社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

LUT カスケード・エミュレータを用いた論理シミュレーションについて

中原　啓貴† 　笹尾　勤†† 　松浦　宗寛††

† 九州工業大学大学院 情報創成工学専攻
†† 九州工業大学 情報工学部
〒 820–8502 福岡県飯塚市大字川津 680–4

あらまし　概要 本論文では, LUT (Look-Up Table) カスケード・エミュレータを用いたサイクルベース形式論理シミュレーションについて述べる. LUT カスケード・エミュレータとは, 複数の LUT (セル) を直列に接続した LUT カスケードを模擬するアーキテクチャである. LUT カスケード・エミュレータは制御部とメモリとレジスタを持ち, 書き換え可能な接続回路でメモリと各レジスタの接続を行い, メモリのアドレスを計算する. そして, メモリに格納したセルを読み出す. 以上を繰り返して論理回路を評価する. 本手法は, LUT カスケード・エミュレータを汎用の PC 上でソフトウェアを用いて実現する. シミュレーション実行時間とシミュレーション準備時間について, 本手法と Levelized Compiled Code (LCC) との比較を行い, 本手法は 18 倍〜2621 倍高速であった.

キーワード　BDD_for_CF, 関数分解, LUT カスケード

A Logic Simulation using an Look-Up Table Cascade Emulator

Hiroki NAKAHARA†, Tsutomu SASAO††, and Munehiro MATSUURA††

† Program of Creation Informatics, Kyushu Institute of Technology
†† Department of Comoputer Science and Eloctronics, Kyushu Institute of Technology
kawazu 680–4, Iizuka, Fukuoka, 820–8502 Japan

Abstract　abstract This paper shows a cycle-based logic simulation method using an LUT cascade emulator. The LUT cascade emulator is an architecture that emulates LUT cascades , where multiple-output LUTs (cells) are connected in series. The LUT cascade emulator has a control part, a large memory, and registers. It connects the memory to each register by a programmable interconnection circuit, and evaluates the given circuit stored in the memory. This method realizes the LUT cascade emulator on a PC by a software. Experimental results show that this method is 18-2621 times faster than the LCC.

Key words　BDD_for_CF, Functional decomposition, LUT cascade

1. はじめに

LUT の大規模化・高集積化に伴い, 設計期間に占める検証の割合が増大しており, 高速な論理シミュレータが求められている.

論理シミュレータは論理回路の論理機能やタイミングを検証するためのツールであり, イベントドリブン (event driven) 形式とサイクルベース (cycle based) 形式の二つに大別できる. イベントドリブン形式論理シミュレータは, 入力信号の変化 (イベント) のある論理ゲートを評価し, その出力を次段の論理ゲートに伝搬させ評価を行っていく. また, 論理機能の検証だけではなく, イベント間の遅延時間を評価することもできる. しかしながら, 配置配線設計前は配線経路が確定しないため, 仮配線長を頼りに, 遅延時間を計算する. 近年, LSI の微細化に伴い, 配線間の遅延時間の見積もりが困難になっており [13], 仮配線長によるタイミング検証は現実的ではない. 一方, サイクルベース形式論理シミュレータは, 論理ゲートの遅延時間は無視し, トポロジカル順序で各ゲートをクロックサイクル毎に一度だけ評価する. タイミング検証は行えないが, 論理検証に特化した分, イベントドリブン形式と比較してはるかに演算量が少なく, 数倍〜100 倍高速に論理機能検証が可能である.

LCC [1] は汎用 CPU を用いたサイクルベース形式論理シミュレータの一種である. LCC では, 論理回路の各ゲートにプログラムコードを割り当て, 入力から出力に対してトポロジカル順序で評価を行う. 本論文では, LUT カスケード・エミュレータを用いたサイクルベース形式論理シミュレータについて述べる. LUT カスケード・エミュレータは制御部とメモリとレジスタを持ち, 書き換え可能な接続回路でメモリと各レジスタを接続し, メモリに格納した論理回路を評価する. 文献 [11] で, Murgai-Hirose-Fujita はメモリを用いた論理シミュレータ

