DA−90−11

サイクルベース・シミュレーション・エンジンの一構成法

伊勢野 総[†]　井口 幸洋[†]　笹尾 勤[‡]　松浦 宗寛[‡]

[†] 明治大学 理工学部 情報科学科
〒214-8571 川崎市多摩区

[‡] 九州工業大学 情報工学部 電子情報工学科
〒820-8502 飯塚市川津 680-4

E-MAIL: {iseno,iguchi}@cs.meiji.ac.jp, {sasao, matsuura}@cse.kyutech.ac.jp

あらまし－BDDに基づくサイクルベース・シミュレータのハードウェア化を行なった. 高速なシミュレーション・エンジンの構成法を提案する. エンジンは, BDDデータを格納する為のメモリと, 入力値に応じてBDDの枝を辿るための制御回路で構成する. 変数毎にこれらのユニットを用意することでパイプライン化を行なう. 予備的な実験から, 提案するエンジンはソフトウェアによる高速なシミュレータに比べ, 非パイプライン化エンジンでおよそ2～4倍, パイプライン化エンジンで$10n$倍程度(nは入力変数の数)の高速化が期待できる.

キーワード －シミュレーション・エンジン, 二分決定グラフ, 論理シミュレーション, FPGA

A Realization of Cycle-based Simulation Engine

Atsumu ISENO[†]　Yukihiro IGUCHI[†]　Tsutomu SASAO[‡]　Munehiro MATSUURA[‡]

[†]Dept. of Computer Science
Meiji University
Kawasaki 214-8571, JAPAN

[‡]Dept. of Electronics and Computer Science
Kyushu Institute of Technology
Iizuka 820-8502, JAPAN

E-MAIL: {iseno, iguchi}@cs.meiji.ac.jp, {sasao, matsuura}@cse.kyutech.ac.jp

Abstract － We propose a cycle-based simulation engine, which is based on a software cycle-based simulator. This engine consists of RAMs and control circuits. BDD data are stored in RAMs. The control circuits, which are implemented in FPGAs, trace the edges according to input vectors. We accelerate it by pipelining. Preliminary experimental results show that a non-pipelined simulation engine is about 2 ∼ 4 times faster, and a pipelined one is about $10n$ times faster than a software cycle-based simulator, where n is the number of inputs.

key words － **Logic simulation, Simulation engine, BDD, FPGA.**

1 はじめに

半導体技術の発展により，一つのチップ上にマイクロ・プロセッサやRAMなど多数の論理回路が搭載されたシステムLSIが作られるようになった．この場合，LSI設計は既存のシステムを統合することとなり，1チップ上に極めて多数の素子が集積される．従って，設計検証は非常に困難で，設計時間の大部分が検証に費やされる．論理検証を効率的に行なうために，形式的検証ツール，論理シミュレータ，シミュレーション・エンジンを組み合わせて使用することが多い [9]. 論理シミュレータは，設計の初期段階から最後まで広く使われる．中でも遅延も取り扱えるイベント・ドリブン・シミュレータが有名である．しかし，処理が複雑なためシミュレーション速度が遅いという欠点がある．そこで，タイミング検証にはスタティック・タイミング・アナライザを，機能検証にはサイクルベース・シミュレータを用いるようになってきた．サイクルベース・シミュレータはLCC(Levelized Compiled Code) 技法を用いたイベント・ドリブン方式の約10倍高速であり，BDD技法を用いたものは，更に約10倍高速である．しかしこのシミュレータの速度でさえ100Hz程度であり，高速化が望まれている [14].

高速化技法としては，専用のハードウェアによるシミュレーションが有望である．シミュレーション専用エンジンとしては，古くはIBMのYSEやNECのHALなどがあり [7, 10]，最近では各CADベンダーからもシミュレーション専用のプロセッサを多数搭載したものが発表されている．これらのエンジンは，イベント・ドリブンやLCC技法を用いたシミュレーションアルゴリズムを実現している．また，FPGA(Field Programmable Gate Array)[4] などの再構成可能デバイスを大量に搭載したエミュレータもある．この場合，実際に回路を論理合成するので，シミュレーションは高速であるが，前処理に相当な時間を必要とする．

本稿の構成は以下の通りである． 第2節で BDD(Binary Decision Diagram) を用いた論理シミュレーションについて概説する．第3節ではBDD技法を用いた論理シミュレータをハードウェア化したハードウェア・シミュレーション・エンジンの構成法を提案し，高速化技法について説明する．第4節で性能評価を行なう．

2 論理シミュレーション

ゲートレベルの論理シミュレーションでは，論理回路図において信号を入力から出力まで順次，ゲート回路図の各信号値を計算していく [1].

