascades from BDDs using functional decompositions described in Chapter 2, and it maps them into the memory of the LUT cascade emulator. Also, it generates the C code that describes the control circuit of the LUT cascade emulator. Next, it compiles the C code into the execution code for simulation of the LUT cascade emulator. And, finally the simulator on a PC evaluates the outputs of the given circuit by using the memory of the LUT cascade emulator.

4.2 Program Code for the LUT Cascade Emulator

This system generates the program code that describes the following operations:

Step 1 Set the input register, and initialize the state register.
Set the input values to the input register. Also, initialize to values of the state register.

Step 2 Evaluate each cell.

Step 2.1 Simulate the programmable interconnection network.
Generate the address of the memory for logic from the values of the input register, the state register, the MBR, and the page address.

Step 2.2 Read the memory for logic.
Read the content of the memory for logic using the address generated in Step 2.1.

Step 2.3 Distribute the output values of the memory for logic.
Send the values read in Step 2.2 to the output register and to the state register.

Step 3 Perform the state transition.
Update the output values of the state register by using $S_{\text{Clock}}$.
Sending each memory output to each register usually
consumes CPU time. Fortunately, the memory outputs are stored in the order of primary outputs, state outputs, and rail outputs. For a 32-bit processor, we can evaluate up to 32 outputs at a time. To obtain required outputs, we shift the memory outputs covered by a mask, and assign into a 32-bit variable. In this way, we can evaluate the multiple output simultaneously. Also, there is an additional merit for performing the state transition. Let $|Y|$ be the number of state variables of the given logic function, then the necessary number of evaluations for the state transition is $\lceil \frac{|Y|}{32} \rceil$ for a 32-bit machine.

Since cascades have many fewer signal lines than the original circuit, the compilation time for cascades are much shorter than that of the conventional LCC method.

### 4.3 Analysis of Simulation Time

When an LUT cascade emulator is implemented on a dedicated hardware [2], the evaluation time is proportional to the number of cells. However, when an LUT cascade emulator is implemented on a standard PC, we need extra time, since the inputs and outputs of a cell must be evaluated sequentially.

To do high-speed simulation for an LUT cascade emulator on a PC, we consider two objects:

a. Reduction of the number of cells.
   This can be done by increasing the number of inputs of each cell. However, the increase of the number of inputs of each cell also increases the evaluation time per cell, which will be explained later.

b. Reduction of the number of cell inputs.
   This decreases the evaluation time per cells, but increases the number of cells.

To find the best strategy, we did the following experiments. We implemented 10 MCNC benchmark functions [8] on the LUT cascade emulator. By changing the maximum number of inputs for cells, we obtained the average number of cell inputs, the number of cells, and the execution time of the LUT cascade emulator. Figure 12 shows the experimental results, where the horizontal axis denotes the maximum number of cell inputs; 0 denotes the lower bound on the maximum number of inputs of cells, that is $\lceil \log_2 \mu_{\text{max}} \rceil + 1$; the vertical axis denotes the ratios of the number of cells, the number of the average cell inputs, and simulation time. We set 1.00 to the ratios when the number of cell inputs is $\lceil \log_2 \mu_{\text{max}} \rceil + 1$.

Figure 12 shows that the simulation time increases with the number of cell inputs. The reason for this will be analyzed in Sect. 5.3. Therefore, our strategy is to reduce the number of cell inputs in the LUT cascade emulator.

### 5. Experimental Results

We implemented Algorithm 3.1 and the simulation system described in Sect. 4.1 in the C programming language. Then, we compared our method with other simulation methods with respect to the simulation time, the simulation setup time, and the size of memories.

#### 5.1 The Benchmark Functions

Table 1 shows the benchmark functions [15], [16] used for simulation. Name denotes the name of the benchmark function; In denotes the number of primary inputs; Out denotes the number of primary outputs; FF denotes the number of flip-flops; and Gate denotes the number of gates.