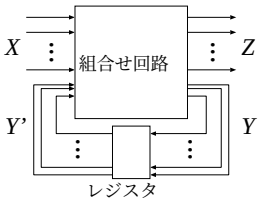

図 1　順序回路の模式図

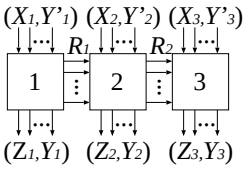

図 2　LUT カスケード

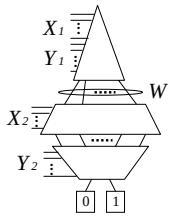

図 3　BDD_for_CF

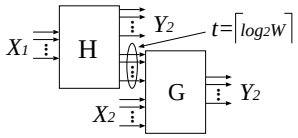

図 4　中間出力のある関数分解

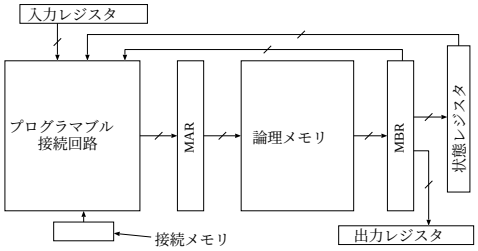

図 5　LUT カスケード・エミュレータ

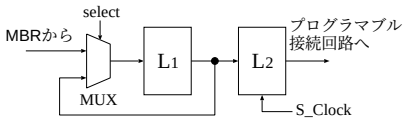

図 6　ダブルランク・フリップ・フロップ

について述べている. Murgai らの手法は, 論理回路を単一出力 LUT のランダム回路に変換し, それらをメモリに格納した後, ハードウェア・アクセラレータを用いて評価を行う. 本手法は, 論理回路を LUT カスケードに変換する. LUT カスケードは LUT (セル) 間の接続を直列接続に限定しているため, Murgai らの手法と比較して制御回路が単純である. また, 本手法は多出力 LUT を用いるため, 多数の出力を同時に評価できる. 本論文では, LUT カスケード・エミュレータを汎用の PC 上でソフトウェアを用いて実現する手法について述べる.

2. LUT カスケード・エミュレータ

2.1 LUT カスケード

図 1 に順序回路の模式図を示す. X は外部入力, Y' は状態入力を表し, Y は状態出力, Z は外部出力を表す. 本手法は順序回路の組合せ部を **LUT カスケード** [3] で表現し, **LUT カスケード・エミュレータ**で評価を行う.

LUT カスケードを図 2 に示す. LUT カスケードは複数の LUT (**セル**) を直列に接続した構造を持つ. 各セル間に接続されている信号線を**レール**と呼ぶ. 各セルはレール出力の他に, 外部出力を持つ. 本論文では X_i をセル i の外部入力, Y_i' をセル i の状態入力, Z_i をセル i の外部出力, Y_i をセル i の状態出力, R_{i-1} をセル i の**レール入力**, R_i をセル i の**レール出力**とする. LUT カスケードは, 多出力論理関数を表現する **BDD_for_CF** に対して**関数分解**を繰り返し適用することにより得られる.

[定義 2.1]　入力変数を $X = (x_1, x_2, \ldots, x_n)$ とし, 多出力関数を $\vec{f} = (f_1(X), f_2(X), \ldots, f_m(X))$ とする. **多出力関数の特性関数**を $\chi(X, Y) = \bigwedge_{i=1}^{m} (y_i \equiv f_i(X))$ とする.

n 入力 m 出力関数の特性関数は, $(n+m)$ 変数の 2 値論理関数であり, 入力変数 $x_i (i = 1, 2, \ldots, n)$ の他に, 各出力 f_i に対して出力変数 y_i を用いる. $B = \{0, 1\}$, $\vec{a} \in B^n$, $\vec{b} \in B^m$, $F = (f_1(\vec{a}), f_2(\vec{a}), \ldots, f_m(\vec{a})) \in B^m$ とすると,

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

である.

[定義 2.2]　関数 f が変数 x に依存するとき, x を f の**サポート変数**という.

[定義 2.3]　多出力関数 $\vec{f} = (f_1, f_2, \ldots, f_m)$ の BDD_for_CF とは, $\vec{f}$ の特性関数 χ を表現する BDD (Binary Decision Diagram) [10] である. ただし, BDD の変数は, 根節点を最上位としたとき, 変数 y_i は f_i のサポート変数の下に置く.

[定義 2.4]　多出力関数 $\vec{f}$ を表現する BDD_for_CF の x_k と x_{k+1} の間を交差する枝の数を**第 k 段目での BDD_for_CF の幅**という. ここで, 同じ節点を指している枝は 1 と計数する. また, 幅を計数する際, 出力を表現する変数から, 定数 0 に向かう枝は無視する.

X_1, X_2 を入力変数の集合, Y_1, Y_2 を出力変数の集合とする. 多出力関数 $\vec{f} = (f_1, f_2, \ldots, f_m)$ を表現する BDD_for_CF の変数順序を (X_1, Y_1, X_2, Y_2) とするとき, BDD_for_CF の X_2 の最上位と Y_1 の最下位の間における幅を W とする (図 3). 多出力関数 $\vec{f}$ を関数分解することにより図 4 の回路構造で実現可能である. ここで, 2 つの回路 H と G の間の接続線数は $t = \lceil \log_2 W \rceil$ である [4].