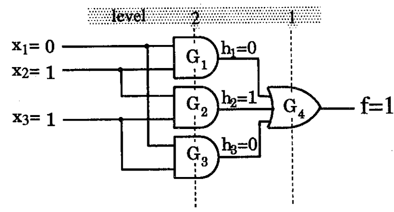

図 2.1: 例 2.1の論理式のゲート回路図

例 2.1 多数決論理関数 $f = x_1x_2 \vee x_2x_3 \vee x_1x_3$ を実現する論理回路を図 *2.1*に示す．入力 $(x_1, x_2, x_3) = (0, 1, 1)$ に対する出力を求めるには，G_1, G_2, G_3 での演算を行い，$h_1 = 0$, $h_2 = 1$, $h_3 = 0$ を求める．次に g_4 での演算を行ない，$f = 1$ を得る． ∎

論理シミュレータとして広く使われているものにイベント・ドリブン・シミュレータがある．このシミュレータを使えば遅延検証と機能検証を同時に行なうことができる．しかし，時間の管理が複雑なため，シミュレーション速度が犠牲になっている．時間の管理という複雑な処理を省き，機能のシミュレーションに限定した高速なシミュレータがあり，サイクルベース・シミュレータと呼ばれる．これには，イベントドリブン・シミュレータのおよそ10倍高速といわれているLCC(Levelized Compiled Code) 技法を用いたものがある．

例 2.2 例 *2.1*の回路を *LCC*技法を用いてシミュレーションする方法を示す．出力 f を求めるには，信号 h_1, h_2, h_3 の値が決まっている必要がある．h_1, h_2, h_3 を求めるためにはそれぞれ (x_1, x_2), (x_2, x_3), (x_1, x_3) の AND をとればよい．どの順番に計算するかは，図 *2.1*に示すように出力からレベル付けした後，出力から一番遠いレベルから順に値を計算する．例 *2.1*の回路から *LCC* 技法を用いたサイクルベース・シミュレータが出力したシミュレーション用のソースプログラム例を図 *2.2*に示す．直接ネイティブコードを出力し，高速化する場合もある． ∎

ゲートレベルのLCC技法を用いたシミュレータでは，全ての信号線を入力ベクトルが変化する度に辿る必要があり，シミュレーション時間は O(ゲート数) に比例する．

LCC技法より更に10倍程度高速なBDDベースのシミュレーションについて概説する．

```
int f (int  x1, x2, x3 ) {
    int    h0, h1, h2 ,h3 ;
    h1 =and( x1, x2 );
    h2 =and( x2, x3 );
    h3 =and( x1, x3 );
    h0 =or(  h1 , h2 ,h3  );
    return  h0;
}
```

図 2.2: 図 2.1の回路のシミュレーション用ソースプログラム

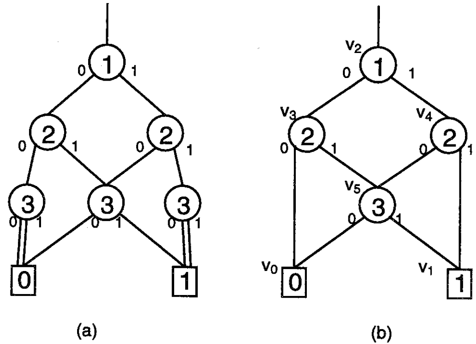

図 2.3: QROBDD と ROBDD

定義 2.1 二分決定グラフ *(BDD: Binary Decision Diagram)* は *2* 種類の節点からなる節点の集合 $\vec{V}$ で表される根付きの有向グラフである．非終端節点 v は，インデックス $index(v) \in \{0,1,\ldots,n\}$ と *2* つの子 $low(v)$, $high(v) \in \vec{V}$ を属性として持つ．終端節点は値 $value(v) \in \{0,1\}$ を属性として持つ．どの非終端節点 v に対しても，もし $low(v)$ が非終端節点であるなら $index(v) < index(low(v))$．同様に，もし $high(v)$ が非終端節点であるなら $index(v) < index(high(v))$ である．論理関数 f と *BDD* は次のように関連づけられる：
終端接点 v に対し：

もし $value(v) = 1$ ならば $f_v = 1$,
もし $value(v) = 0$ ならば $f_v = 0$.