#### 5.2 Comparison with LCC

We implemented the LCC simulator in the C programming language. Table 2 compares our method with the LCC. Name denotes the name of benchmark function; Cas denotes the number of LUT cascades; Cell denotes the total number of cells; ASM denotes the number of instructions in the assembly code; Code denotes the code image size (kilo bytes); E.in denotes the average number of external inputs to cells; P.out denotes the total number of external output(s); and S.out denotes the total number of state output(s). Sim denotes the evaluation time (sec). In order to obtain the raw simulation time, we generated one million random test vectors, and obtained the time excluding the time for reading and writing vectors. Setup denotes the setup time (sec) for the simulation. Setup of LCC includes the time for the C-code generation and the compilation, while Setup of the LUT cascade emulator includes the time for partition the circuit, BDD generation, LUT cascade synthesis, memory mapping, C-code generation, and the compilation. Literals denotes the total number of literals.
in expressions of the C-code generated by the LCC. Ratios denote that of the simulation setup time and that of the simulation execution time (LCC/LUT cascade emulator). To produce the executable code for LCC, we used gcc compiler with optimization option -O3. Also, we generated program codes for LUT cascade emulator, and compiled them with the same conditions as LCC. In the experiments, we used an IBM PC/AT compatible machine, Pentium4 Xeon 2.8 GHz, L1 Data Cache: 8 KB, L1 Instruction Cache: 12,µops, L2 Cache: 512 KB, Memory: 4 GByte, and OS: Redhad (Linux 7.3).

Table 2 shows that the LUT cascade emulator is 3–10 times faster than the LCC. Also, the setup for the LUT cascadce emulator is 2–9 times faster than the LCC. Since the size of C-code for b17, b18, and b22 were too large, gcc could not optimize the codes with the option. Although we could simulate these benchmarks when we removed the optimize option for gcc, the simulation times were too long. Thus, we excluded these data from Table 2. The code image sizes for the LCC are larger, since the LCC converts all the gates and signals into the C-code. On the other hand, the code image size for the LUT cascade emulator is smaller, since the LUT cascade emulator partition the given circuit into the memory for logic, and only the C-code that emulates the control part is generated. Although the LUT cascade emulator requires extra memory, they can be stored in the memory of our PC.

To analyze the difference of the simulation time, we compare the estimated values with the experimental values. The number of operations in the LUT cascade emulator is estimated as follows:

\[ \text{EST.Cas} = \text{E.in} \times \text{Cell} \]

where, \( \text{Rail} = \text{Cell} - \text{Cas} \). The first term of expression (1) denotes the setup time of all the external inputs of the cells; the second term denotes the access time to the memory for logic; the third term denotes the setup time for the output register; the fourth term denotes the setup time for the state register; and the last term denotes the setup time for the rail inputs. In Fig. 13, the right vertical axis denotes the experimental value SIM.Cas (sec), and the left vertical axis denotes the estimated number of operations EST.Cas. Also, we conjecture that \( \text{Literals} \) is proportional to the simulation time for LCC. In Fig. 14, the right vertical axis denotes the experimental value SIM.LCC (sec), and the left vertical axis denotes the estimated number of literals EST.LCC = \( \text{Literals} \).

Figures 13 and 14 show that the SIM.Cas and SIM.LCC can be estimated from EST.Cas and EST.LCC, respectively.

In Figs. 13 and 14, we can see that SIM.Cas is smaller than SIM.LCC, and also EST.Cas (number of operations) is smaller than EST.LCC (number of literals). To analyze these results, we converted the C-codes of the LUT cascade emulator and the LCC into the assembly-instructions. Figure 15 compares the numbers of assembly-instructions and the estimated values. In Fig. 15, the vertical axis denotes the number of instructions. The number of the assembly-instructions for both methods are larger than the numbers of C instructions. Especially, that of the LCC increased. This is because the LCC compiler generates extra codes to evaluate negative literals and logic gates, and to produce the output

