社団法人 情報処理学会 研究報告
IPSJ SIG Technical Reports

2006－ＳＬＤＭ－125（８）
2006／5／12

値独立なシミュレータカーネルを用いた3値論理シミュレータの実装

和田 崇臣† 日比野 靖†

† 北陸先端科学技術大学院大学情報科学研究科
〒923-1293 石川県能美市旭台 1-1
E-mail: †{t-wada,hibino}@jasit.ac.jp

あらまし 多値論理は少ない桁数での表現が可能であり，配線数の削減等による LSI の集積密度向上が期待できる．また，暗号処理において 3 値論理を用いた効率的処理が可能となる．我々は 3 値論理回路を設計する上で必要となる 3 値論理シミュレータの実装を行った．実装をする上で拡張性を向上させるために値独立なカーネルを用いた．3 値の入出力時の表現と内部表現との変換を設定することにより，任意の 3 値表現を用いた論理シミュレーションが可能である．また，使用する論理素子の定義をカーネル外部に追加することによる柔軟な拡張が可能である．
キーワード 3 値論理，論理シミュレーション

An Implementation of a Ternary-valued Logic Simulator using a Value-independent Simulator Kernel

Takatomi WADA† and Yasushi HIBINO†

† School of Information Science, Japan Advanced Institute of Science and Technology(JAIST)
1-1, Asahidai, Nomishi, Ishikawa 923-1293 Japan
E-mail: †{t-wada,hibino}@jasit.ac.jp

Abstract In multi-valued logic, the expression by a little number of digits can be possible. So the integration of LSI is improved by the reduction in the number of wiring etc. Moreover, efficient processing that uses ternary-valued logic in the encryption processing is possible. We implement ternary-valued logic simulator needed in designing ternary-valued logical circuit. A value-independent kernel was used to improve the extendibility in implementation. Logic that uses an arbitrary ternary-valued expression can be simulated by setting the conversion of external expression into internal expression. And a flexible extension of the simulator is possible by adding the definition of the logic function used outside of the kernel.
Key words ternary-valued logic, logic simulation

1. はじめに

多値論理は 2 値論理と比較して同じ数を少ない桁数で表現することが可能である．そのため，ハードウェア実装において配線量や入出力ピン削減による集積密度の向上が望める．また，数表現が多値符号化に適した演算方式が見いだされている [1]．

情報セキュリティにおける主要な技術である暗号の分野においては，楕円曲線暗号や XTR の暗号処理効率化に標数 3 の体を用いることが提案されている [2] [3]．標数 3 の体は {0,1,2} 内での演算が主である．従来の計算機で実装しようとすると，3 値論理の 1 桁のために 2 ビットが必要となり無駄が多くなってしまう．3 値論理ゲートを使用することにより，この無駄がなくなる．また，{-1,0,1} を用いる符号付きバイナリ表現を用いた演算の効率化も提案されている．

このように特定なシステムの専用ハードウェア化において多値論理回路の使用が有効である．以前より多値論理ゲートに関する研究が行われてきた．[4] では現在の計算機で一般的に用いられている CMOS デバイスを用いた 3 値論理ゲートの構成法が提案されている．この手法ではしきい値が異なる複数の MOSFET 及び電圧の異なる複数の電源を用いることで 2 値論理の CMOS デバイスと同様に製造でき，現在の技術でも実装が可能である．

回路設計を行っていく上で、回路の動作を検証するためのシミュレーションは必要不可欠である．しかし，3 値論理回路を直接取り扱うシミュレータに関する研究報告は行われていない．

今回，我々は様々な値表現への対応を可能とする値独立なシ

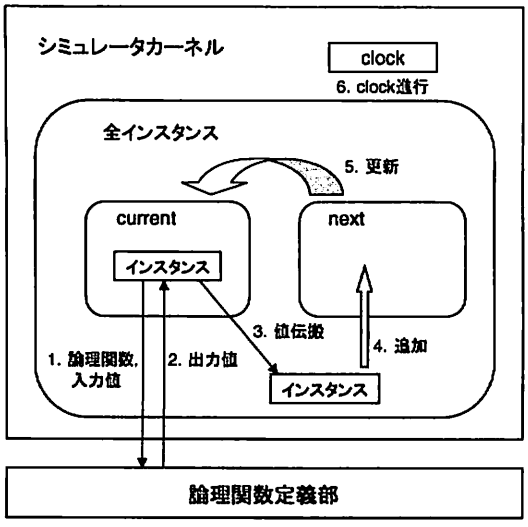

図1 シミュレータカーネルの概念

フラグ	出力ピン接続先	入力ピン	出力ピン	論理関数

図2 インスタンスの構造

ミュレータカーネルを作成した．そして，このカーネルを利用し3値を取り扱うことが可能な論理シミュレータの実装を行った．実装に使用する言語としてはリスト処理の容易性の点からLISPを選んだ．