[定理 2.1]　[3] n 変数多出力関数 $\vec{f}$ を実現する BDD_for_CF の幅の最大値を μ_{max} とする. $u = \lceil \log_2 \mu_{max} \rceil \leqq k - 1$ ならば, $\vec{f}$ は図 2 で示すような k 入力-LUT カスケード回路で実現可能である. 図 2 は関数分解を 2 回繰り返して得られた回路である.

2.2 LUT カスケード・エミュレータ [2]

図 2 に LUT カスケード・エミュレータを示す. LUT カスケード・エミュレータは, LUT カスケードのセルデータを格納する論理メモリ, 外部入力の値を保持する**入力レジスタ**, メモリ番地を保持する **MAR**(Memory Address Register), メモリの出力値を保持する **MBR**(Memory Buffer Register), **プログラマブル接続回路**, **接続メモリ**, データのシフトを行う**シフタ**, 状態変数を記憶する**状態レジスタ**, 外部出力を記憶する**出力レジスタ**, **制御回路**から構成されている. また, 順序回路を評価するため, 状態レジスタは状態入力と状態出力を保持しなければならない. 図 6 に状態レジスタを構成する, **ダブルランク・フリップ・フロップ**を示す. L_1, L_2 はそれぞれ D ラッチである. カスケードの評価後に select 信号を ON にし, L_1 に状態変数の値を格納する. 全カスケードの評価終了後, S_Clock にパルスを加え, 状態変数の値を L_2 に転送することにより, 状態遷移が終了する.

[例 2.1]　図 7 に, LUT カスケード・エミュレータを用いた順序回路の模擬を示す. ただし, 図 7 において, X_i は入力変数を,

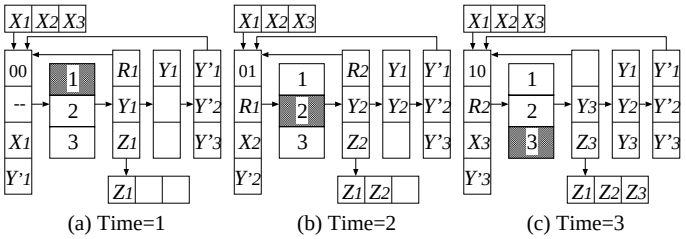

図 7　LUT カスケード・エミュレータを用いた順序回路の模擬

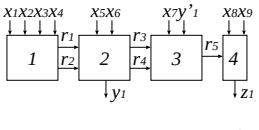

図 8　LUT カスケードの例

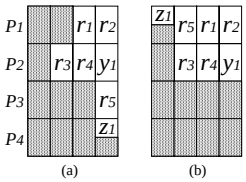

図 9　メモリ・パッキングの例

Y_i は状態変数を, Z_i は外部出力を, R_i はレール変数を示す. また, 順序回路の組合せ部は図 7 の LUT カスケードで実現されているものとする.

$time = 1$ でセル 1 の評価を行う. セル 1 を参照するため, MAR の上位 2 ビットを 00 にセットし, プログラマブル接続回路を介してセル 1 の入力変数 (X_1, Y_1') をセットする. セル 1 を読み, 出力値を MBR にセットする. 外部出力 (Z_1) を出力レジスタにセットし, 状態出力 (Y_1) を状態レジスタにセットする (図 7(a)).

$time = 2$ でセル 2 の評価を行う. セル 2 を参照するため, MAR の上位 2 ビットを 01 にセットし, プログラマブル接続回路を介してフィードバックされたセル 1 のレール出力 (R_1) とセル 2 の入力変数 (X_2, Y_2') をセットする. セル 2 を読み, 出力値を MBR にセット後, 各出力をそれぞれのレジスタにセットする (図 7(b)).

$time = 3$ でセル 3 の評価を行う. セル 3 を参照するため, MAR の上位ビットを 10 にセットし, プログラマブル接続回路を介して入力変数をセットする. セル 3 を読み, 出力値を MBR にセット後, 各出力をそれぞれのレジスタにセットする.

すべてのセルが評価されたので, 制御回路は S_Clock を状態レジスタと出力レジスタに送り, 状態レジスタの値 (Y_1, Y_2, Y_3) を状態入力レジスタに転送する. また, 出力レジスタの値 (Z_1, Z_2, Z_3) を外部出力として出力する (図 7(c)). (例終り)

3. LUT カスケード・エミュレータの合成

3.1 出力の分割

一般に, 多出力関数を単一の BDD_for_CF で表現した場合, 節点数が増大し, コンピューターのメモリ上に格納できない場合が多い. また, BDD_for_CF の幅も増大し, LUT カスケードの実現ができない場合も多い. 多出力関数を単一の BDD_for_CF で表現できても, 最適化に多大な時間を必要とするので効率が悪い. よって, 出力関数を複数のグループに分割し, 各出力集合を個別の LUT カスケードで実現することを考える.