---

**Table 1**

<table>
<thead>
<tr>
<th>Name</th>
<th>In</th>
<th>Out</th>
<th>FF</th>
<th>Gate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wb_dma</td>
<td>216</td>
<td>215</td>
<td>521</td>
<td>3389</td>
<td>DMA Bridge IP Core</td>
</tr>
<tr>
<td>mem_ctrl</td>
<td>132</td>
<td>1021</td>
<td>1449</td>
<td></td>
<td>Memory Controller</td>
</tr>
<tr>
<td>usb_func</td>
<td>113</td>
<td>1204</td>
<td>1368</td>
<td></td>
<td>USB Function Core</td>
</tr>
<tr>
<td>b14</td>
<td>32</td>
<td>54</td>
<td>243</td>
<td>10098</td>
<td>A copy of b14</td>
</tr>
<tr>
<td>b15</td>
<td>36</td>
<td>70</td>
<td>449</td>
<td>8932</td>
<td>A copy of b15</td>
</tr>
<tr>
<td>b17</td>
<td>37</td>
<td>97</td>
<td>1415</td>
<td>32326</td>
<td>A copy of b14 and two of b17</td>
</tr>
<tr>
<td>b18</td>
<td>36</td>
<td>23</td>
<td>320</td>
<td>11462</td>
<td>A copy of b14 and two of b17</td>
</tr>
<tr>
<td>b20</td>
<td>32</td>
<td>22</td>
<td>490</td>
<td>40226</td>
<td>A copy of b14 and a modified version of b14</td>
</tr>
<tr>
<td>b21</td>
<td>32</td>
<td>22</td>
<td>490</td>
<td>20571</td>
<td>Two copies of b14</td>
</tr>
<tr>
<td>b22</td>
<td>32</td>
<td>22</td>
<td>735</td>
<td>29951</td>
<td>Two copies of b14</td>
</tr>
</tbody>
</table>

**Table 2**

<table>
<thead>
<tr>
<th>Name</th>
<th>LUT cascade emulator</th>
<th>LCC</th>
<th>Ratios</th>
</tr>
</thead>
<tbody>
<tr>
<td>wb_dma</td>
<td>13</td>
<td>581</td>
<td>3.2</td>
</tr>
<tr>
<td>mem_ctrl</td>
<td>109</td>
<td>1745</td>
<td>3.5</td>
</tr>
<tr>
<td>usb_func</td>
<td>47</td>
<td>1429</td>
<td>4.1</td>
</tr>
<tr>
<td>b14</td>
<td>41</td>
<td>1081</td>
<td>2.8</td>
</tr>
<tr>
<td>b15</td>
<td>38</td>
<td>1005</td>
<td>3.5</td>
</tr>
<tr>
<td>b17</td>
<td>121</td>
<td>4134</td>
<td>3.9</td>
</tr>
<tr>
<td>b18</td>
<td>273</td>
<td>7386</td>
<td>4.7</td>
</tr>
<tr>
<td>b20</td>
<td>70</td>
<td>2009</td>
<td>3.0</td>
</tr>
<tr>
<td>b21</td>
<td>79</td>
<td>2167</td>
<td>3.1</td>
</tr>
<tr>
<td>b22</td>
<td>121</td>
<td>2953</td>
<td>3.2</td>
</tr>
</tbody>
</table>
signals. In the LCC, its operands frequently move between registers and the memory. For the gate with fan-outs, the LCC stores the output values of the gate into a variable temporarily, and uses it as the input of two or more gates. On the other hand, the LUT cascade emulator uses only the rail values stored in a single register variable. Therefore, only the input register, the output register, the memory for logic, and the connections for each group require memory references. Experimental results show that the simulator based on an LUT cascade emulator is 3–10 times faster than the LCC. One reason for this is the difference of the representations: the cascade has many fewer signals than the random logic network. Another reason is the CPU architecture of the PC. The access time of the data in the main memory is about 200 times longer than one in the L1 cache. So, the CPU time heavily depends on the frequency of cache misses. In the case of the LCC simulator, the circuit data and control are mixed, and the instruction data is too large to be stored in the data cache. On the other hand, in the case of an LUT cascade emulator, the cascade data and control data are separated. Control data is in the instruction cache, while the cascade data is in the data cache. Thus, we can expect fewer cache misses in the LUT cascade emulator.

Figures 16 and 17 show the relation between the simulation setup time and the code image size. In these figures, the right vertical axis denotes the simulation setup time (sec), and the left vertical axis denotes the code image size (kilo bytes). These figures show that the code image size affects the simulation setup time. The code image size for the LUT cascade emulator is smaller than that of the LCC. Therefore, the LUT cascade emulator is faster than the LCC with respect to the simulation setup time. Note that, in the LUT cascade emulator, we need data for logic in addition to the code.