2. シミュレータカーネル

2.1 カーネルと論理関数シミュレーションの分離

シミュレータを実装するにあたり，次に挙げる点を考慮することとした．

- 任意の値表現の利用
- 使用する論理関数の追加，変更
- 多値論理シミュレーションへの拡張

以上の点について柔軟に対応するため，シミュレータカーネルは評価対象回路中の値とは独立して動作するものを用いることとした．シミュレータが扱う値表現の設定と論理関数についてはカーネル外部に記述することとした．

作成したシミュレータカーネルの概念図1に示す．カーネルは評価対象回路に含まれる各ゲートごとにインスタンスを作成し，ゲートに関する情報を保持する．インスタンスには論理関数の種類，入出力ピン，出力ピンの接続先，フラグが含まれている（図2）．フラグはゲートの状態を表すために用いる．また，カーネルはcurrentとnextという2つのインスタンスリスト及びシミュレーションの時間経過をカウントするclockを有する．インスタンスリストcurrentには現在のclockで動作中のインスタンスが含まれている．インスタンスリストnextには次のclockで動作するインスタンスが含まれる．この2つのリストを利用してカーネルは回路中の信号伝搬をシミュレートする．

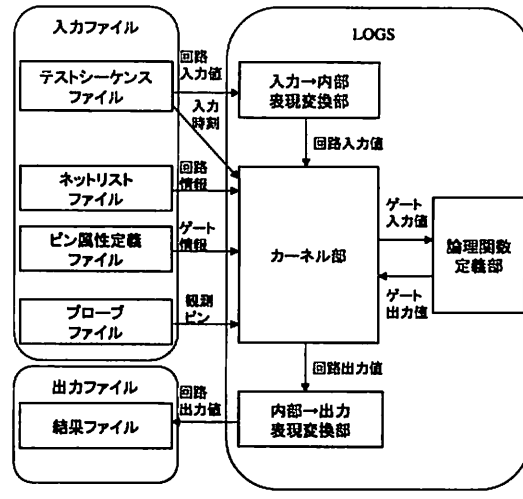

図3 シミュレータの概要

2.2 カーネルの動作

カーネルの動作は以下のように行われる．

（1） 回路の入力ピンに接続されているゲートのインスタンスをcurrentとする．

（2） currentに含まれる全インスタンスについて出力値を求める．

（3） 出力値が変化したゲートの出力ピン接続先に値を渡す．

（3a）接続先がゲートならば接続先インスタンスをnextに追加する．

（3b）接続先が回路の出力ピンの場合は，シミュレータの出力とする．

（4） 回路の入力値が変化した場合は入力ピンの接続先インスタンスをnextに追加する．

（5） nextをcurrentへと変更しclockを1つ進め（2）に戻る．

（6） 終了クロックになった時点でシミュレータを終了する．

2.3 論理関数シミュレーション

currentに含まれるインスタンスの論理演算をカーネル自身は行わない．ゲートの入力値を別に用意した論理関数定義部に渡し，その演算結果を受け取ってゲートの出力値とする．カーネルは値を評価することなく，クロックに応じて信号を伝搬させる役割のみを有する．そのため，このカーネルは様々な値表現に対応することが可能なのである．

3. 論理シミュレータLOGS

作成したカーネルを基に論理シミュレータLOGSを構築した．LOGSの構成は図3の通りである．LOGSはカーネルの他，入（出）力時の表現と内部表現との表現変換部，論理関数定義部から構成されている．

シミュレーション実行時にはネットリストファイル，ピン属性定義ファイル，テストシーケンスファイルを入力ファイルとして用意する．シミュレーションの実行結果は画面に表示する

とともに，結果ファイルへと出力する．また，オプションとしてプローブファイルを入力することによって，評価対象回路の任意の中間値を観測することも可能である．

LOGS の動作は次のような流れとなる．

（1） 入力ファイルをカーネルに読み込みインスタンスを作成する．入力信号値は表現変換部を通して読み込む．

（2） current のインスタンス毎に論理関数定義部を参照しつつシミュレーションを行う．

（3） 出力値を伝搬させ，next を作成する．

（4） カーネルの出力結果は表現変換部を通して結果ファイルとして出力される．

カーネル以外の構成部と各入出力ファイルについて，次に説明していく．

3.1 論理関数定義部

論理関数定義部では，使用者が予め用意するゲートの論理関数を定義する．多値論理では考えうる論理関数が膨大であるため，全ての論理関数を用意することは現実的ではない．例えば 2 入力ゲートを考えた場合，2 値では $2^{2^2} = 16$ 種類のゲートが考えられる一方，3 値では $3^{3^2} = 19683$ 種類が考えられる．使用者はシミュレーションで必要となるゲートを定義すれば良い．