非終端節点 v に対し：もし，$index(v) = i$ ならば，f_v は次のような関数である．即ち，

$$f_v(x_1,\ldots,x_n) = \bar{x}_i \cdot f_{low(v)}(x_1,\ldots,x_{i-1},0,x_{i+1},\ldots,x_n) \vee x_i \cdot f_{high(v)}(x_1,\ldots,x_{i-1},1,x_{i+1},\ldots,x_n)$$

根が関数 f それ自身を表していることに注意．

定義 2.2 *BDD* が次の条件を満たすとき，それを *Quasi Reduced Ordered Binary Decision Diagram(QROBDD)* という．

1. v_1 を根とした部分グラフと v_2 を根とした部分グラフとが等しいような *2* つの相異なる節点 v_1, v_2 を持たない．
2. 根から終端節点までのどの経路も全ての変数を含む．

定義 2.3 *QROBDD* の幅とは，*1* つの変数に関する節点数の最大値をいう．

定義 2.4 *BDD* が次の条件を満たすときそれを *Reduced Ordered Binary Decision Diagram (ROBDD)* という．

1. $low(v) = high(v)$ であるような v を含まない．
2. v_1 を根とした部分グラフと v_2 を根とした部分グラフとが等しいような節点 v_1, v_2 を持たない．

例 2.3 例 *2.1*の回路が実現している多数決論理関数の *QROBDD* と *ROBDD* を図 *2.3(a)(b)* に示す．なお，各節点内の数字は変数の添字を，枝についている数字の *0* は $low(v)$ 側を，*1* は $high(v)$ 側を意味している．∎

BDD 技法を用いたサイクルベース・シミュレータの原理を例を用いて説明する．

例 2.4 図 *2.3(b)* の *ROBDD* を用いて $f(x_1,x_2,x_3) = f(0,1,1)$ を求める．まず根 v_2 のラベルは *1*(x_1 を表す) である．x_1 の値は *0* であるので *0* 側を進み，節点 v_3 に至る．v_3 のラベルは *2*(x_2 を表す) である．x_2 の値は *1* であるので *1* 側に進む．v_5 のラベルは *3*(x_3 を表す) である．x_3 の値は *1* であるので *1* 側に進む．終端節点 [1] (v_1) に到達したので f の値は *1* と決定できる．∎

このように各変数の値に従って根からグラフを辿っていくことで値を求められる．シミュレータは図 2.1の回路に対応した図 2.3(b) の BDD から図 2.4に示すようなソースプログラムを出力する．これをコンパイルし，実行する．x_1, x_2, x_3 に値を加えればシミュレーションすることができる．

n 入力論理関数の 1 回の評価時間は根から終端節点まで辿る枝数に相当するので $O(n)$ となる．通常，入力数 n は内部の信号線数よりはるかに小さいので，BDD に基

```
int f (int  x1, x2, x3 ) {
        if (x1) goto v4 ;
        else goto v3 ;
    v3:
        if (x2) goto v4 ;
        else return(0) ;
    v4:
        if (x2) return(1) ;
        else goto v3 ;
    v5:
        if (x3) return(1) ;
        else return(0) ;
}
```

図 2.4: 図 2.3(b) のソースプログラム

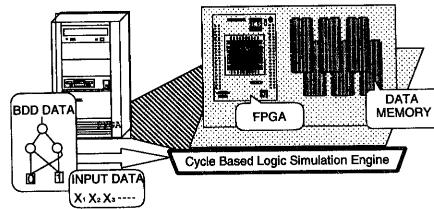

図 3.1: システムの概要図

INDEX	LOW	HIGH

図 3.2: メモリの格納フォーマット

づくシミュレータはゲートレベルの LCC ベースのものに比べて高速である.