出力関数をグループ化する際, なるべく小さな BDD_for_CF を構成するためにサポート変数が増えないように出力関数を分割する.

［定義 3.5］ 論理関数の集合を $F = \{f_1, f_2, \ldots, f_m\}$ とし, $G \subseteqq F$ とし, $f_i \in F - G$ とする. G に**最も類似した出力** f_i とは

$$Similarity(i, G, F) = |Sup(f_i) \cap Sup(G)|, \qquad (1)$$

の値を最大にする f_i のことである. ここで, $Sup(F)$ は F のサポート変数の集合を示す.

次に, 与えられた多出力論理関数の出力を分割するアルゴリズムを示す.

［アルゴリズム 3.1］ (出力の分割と BDD_for_CF の構成)

論理関数の集合を $F = \{f_1, f_2, \ldots, f_m\}$, 分割後の出力関数の部分集合の集合を $\mathcal{Z} = \{G_1, G_2, \ldots, G_r\}$, 分割の個数を r, 分割された出力関数の集合を表現する BDD_for_CF の節点数のしきい値を Th とする.

1. $\mathcal{Z} \leftarrow \phi$, $r \leftarrow 0$.
2. While $F \neq \phi$, do Steps (a)-(d).

(a) $Node \leftarrow 0$, $G_r \leftarrow \phi$.

(b) While $Node \leqq Th\_node$, $F \neq \phi$, and G_r is cascade realizable, do Steps i-iii.

i. Select f_i with the maximum $Similarity(i, G_r, F)$. If $G_r = \phi$, then select f_i that has the largest support.

ii. $G_r \leftarrow G_r \cup \{f_i\}$, $F \leftarrow F - \{f_i\}$.

iii. Construct BDD_for_CF that realizes G_r, and $Node \leftarrow (the\ number\ of\ nodes)$.

(c) If G_r is not cascade realizable, then $G_r \leftarrow G_r - \{f_i\}$, $F \leftarrow F \cup \{f_i\}$.

(d) $\mathcal{Z} \leftarrow \mathcal{Z} \cup \{G_r\}$, $r \leftarrow r + 1$.

3. Terminate.

本手法はカスケード実現可能かつ, BDD の節点数がしきい値を越えない間, 出力を併合する. 状態遷移関数などのサポート変数が少ない関数を多く併合する場合, セル外部出力数がセル出力制限数を越えるため, カスケード実現が不可能となる. また, 算術関数などのサポート変数が多い関数を併合する場合, BDD のサイズがメモリに格納できなくなる. アルゴリズム 3.1 は出力をグループに分割するため, 得られた BDD は非常に小さい.

3.2 メモリ・パッキング

アルゴリズム 3.1 を用いて, 与えられた論理関数の出力を複数のグループに分割し, 各グループを BDD_for_CF で表現し, LUT カスケードで実現する. 次に, LUT のデータを LUT カスケード・エミュレータの論理メモリに格納する.

［例 3.2］ 図 8 に 4 入力セルで構成された LUT カスケードを示す. 図 9(a) にセルデータを格納したメモリマップを示す. ここで, メモリのアドレス数は 6, ワード数は 4 であり, 斜線部は未使用領域を示す. また, P_i はページ番号である. (例終り)

例 3.2 では, 1 ページ毎にセルデータを 1 つ格納している. そのため, メモリ領域の半分以上が未使用である. 一般に, 同一段のセルデータは同時に読み出す必要があり, 同一ページ内に格納する必要がある. しかし, 空きがあれば, 同一ページ内に異なる段のセルデータを格納できる. これにより必要メモリ量を削減できる. これをメモリ・パッキングと呼ぶ [6].