5.3 Comparison with the QRMDD

In this part, we compare with an MDD-based logic simulator. As for the definitions on MDD (Multi-valued Decision Diagram), refer [14], [21]. Let \( (X_1, X_2, \ldots, X_u) \) be the input variables. When all \( X_i \) \((i = 1, 2, \ldots, u)\) appear in this order in all paths of an MDD \((k)\), the MDD \((k)\) is a QRMDD \((k)\) (Quasi-Reduced Multi-valued Decision Diagram) with \(k\) bits. The length of an arbitrary path in a QRMDD \((k)\) is equal to \(u\), the number of input variables. Note that, a QRMDD usually has redundant nodes. By combining the
binary nodes of a BDD into a multi-valued node of $2^k$ inputs, we obtain a QRMDD ($k$) [14].

We can generate a code to evaluate the a QRMDD: Store a QRMDD in a table, and use a generic program to evaluate the QRMDD [11]. For example, a table for a QRMDD (2) is obtained from a BDD in Fig. 5.4. Also, Example 5.5 shows the pseudo-code for evaluating a QRMDD (3).

**Example 5.4:** From a BDD (Fig. 18(a)), by combining the nodes into multi-valued nodes, we have an MDD (Fig. 18(b)). From the MDD (Fig. 18(b)), we have a QRMDD (2) (Fig. 18(c)). From the QRMDD (2) (Fig. 18(c)), we have a table for QRMDD (2) (Fig. 18(d)).

**Example 5.5:** (Pseudo-code to evaluate QRMDD (3))

1. $ptr \leftarrow \text{root\_index}; i \leftarrow 0$
2. $node\_input \leftarrow \{x_1, x_2, x_3\}$
3. $ptr \leftarrow \text{table}[ptr + node\_input]$
4. $i \leftarrow i + 1$
5. If ($i < \text{the length of the path for QRMDD (3)})$ then go to 2.
6. Terminate.

The code to evaluate a QRMDD ($k$) is quite similar to the code to evaluate an LUT cascade emulator. Also, the procedures for pre-computing the circuit beforehand are almost the same. The difference between them is a data structure for the function. The QRMDD ($k$)-based method represents the circuit by multiple QRMDDs ($k$), and stores them to the table. On the other hand, the LUT cascade emulator represents it by multiple LUT cascades, and stores them to the memory for logic, using memory packing.

Table 3 compares the LUT cascade emulator with the QRMDD ($k$). Name, Cell, Code, Setup, and Sim are the same as Table 2. Rail denotes the average number of rails for the LUT cascade emulator; Mem denotes the amount of memory (kilo bytes); Path denotes the total path length for each QRMDD ($k$); Width denotes the average widths for QRMDDs ($k$); and Table denotes the total memory size of tables (kilo bytes). The environment for the experiment is the same as the case of the LCC. Also for each benchmark, the LUT cascade and the QRMDD ($k$) are generated from the same BDDs. The setup time for the QRMDD ($k$) includes the time for partition the circuit, BDD generation, QRMDD ($k$) generation, table generation, C-code generation, and the compilation. Ratios denote that of the simulation setup time, that of the simulation execution time, and that of the memory size (QRMDD ($k$)/LUT cascade emulator).

Since the simulation time for the QRMDD ($k$) is minimum when $k = 3$, we set to $k = 3$.

Table 3 shows that the LUT cascade emulator is 1.1–3.9 times faster than the QRMDD (3)-based simulator. The setup for the LUT cascade emulator is also faster than that of the QRMDD (3). The memory size for QRMDD (3) is 1.8–5.5 times larger than that of the LUT cascade emulator.