3 ハードウェア・シミュレーション・エンジン

論理シミュレータを以下の 3 点の手法で高速化する.

1. ゲートレベルの論理シミュレーションよりも高速な, BDD に基づくシミュレーション [3] を用いる.
2. BDD に基づくシミュレーションを専用のハードウェア (シミュレーション・エンジン) によって, 更に高速化する.
3. ハードウェア上での処理をパイプライン化することによって, 更に高速化する.

3.1 システムの概要

提案するサイクルベース・シミュレーション・エンジンは, ソフトウェアで実現する場合と同様に, BDD を辿ることでシミュレーションを行なう. エンジンは, 主にメモリと制御回路で構成する. メモリにはシミュレーション対象の回路を表現する BDD データを格納する. 制御回路は FPGA で実現し, 入力値に従って BDD を一変数ずつ辿る機能を実現する. システムの概要図を図 3.1に示す.

シミュレーションに必要な BDD のデータや入力ベクトルは, ホスト PC のバス経由で渡す. BDD の生成や送信, 入力ベクトルの送信, シミュレーション結果の保存, 表示はホスト PC で行なう. エンジンはシミュレーションに必要なデータを受信次第, シミュレーションを始める. シミュレーションはエンジンのみで実行し, ホスト PC は全く関与しない. 専用ハードウェアでシミュレーションするので, 命令のフェッチ・デコードを行なわなくて済む分, ソフトウェアでのシミュレーションよりも高速になる.

3.2 シミュレーション・エンジンの動作

ここではシミュレーション・エンジンの具体的な仕様・動作を説明する.

エンジンの制御回路の役割は次の 2 点である.

1. ホスト PC, エンジン間の通信の調停.
2. シミュレーションの実行.

エンジン内のメモリに格納する BDD の格納フォーマットを, 図 3.2に示す. BDD の節点は, INDEX, 0 側, 1 側という 3 つのパラメータを持つ. INDEX は非終端節点については入力変数のラベルを示す. 0 側と 1 側には次に参照する節点の番地を格納する. INDEX が示す入力値が 0 ならば LOW, 1 ならば HIGH が示すアドレスを参照する. 0 番地は終端節点[0]を, 1 番地は終端節点[1]を示すことにする.

例 3.1 例 *2.1*に示した論理関数の *BDD*(図 *2.3(b)*) は, 図 *3.3*に示したようにメモリに格納する. $\vec{a} = (0, 1, 1)$ を入力ベクトルとする. $f(\vec{a})$ を求める動作を図 *3.3*に示す.

	ADDRESS	INDEX	LOW	HIGH
v_0	0	*	*	*
v_1	1	*	*	*
v_2	2	1	3	4
v_3	3	2	0	5
v_4	4	2	5	1
v_5	5	3	0	1

START

* :DON'T CARE

図 3.3: メモリ格納例とシミュレーション例

1. まず最上部の節点 v_2 の $INDEX(v_2)$ を調べる.
2. $INDEX(v_2) = 1$ なので, x_1 の値を調べる.
3. $x_1 = 0$ なので v_2 の *0* 側を参照すると *3* 番地を得る. これは v_3 を表す.
4. $INDEX(v_3) = 2$ なので, x_2 の値を調べる.
5. $x_2 = 1$ なので v_3 の *1* 側を参照すると *5* 番地を得る. これは v_5 を表す.
6. $INDEX(v_5) = 3$ なので, x_3 の値を調べる.
7. $x_3 = 1$ なので v_5 の *1* 側を参照すると *1* 番地が得られる. これは終端節点 v_1 を表し, $f(0,1,1) = 1$ を得る.

3.3 エンジンの高速化

エンジンでは大量の入力ベクトルに対してシミュレーションを行う必要がある. パイプライン処理を組み込むことで, スループットの向上を図る. ここでは. 入力変数と同じ個数 (n) のとユニットを個別にそれぞれ用意することによりパイプライン処理を行なう.