［例 3.3］ 図 9(a) において, セルデータ r_5 と z_1 をページ 1 に格納することにより, 図 9(b) に示されたメモリマップを得る. メモリ・パッキングを用いることにより, 必要メモリ量を半減できた. (例終り)

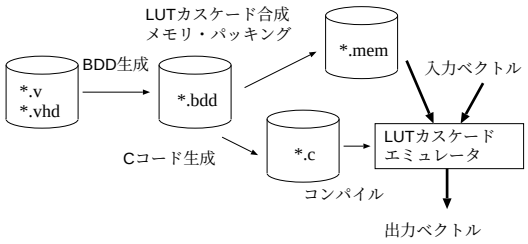

図 10　LUT カスケード・エミュレータ論理シミュレーションシステムのデータフロー

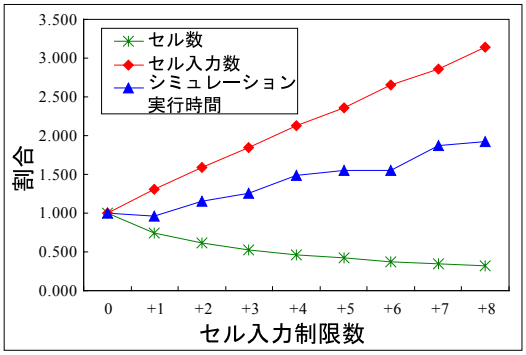

図 11　セル入力制限数とシミュレーション時間の関係

4. LUT カスケード・エミュレータを用いた論理シミュレーション

4.1 シミュレーション実行コードの生成

Fig. 10 に LUT カスケード・エミュレータ論理シミュレーションシステムのデータフローを示す.

まず, 回路を記述した Verilog-VHDL を Shared-BDD [9] に変換する. 次に, 変数順序変更 [7] を行い, BDD のノード数を削減し, BDD を最適化する. そして, 関数分解を繰り返し適用し, BDD から LUT カスケードを合成し, セルデータを PC 上のメモリに格納する. 同時に, LUT カスケード・エミュレータの制御回路を模擬する C コードを生成し, コンパイルを行って実行コードを生成する. シミュレーションシステムはメモリと実行コードを用いて回路の論理シミュレーションを行う.

4.2 LUT カスケード・エミュレータを模擬するプログラムコード

シミュレーションシステムは以下の命令を記述したプログラムコードを生成する.

Step 1　外部入力を入力レジスタにセットし, 状態レジスタを初期化する.

Step 2　各セルの評価を行う.

Step 2.1　プログラマブル接続回路の模擬を行う. ページアドレス, 入力レジスタ, 状態レジスタ, および MBR から論理メモリのアドレスを生成する.

Step 2.2　Step 2.1. で生成したアドレスにアクセスし, セルデータを読み出す.

Step 2.3　Step 2.2 で読み出したデータを出力レジスタと状態レジスタに代入する.

Step 3.　全てのセルの評価が終了したので, 状態遷移を行う. 状態出力レジスタの値を状態入力レジスタに代入し, Step 1 に戻る.

ソフトウェア実現では, メモリ出力を逐一レジスタに代入しなければならない. しかしながら, 本手法はメモリ出力を, 外部出力, 状態出力, レール出力の順に並べているので各出力にマスクを掛けシフトすることで, 同時代入できる. 32 ビットプロセッサ実現では, 最大で 32 個の出力が同時評価可能である. また, 状態遷移も高速に行うことができる. 論理回路の状態変数の個数を $|Y|$ とすると, 32 ビットプロセッサ実現では, 状態遷移を $\lceil \log_2 \mu_{max} \rceil + 1$ 回で行える. カスケード構造を採用するもう一つの利点として, LCC などのランダム回路と比較して, LUT カスケードの信号線数は少ないので, コンパイル時間を削減可能である.

4.3 シミュレーション実行時間の解析

LUT カスケード・エミュレータをハードウェア実現した場合, 評価時間はセル数に比例する [2]. しかしながら, ソフトウェア実現の場合, 入出力信号線の同時評価を行うことができず, 逐次評価を行わなければならない.

ソフトウェア実現において, LUT カスケード・エミュレータの高速シミュレーションを行うには, 二つの背反した方法が考えられる.

a.　セル数を削減する

セルの入力数を増やすことで, セル数を削減することができる. しかしながら, セルの入力数を増やすことにより, 各セル毎の評価時間が増大してしまう.

b.　セルの入力数を削減する

この方法では, 各セル毎の評価時間は削減できるが, セル数が増加してしまう.

どちらの方法が有効な方法であるか検証するため, 10 個の MCNC ベンチマーク関数 [8] を LUT カスケード・エミュレータ上に実現し, 予備実験を行った. そして, セル入力制限数を変化させ, 平均セル入力数, セル数, LUT カスケード・エミュレータの実行時間を求めた. 図 11 に実験結果を示す. 横軸はセルの最大入力数を示し, 0 はセル入力制限数の下界 $\lceil \log_2 \mu_{max} \rceil + 1$ である. 縦軸は平均セル入力数, セル数, シミュレーション時間の割合を示し, セル入力制限数が $\lceil \log_2 \mu_{max} \rceil + 1$ のときに, それぞれ 1.000 とした.