To evaluate the memory size, we compare an estimated memory size with an actual memory size. Let $k_i$ be the number of external inputs of $i$-th cell, $\mu_i$ be the number of rail inputs, and $c_i$ be the number of cells. Note that, $\mu_i$ denotes the number of rail inputs to the first cell, so $\mu_0 = 0$. Let $\tilde{M}_{\text{Cas}}$ be the size of the memory for logic in the LUT cascade emulator. Then, we have

$$
\tilde{M}_{\text{Cas}} = 2^{k_i} + \bar{\mu}_1 + 2^{k_2} + \bar{\mu}_2 + \cdots + 2^{k_c} + \bar{\mu}_c
$$

By replacing $\mu_i (j = 0, 1, \ldots, c)$ with $\bar{\mu}$, the average of $\mu_j$, and $k_i (j = 1, 2, \ldots, c)$ with $k$, the average of $k_j$, we have the following approximation:

$$
\tilde{M}_{\text{Cas}} = 2^{k+\bar{\mu}}
$$

Figures 19 illustrates a node for a QRMDD ($k$), and Fig. 20 illustrates nodes with respect to $X_j$. Let $p$ be the path length, $w_j (j = 0, 1, \ldots, p)$ be the width of the QRMDD ($k$) with respect to $X_j$. Note that, $w_0$ denotes the width on a root node, and $w_0 = 1$. As shown in Fig. 20, one node for a QRMDD ($k$) can be represented by the table storing $2^k$ pointers to the next nodes. Let $a$ be the number of bits for the pointer, then the memory size for the table representing a node in a QRMDD ($k$) is $2^k a$.

---

**Table 3** Comparison with the QRMDD (3).

<table>
<thead>
<tr>
<th>Name</th>
<th>LUT cascade emulator</th>
<th>QRMDD (3)</th>
<th>Ratios</th>
</tr>
</thead>
<tbody>
<tr>
<td>wbalu</td>
<td>1745</td>
<td>5.5</td>
<td>208</td>
</tr>
<tr>
<td>ncu</td>
<td>1429</td>
<td>4.7</td>
<td>213</td>
</tr>
<tr>
<td>b10</td>
<td>1081</td>
<td>5.4</td>
<td>201</td>
</tr>
<tr>
<td>b15</td>
<td>1833</td>
<td>5.8</td>
<td>239</td>
</tr>
<tr>
<td>b18</td>
<td>7967</td>
<td>4.6</td>
<td>554</td>
</tr>
<tr>
<td>b20</td>
<td>2009</td>
<td>5.2</td>
<td>193</td>
</tr>
<tr>
<td>b21</td>
<td>2167</td>
<td>5.4</td>
<td>198</td>
</tr>
<tr>
<td>b22</td>
<td>2953</td>
<td>5.2</td>
<td>270</td>
</tr>
</tbody>
</table>

---

**Fig. 18** BDD for $F = x_1x_2x_3$ (a), MDD for $F$ (b), QRMDD(2) for $F$ (c), Table for QRMDD (2) (F. d).
Let $M_{QRMDD}$ be a size of table for a QRMDD ($k$). Then, we have $M_{QRMDD} = 2^a k \sum_{i=1}^{p} w_i$. Let $\bar{w}$ be the average of $w_i (j = 0, 1, \ldots, p)$. Then, we have following approximation:

$$M_{QRMDD} = 2^a k \bar{w}$$  \hspace{1cm} (3)

When $a = 32$, the experimental results show that $M_{QRMDD}$ is almost the same as actual size of the table. However, $M_{Cas}$ is larger than the actual size of the memory for logic. To investigate this fact, we obtained the size of memory for logic without memory packing. Figure 21 compares $M_{QRMDD}$, $M_{Cas}$, and the sizes of memory for logic with and without memory packing. The vertical axis denotes the memory size (kilo byte). $NonPack_Mem$ denotes the size of memory for logic without memory packing; $Pack_Mem$ denotes the size of memory for logic with memory packing; $EstMem$ denotes the estimated size $M_{Cas}$; $EstQTable$ denotes the estimated size $M_{QRMDD}$. Figure 21 shows Pack is 1.9 to 2.1 times smaller than NonPack_Mem. Also, NonPack_Mem is almost equal to EstMem. Another reason for the difference of the memory sizes is due to the difference of $\bar{w}$ and $\bar{\mu}$. When we generate an LUT cascade, we select functional decomposition that minimizes $\bar{\mu}$ using a dynamic programming [6], while to construct a QRMDD (3), we do not optimize the size of decompositions. Therefore, $\bar{\mu}$ is smaller than $\bar{w}$. From Eqs. (2) and (3), these differences affected the memory size.