例 3.2 非パイプライン化エンジンで, 論理関数 $f = x_1x_2 \vee x_2x_3 \vee x_1x_3$ に *4* つの入力ベクトルを連続でシミュレーションする. 入力ベクトルを

$$\vec{a} = (a_1, a_2, a_3) = (0, 1, 0)$$
$$\vec{b} = (b_1, b_2, b_3) = (0, 0, 1)$$
$$\vec{c} = (c_1, c_2, c_3) = (1, 0, 1)$$
$$\vec{d} = (d_1, d_2, d_3) = (1, 1, 0)$$

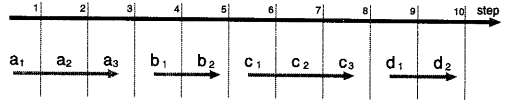

図 3.4: 非パイプライン化エンジンでの連続シミュレーション

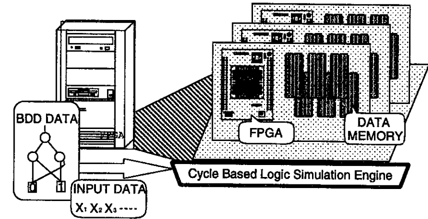

図 3.5: パイプライン化エンジンの概要図

とする. *BDD* の枝を *1* 変数分辿る処理を *1* ステップと呼ぶ. 図 *3.4*に示すように *10* ステップで *4* つの入力ベクトルに対するシミュレーションが終了する. また, この論理関数の *BDD* は最大 *3* 段, 最小 *2* 段なので非パイプライン化エンジンでは p 個の入力ベクトルを連続に処理する場合 $2p \sim 3p$ ステップ必要である.

3.3.1 入力変数毎のパイプライン化

入力変数毎にメモリを分割すれば, 同時に n 変数分のメモリを参照できる. パイプライン化エンジンの概要を図 3.5に示す. パイプライン化による高速化を例 3.2の論理関数と入力ベクトルを用いて説明する. 例 3.2の論理関数は 3 入力なので, 最大 3 ステップで評価が終了する. 一つの入力ベクトルに対して, 入力変数はそれぞれ 1 回だけ参照され, ラベルが 1 の節点, 2 の節点, 3 の節点が順番に参照される. 非パイプライン化エンジンでは 1 つのベクトルのシミュレーション終了まで次のベクトルのシミュレーションを開始できない. よって, シミュレーションに必要なステップ数は, 個々のベクトルが辿る節点数の合計に等しい.

提案するパイプライン処理では, ラベル $1, 2, \cdots, n$ に対して共有していたハードウェアを, 入力変数それぞれに専用のものを用意する. 入力ベクトルがどのような順番であっても, 先に入力されたベクトルから順にシミュレーションする必要がある. このためには, どの経路を

辿っても経由する節点数が一定になるようにするため QROBDD を使用する．

例 3.3 図 *3.6(a)* に，多数決論理関数 $f(\vec{X}) = x_1x_2 \vee x_2x_3 \vee x_1x_3$ の *QROBDD* を用いて入力変数ごとにパイプラインする例を示す．パイプライン化に伴い，格納する *BDD* データも入力変数 *(INDEX)* ごとに分割する．*(b)* には *QROBDD* のデータを *INDEX* 毎に分割して各々のメモリに格納する手順を示す．*(1)* データから *INDEX* 毎にアドレスを取り出す．*memory1* は x_1, *memory2* は x_2, *memory3* は x_3 にそれぞれ対応している．*(2)* メモリが無駄なく使えるようにアドレスを *2* 番地からに付け直す．このとき，次の節点のアドレスも変化しているのでポインタも変える必要がある．*(3)* メモリは入力変数毎に専用になるので，メモリ格納フォーマットの要素から *INDEX* の欄を取り除く． ∎

例 3.4 入力変数毎にパイプラインを組み込んだエンジンで，論理関数 $f(\vec{X}) = x_1x_2 \vee x_2x_3 \vee x_1x_3$ に *4* つの入力ベクトルを連続でシミュレーションする．入力ベクトルを

$$\vec{a} = (a_1, a_2, a_3)$$
$$\vec{b} = (b_1, b_2, b_3)$$
$$\vec{d} = (c_1, c_2, c_3)$$
$$\vec{d} = (d_1, d_2, d_3)$$