図 11 より, 平均セル入力数が増加するに伴ってシミュレーション時間も増加している. 本手法は論理メモリのアドレス計算に時間を費す. 計算時間はセル入力数の増加に伴って増加する. 従って, この結果より, 本手法はセル入力数を削減する方法で, LUT カスケード・エミュレータを合成する.

5. 実験結果

5.1 MCNC ベンチマーク関数のシミュレーション結果

LUT カスケード・エミュレータと LCC を PC 上に実現し, いくつかの MCNC ベンチマーク関数のシミュレーションを行った. 表 1 に実験結果を示す. *Name* はベンチマーク関数名を, *In* は外部入力数を, *Out* は外部出力数を, *State* は状態変数の個数を, *Cas* は合成された LUT カスケードの本数を, *Cell* はセル数を, *Mem* は必要メモリ量を表し, 単位は (Mega Bits) である. また, *EXT.in* はセルの平均入力数を, *P.out* は外部出力を持つセルの個数を, *S.out* は状態出力を持つセルの個数を示す *Sim* は 100 万個のランダムテストベクトルのシミュレーション実行時間であり, 単位は (sec) である. *Setup* はシミュレーション準備時間であり, 単位は (sec) である. LCC のシミュレーション準備時間は C コード生成時間, 及びコンパイル時間である. 一方, LUT カスケード・エミュレータのシミュレーション準備時間は

BDD 生成時間, LUT カスケード合成時間, メモリマッピング時間, C コード生成時間, 及びコンパイル時間である. *Literals* は LCC によって生成された C コードの論理式のリテラル数の総数であり, *Ratio* は LCC と LUT カスケード・エミュレータのシミュレーション準備時間とシミュレーション実行時間の比率を表し, (LCC/LUT カスケード・エミュレータ) である. 実験環境は, IBM PC/AT 互換機, Pentium4 Xeon 2.8GHz (L1 Cache: 8KB, L2 Cache: 512KB), メモリ 4GByte, であり, OS は Redhad (Linux 7.3) である.

表 1 より, LUT カスケード・エミュレータは LCC より 18〜2621 倍高速にシミュレーションを実行できた. また, シミュレーション準備時間も s5378 を除いて 1.2〜57 倍高速であった. ベンチマーク関数 *clma* は他のベンチマーク関数と比較して *Ratio* が多かった. これは, *clma* はゲート間の信号線を多く持つが, 各論理関数のサポート変数の個数が少ないからである. 分割した BDD_for_CF で *clma* を表現した場合, 幅が狭く, サポート変数が少ないので, 小さな LUT カスケードで実現でき, 高速にシミュレーションを実行できた. この事実を確認するため, 追加実験を行った. まず, *clma* の元の BLIF を BDD に変換し, 最適化を行った. 次に, BDD の各ノードをマルチプレクサに置き換えた BLIF を出力した. LCC でこの BLIF を実現したところ, *Literals* は 2518 に減少し, シミュレーション実行時間は 5.74(sec) であった. *Ratio* を再計算すると, 44.15 であり, 他のベンチマーク関数の結果とほぼ同じ傾向を示した.

5.2 シミュレーション実行時間の見積もりと考察

LUT カスケード・エミュレータの実行時間を命令数で見積もった. LUT カスケード・エミュレータの命令数は以下の式で求められる.

$$EST.LUT = EXT.in \times Cell + Cell + P.out + S.out + Rail, \quad (2)$$

ただし, $Rail = Cell - Cas$ である. 式 2 の第 1 項は全外部入力数を, 第 2 項は論理メモリのアクセス回数を, 第 3 項は外部出力数を, 第 4 項は状態出力の個数を, 最終項はレール入力数をそれぞれ表す.

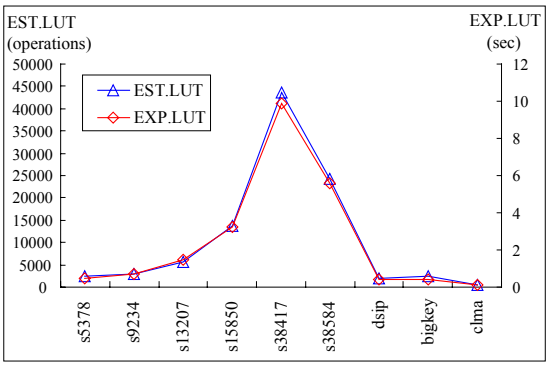

図 12　LUT カスケード・エミュレータのシミュレーション実行時間

図 12 において, 右縦軸は LUT カスケード・エミュレータのシミュレーション実行時間の実験値 EXP.LUT を表し, 単位は (sec) である. 左縦軸は LUT カスケード・エミュレータのシミュレーション実行時間の見積もり値である命令数 EST.LUT を表す. また, LCC のシミュレーション実行時間を *Literals* で見積もった. 図 13 において, 右縦軸は LCC のシミュレーション実行