Both methods perform logic simulation by accessing the data stored in the memory. Thus, the simulation time is affected by the time for accessing memory, the number of memory accesses, and the time for handling the read data. The memory access time heavily depends on the frequency of cache misses. When a cache miss occurs, the CPU accesses the main memory and reads the data. The CPU manages the main memory per page. First, it convert a virtual address into a physical address using a special cache called TLB (Translation Lookaside Buffer), next it accesses the main memory at a high speed and reads the data [17]. Therefore, the size of data stored in a memory affects the memory access time. Figure 22 shows an influence of memory packing on simulation time. In Fig. 22, $Pack_Sim$ denotes the simulation time (sec) with memory packing; $NonPack_Sim$ denotes the simulation time (sec) without memory packing. Since smaller memory tends to have fewer cache misses, Pack_Sim is faster than NonPack_Sim. To predict the ratio of simulation time, we define the ratio of simulation time for LUT cascade emulator and QRMDD ($k$)-based simulator as follows:

$$EST_{ratio} = \frac{\alpha \text{Path} + \beta M_{QRMDD}}{\alpha \text{Cell} + \beta M_{Cas}}$$

where Path denotes the path length, Cell denotes the number of cells, $M_{Cas}$ denotes the size of memory for the LUT cascade emulator, and $M_{QRMDD}$ denotes the size of table for the QRMDD ($k$). Figure 23 compares $SIM_{ratio}$ with $EST_{ratio}$, where

$$SIM_{ratio} = \frac{\text{Simulation time for QRMDD (3)-based simulator}}{\text{Simulation time for LUT cascade emulator}}$$

The vertical axis denotes the ratio of the simulation time. In this figure, we set $\alpha = 100$ and $\beta = 1$. Figure 23 shows that $EST_{ratio}$ has the same tendency as $SIM_{ratio}$ except for mem_ctrl. In mem_ctrl, many of adjacent cells are stored in the same page of the memory for logic. Since adjacent
cells are read continuously, the miss rate of the cache for 
mem_ctrl was low, and the simulation time is short, and 
SIM_ratio is high. We can reduce the simulation time, if we 
perform memory packing so that adjacent cells are stored in 
the same page of the memory.

The simulation setup time for both methods are almost 
same, since the difference of the code image size are also 
almost the same.

6. Conclusion and Comments

In this paper, we showed a cycle-based logic simulator using 
the LUT cascade emulator running on a standard PC. This 
method first converts the circuit into LUT cascades. Then, 
it stores the LUT data in the memory of the LUT cascade 
emulator. Next, it generates the program code for the control 
circuit of the LUT cascade emulator. This paper also 
compares the method with a LCC simulator and a simulator 
using a QRMDD. Our simulator is 3.5 to 10.6 times faster 
compares the method with a LCC simulator and a simulator

The proposed method is a kind of a cycle-based simulator. Note that, special primitives, such as tri-state buffer, are not implemented in the current version.

One of the future projects is to develop a efficient 
mixed simulator using cycle-based simulation and event-
driven simulation.

Acknowledgments

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

References

[1] M. Abramovici, M.A. Breuer, and A.D. Friedman, Digital Sys-
1994.
sequential circuit using LUT ring,” IEE Trans. Fundamentals, 
multiple-output function for reconfigurable hardware,” IWLS-2001, 
logic functions,” Proc. Design Automation Conference, pp.428–433, 
San Diego, CA, USA, June 2004.
look-up table rings,” IWLS-2004, pp.431–437, Temecula, Califor-
nia, USA, June 2004.
[8] S. Yang, Logic synthesis and optimization benchmark user guide 
[9] R.E. Bryant, “Graph-based algorithms for boolean function manip-
1986.
telli, and P. Scaglia, “Fast discrete function evaluation using decision 
Logic Synthesis and Verification, ed. S. Hassan and T. Sasao, 
ing an LUT cascade emulator,” ASPDAC 2006, pp.466–465, Yoko-
hama, Jan. 2006.
[17] J.L. Hennessy and D.A. Patterson, Computer Architecture: A Quan-
for microelectronics,” IEEE Trans. Syst. Cybern., vol.SSC-4, 
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.