とする．パイプライン処理を行なう様子を図 *3.7* に示す．横軸をステップとする．

1. ユニット *1* では $x_1 = a_1$ を処理する．

2. ユニット *1* では $x_1 = b_1$, ユニット *2* では $x_2 = a_2$ を加える．複数のユニットが並列動作していることに注意．

3. ユニット *1* では $x_1 = c_1$, ユニット *2* では $x_2 = b_2$, ユニット *3* では $x_3 = a_3$ を処理する．ここで，$\vec{a}$ のシミュレーションが終了する．

4. ユニット *1* では $x_1 = d_1$, ユニット *2* では $x_2 = c_2$, ユニット *3* では $x_3 = b_3$ を処理する．ここで，$\vec{b}$ のシミュレーションが終了する．$\vec{a}$ から $\vec{b}$ の結果が出るまでわずか *1* ステップしかかからない．

5. ユニット *2* では $x_2 = d_2$, ユニット *3* では $x_3 = c_3$ を処理する．ここで，$\vec{c}$ のシミュレーションが終了する．最初のベクトルのシミュレーション結果が出てから *1* ステップ毎に結果が出る．

6. ユニット *3* では $x_3 = d_3$ を処理する．ここで，全ての入力ベクトルのシミュレーションが終了する．

以上 *6* ステップで全ベクトルのシミュレーションを終了した． ∎

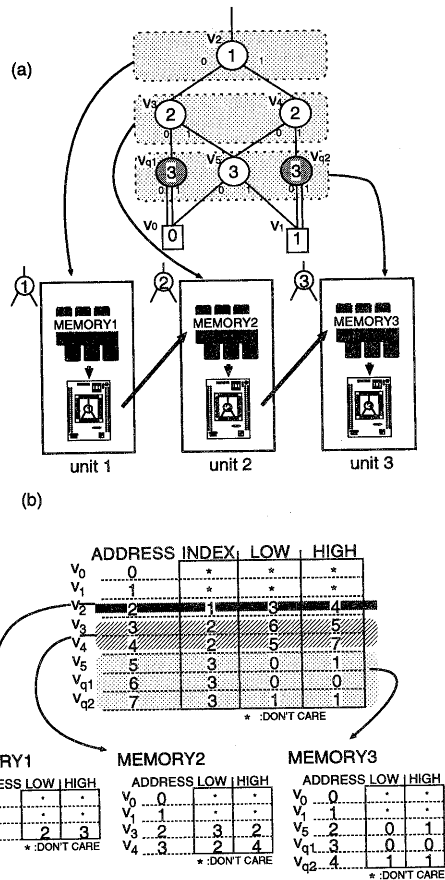

図 3.6: パイプライン処理のハードウェアとメモリ配分

非パイプライン化エンジンで p 個の入力ベクトルをシミュレーションするには，最大 np ステップが必要となる．一方，パイプライン化エンジンでは，どのような入力ベクトルに対しても一ベクトルに対し n ステップ必要になるが，連続的に入力ベクトルを加えれば，全体と

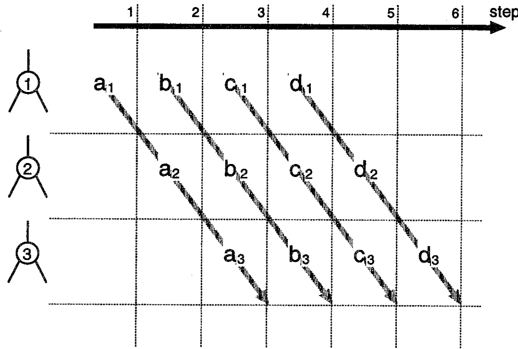

図 3.7: パイプライン処理の例

して $n+p-1$ ステップで十分である. 1 ステップ毎に 1 つのシミュレーション出力が得られ, スループットが大幅に向上する. 但し, ユニット数は n 個必要である.

4 エンジンの評価

ベンチマーク回路について ROBDD と QROBDD を作成し, その節点数を比較したものを表 4.1に示す. 例えば, accpla は 50 入力, 69 出力の関数で, ROBDD の非終端節点数が 1587 個, QROBDD の非終端節点数が 5822 である. QROBDD の幅で最大のものは 234 である. QROBDD の節点数は ROBDD の比べ, 数倍から 10 倍程度である. また, 入力変数毎に分割した場合に必要になる最大メモリ量は, QROBDD の幅を w とすると, $2\lceil log_2 w\rceil[bit] \times w[bit]$ である. パイプラインを行なう場合, この中で幅が最大の C3540 でも 1 ユニット当たり $26[bit] \times 8192[bit]$(208Kbyte) のメモリを用意しておけば十分である.