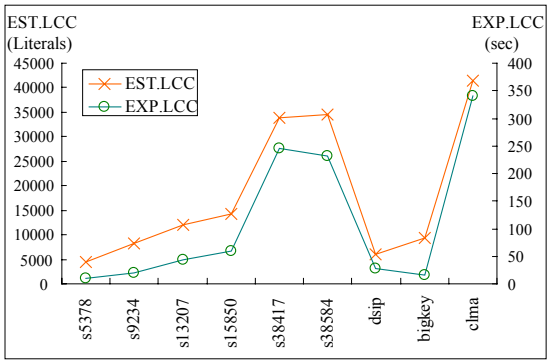

図 13　LCC のシミュレーション実行時間

時間の実験値 EXP.LCC を表し, 単位は (sec) である. 左縦軸は LCC シミュレーション時間の見積もり値である *Literals* の個数 EST.LCC を表す. 図 12, 13 より, EXP.LUT と EXP.LCC はそれぞれ, EST.LUT と EST.LCC で見積もることができる.

図 12, 13 において注目するべき点は, EXP.LUT は EXP.LCC より小さいが, いくつかのベンチマーク関数において, EST.LUT (命令数) が EST.LCC (リテラル数) より大きいことである. 検証を行うため, これらの C コードをアセンブラコードに変換し, 解析を行った. LCC のアセンブラコード量は EST.LCC よりも数倍多かった. その一方で, LUT カスケード・エミュレータのアセンブラコード量は EST.LUT よりも少なかった. これは LCC と LUT カスケードの回路構造によるものである. LCC はランダム回路で表現するため, 複数のファンアウトを持つゲートを評価する際, ゲートの出力値を保持しなければならない. 従って, ゲートの出力値がたびたび呼び出されるため, CPU のメモリ-レジスタ参照を行う命令を必要とする. また, 否定リテラルの評価のための命令も必要である. 一方, LUT カスケード・エミュレータはセル間の接続線は前後のセル間のレールに限られており, しかも CPU のレジスタ 1 つで評価できるため, メモリ-レジスタ間の参照は入出力数とセル数に限られる. 実験結果が見積もり値と相違していたもう一つの理由として, CPU のアーキテクチャが考えられる. 一般に, メインメモリのアクセスタイムは L1 キャッシュのアクセスタイムよりも約 200 倍遅い. したがって, CPU の計算時間はキャッシュミスに非常に影響を受ける. LCC シミュレータの場合, 回路データと制御部は混在しており, キャッシュのサイズと比較して命令データは非常に大きい. その一方で, LUT カスケード・エミュレータは, 回路データ (セルデータ) と制御部が分離されており, 制御部は命令キャッシュに, 一方で, 回路データ (セルデータ) はデータキャッシュに格納される. 従って, LUT カスケード・エミュレータのほうがキャッシュミスは少ないと考えられる.

5.3 シミュレーション準備時間の考察

図 14 は, LUT カスケード・エミュレータのシミュレーション準備時間における各工程の割合を示す. 横軸はシミュレーション準備時間における各工程の割合を示し, それぞれ, BDD_for_CF 生成と最適化 (BDD), LUT カスケード合成 (Cascade), C コード生成とコンパイル (Compile), 及びメモリパッキング (Packing) である. 縦軸はベンチマーク関数名であり, *overall* は全ベンチマーク関数の平均値を表す. 図 14 より, BDD 生成と最適化は時間のかかる工程である. また, C コード生成とコンパイルも時間のかかる工程であり, ほとんどがコンパイルに費され

表 1　MCNC ベンチマーク関数のシミュレーション結果

Name	In	Out	State	LUT カスケード・エミュレータ								LCC			Ratio	
				Cas	Cell	Mem [Mbit]	EXT.in	P.out	S.out	Setup [sec]	Sim [sec]	Literals	Setup [sec]	Sim [sec]	Setup	Sim
s5378	35	49	164	40	543	0.82	2.39	105	28	14.59	0.49	4424	10.26	10.46	0.70	21.35
s9234	36	39	211	44	599	0.91	2.63	26	120	14.68	0.71	8220	41.01	20.64	2.79	29.07
s13207	62	152	638	93	1245	3.28	2.27	109	390	31.09	1.46	11954	83.04	43.76	2.67	29.97
s15850	77	150	534	105	3370	8.95	1.95	115	338	79.25	3.25	14328	115.72	58.71	1.46	18.06
s38417	28	106	1636	389	9411	45.36	2.55	60	964	763.07	9.87	33769	917.40	245.63	1.20	24.89
s38584	38	304	1426	270	5118	16.90	2.55	232	956	159.09	7.61	34485	830.27	230.76	5.22	30.33
dsip	229	197	224	45	473	6.60	1.96	108	115	10.90	0.42	5959	26.39	27.08	2.42	64.48
bigkey	263	197	224	48	541	3.76	1.97	171	121	11.74	0.42	9262	27.58	16.26	2.35	38.71
clma	383	82	33	11	129	0.25	2.05	41	21	3.64	0.13	41353	207.76	340.75	57.08	2621.15