非パイプライン化エンジンを, VHDL で記述し XILINX 社の XC4010E-PG191-1 を用いて実現した場合, 最大動作速度は 22.7MHz を得た. パイプライン化エンジンの最大動作速度は 52.7MHz であった. ソフトウェアで実現されたサイクルベース・シミュレータ Prestissimo[3] と, 提案したエンジンの比較を表 4.2に示す. 例えば, C3540 は 50 入力 22 出力の関数であり, 1 ベクトルのシミュレーションに要する時間は Prestissimo では 29.0[μsec/vec] である. 本稿で提案した非パイプライン化エンジンでは, これの 13.1 倍の速度, パイプライン化エンジンで $30.5n$(n は入力変数の数) 倍の速度を得た. ここでパイプライン化エンジンの速度はスループットであることに

表 4.1: ROBDD と QROBDD の比較

Function	In	Out	ROBDD	QROBDD	Width
accpla	50	69	1587	5822	234
apex1	45	45	1275	4274	251
apex5	117	88	1077	10310	209
signet	39	8	1440	3894	224
xparc	41	73	1875	4749	184
C432	36	7	1075	1996	101
C499	41	32	27876	28146	2176
C880	60	26	4166	11467	466
C1908	33	25	7456	9705	620
C2670	233	140	2847	23887	411
C3540	50	22	34710	119358	5420
C5315	178	123	2564	21973	258
C7552	207	108	2945	19963	193

表 4.2: ソフトウェアシミュレーションとの比較

Function	In	Out	Prestissimo [μsec/ vector]	simulation-engine non-Pipelined	Pipelined
C432	36	7	3.2	2.0	$4.7n$
C499	41	32	15.4	8.5	$19.8n$
C880	60	26	19.2	7.3	$16.9n$
C1908	33	25	7.6	5.2	$12.1n$
C2670	233	140	55.6	5.4	$12.6n$
C3540	50	22	29.0	13.1	$30.5n$
C5315	178	123	112.8	14.4	$33.4n$
C7552	207	108	46.5	5.1	$11.8n$

注意. 提案した非パイプライン化エンジンは Prestissimo に比べ 2.0 ~ 14.4 倍, パイプライン化エンジンは $4.7n$ ~ $33.4n$ 倍 (n は入力変数の数) の高速化が達成された.

5 あとがき

BDD を用いたサイクルベース・シミュレータをハードウェアで実現する方法を示した. ソフトウェア・シミュレータより, 高速化技法として変数毎のパイプラインを組み込むことにより $10n$ 倍 (n は入力の個数) の高速化が可能なことを示した.

参考文献

[1] M. Abramovivi, 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] P. C.McGeer, K. L. McMillan, A. Saldanha, A. L. Sangiovanni-Vincentelli, and P. Scaglia, "Fast dis-

crete function evaluation using decision diagrams," *ICCAD'95*, pp.402-407, Nov. 1995.

[4] S. D.Brown, R.J. Francis, J. Rose, and Z.G. Vranesic, "*Field Programmable Gate Arrays*," Kluwer Academic Publishers, Boston, 1992.

[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] 石垣壮, 米田友洋, "ハードウェアによるZBDD処理の実現に関する研究," 信学技報 ICD98-3, CPSY98-3, FTS98-3 (1998-04).

[7] 小池誠彦 "CADマシン," オーム社, 1989.

[8] S. Minato, "Graph-based representation of discrete functions," in [12].

[9] 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.

[10] T. Nakata, and N. Koike, "Design automation machine," *Design Methodologies*, S. Goto(Editor), North-Holland, pp. 465-499, 1986.

[11] T. Sasao (ed.), *Logic Synthesis and Optimization*, Kluwer Academic Publishers, 1993.

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

[13] 笹尾勤, "論理設計: スイッチング回路理論 [第2版]," 近代科学社, 1998.

[14] "LSIの検証が変わる," 日経エレクトロニクス, pp.990125, No.669, Sep. 1996.