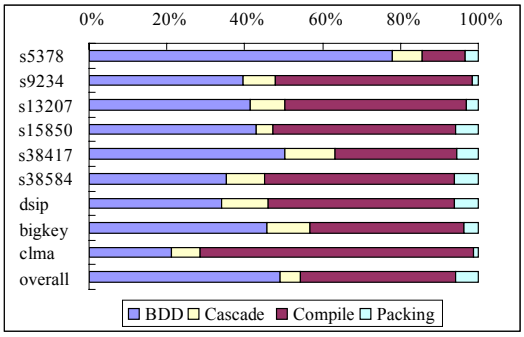

図 14　シミュレーション準備時間における各工程の割合

る. コンパイル時間を削減するにはコードサイズを削減すればよく, 例として, セル数を削減する事が挙げられる. しかしながら, セル数を削減するにはセル入力数を増やさなければならないが, セル入力数の増加は, メモリパッキング時間の増加や必要メモリ量の増加を招く. セル入力数を k, セル出力数を u とすると, 各セルに必要なメモリ量は $2^k \cdot u$ である. 従って, セルの入力数を増加させるとメモリ量やメモリの値を計算する時間が増加する. 本論文中での実験は, セル入力数がなるべく小さくなるように合成したため, メモリパッキングはシミュレーション準備時間に影響を及ぼさなかった.

6. 結　　論

本論文では, LUT カスケード・エミュレータを用いた論理シミュレーションについて述べた. まず, 論理回路を BDD で表現し, LUT カスケードを合成する. 次に, PC 上のメモリにセルデータを格納する. そして, 制御回路を模擬する C コードを生成し, コンパイルを行って実行コードを生成する. シミュレーション実行時間とシミュレーション準備時間について, 本手法と LCC との比較を行い, 本手法は 18 倍〜2621 倍高速であった.

7. 謝　　辞

本研究は, 一部, 文部科学省・科学研究費補助金, 日本学術振興会・科学研究費補助金, および, 文部科学省・北九州地域知的クラスター創成事業の補助金による.

文　　献

[1] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design,* Wiley-IEEE Press; Rev. Print edition, Sept. 1994.

[2] T. Sasao, H. Nakahara, M. Matsuura and Y. Iguchi, "Realization of sequential circuits by look-up table ring," *MWSCAS2004*, Hiroshima, July 25-28, 2004, pp.I517-I520.

[3] T. Sasao, M. Matsuura, and Y. Iguchi, "A cascade realization of multiple-output function for reconfigurable hardware," *IWLS-2001*, Lake Tahoe , CA, June 12-15,2001, pp.225-300.

[4] T. Sasao and M. Matsuura, "A method to decompose multiple-output logic functions," *Proc. Design Automation Conference*, San Diego, CA, USA, June 2-6, 2004, pp.428-433.

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

[6] T. Sasao, M. Kusano, and M. Matsuura, "Optimization methods in look-up table rings," *IWLS-2004*, June 2-4, 2004, Temecula, California, USA, pp.431-437.

[7] R. Rudell, "Dynamic variable ordering for ordered binary desicion diagrams," *ICCAD'93*, pp. 42-47, 1993.

[8] S. Yang, Logic synthesis and optimization benchmark user guide version 3.0, MCNC, Jan. 1991.

[9] S. Minato, N. Ishiura, S. Yajima, "Shared binary decision diagram with attributed edges for efficient boolean function manipulation," *Proc. Design Automation Conference*, June 1991, pp.52-57.

[10] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transact. Comput.*, C-35, Aug.1986, pp.677-691.

[11] R. Murgai, F. Hirose, and M. Fujita, "Logic synthesis for a single large look-up table," *ICCD1995*, pp.415-424, Oct. 1995.

[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. K. Brayton, "The future of logic synthesis and verification," *in S. Hassoun and T. Sasao (e.d.), Logic Synthesis and Verification, Kluwer Academic Publishers, 2001.*