3.2 表現変換部

表現変換部では入（出）力時の表現と内部表現との値表現の変換を行う．この変換部の設定により，ゲート関数の定義を変更することなく {-1, 0, 1} や {0, 1, 2}, {1, 2, 3} など使用者が用いたい表現を扱うことが可能である．内部表現については論理関数定義部で用いている値と統一しておく必要がある．

3.3 入出力ファイル

・ネットリストファイル

評価対象の回路を記述するもので，回路の入力ピンと出力ピン，使用するゲート，ネットリストからなる．ネットリストは信号ソースのピンからデスティネーションのピンへのリストで表す．各ゲートのピン名についてはピン属性定義ファイルで記述した名前とする．

・ピン属性定義ファイル

ネットリストファイルで使用するゲートの入出力ピン名及び論理関数を定義する．論理関数はシミュレータの論理関数定義部で用意されたものから選ぶ．

・テストシーケンスファイル

評価対象回路の入力ピンに加えるテスト信号の値と信号を与える時刻（clock）を記述する．また，観測する出力ピンを指定する．

・プローブファイル

テストシーケンスファイルで観測のために指定できるピンはネットリストファイルで回路の出力ピンとして記述したピンのみである．しかし，回路の検証を行う上で中間値の観測が必要な場合がある．そこで，回路中の任意の場所を観測する手段としてのプローブ端子を記述するものとしてプローブファイルをオプションとして用いることができる．プローブファイルは任意のゲートの入力ピンまたは出力ピンを記述する．

・結果ファイル

```
(defun simulate (test-seq result-out probe-in
                 interactive &aux inst)
  (setq sys-clock 0)
  (initialize-system)
  (set-constant-and-awake)
  (loop
   (when (> sys-clock sys-clock-end)
         (return))
   (get-input-signals test-seq)
   (dolist (a-inst current-inst-list)
    (setq inst (symbol-value a-inst))
    (cond ((eq (first inst) 'logic)
           (sim-logic inst))
          (t (error-report) )))
   (dolist (a-inst current-inst-list)
    (setq inst (symbol-value a-inst))
    (unless (eq (second inst) 2)
      (propagate-signal (fifth inst) ) ))
   (output-result result-out interactive)
   (incf sys-clock)
   (swap-current-to-next-inst-list)
   ))

(defun sim-logic (inst)
  (funcall (sixth inst)
           (fourth inst) (fifth inst)) )
```

図 4 シミュレータカーネル

シミュレータの実行結果が出力される．結果ファイルにはテストシーケンスファイルで指定された入出力ピンの値の変化とその時のクロックが記録される．プローブファイルが入力されていた場合は，プローブ端子の値の変化も含めて出力される．

4. 3 値論理シミュレータの実装

今回行った 3 値論理シミュレータの実装について，2 値論理での実装と併せて説明する．まず，シミュレータカーネルの主要部分を図 4 に示す．インスタンスは

`('logic flag inst-cell inpins outpins fn logic-fn)`

の形で実装されており，カーネル中の関数 sim-logic を用いて論理関数定義部を参照する．LOGS は表現変換部と論理関数定義部の記述を変更するだけで，2 値論理と 3 値論理のいずれの論理シミュレーションも行うことが可能である．

まず，論理関数の定義例として 2 値論理での or と 3 値論理での or をそれぞれ図 5，図 6 に示す．記述は論理関数の真理値表に基づき行えば良い．3 値論理の or については，入力値のうち大きい値を出力する max 関数として定義している．ここでは，2 値論理では内部表現として {-1,1} を用いており，不定値を 0 として扱っている．3 値論理では内部表現として {0,1,2} を用いており，不定値は-1 である．この例では記述に工夫をし，1 つの定義で任意の入力数のゲートを扱うことができる．ゲートの入力ピン数はピン属性定義ファイルで定義した数となる．

次に入力表現から内部表現への変換設定例を 2 値論理と 3 値論理についてそれぞれ図 7，図 8 に示す．この例では入力時の不定値表現 X を関数定義部で用いている不定値の表現に変換

```
(defun $or (inpins outpins &aux inpin flag
              old-out-value out-value)
  (setq flag nil)
  (setq out-value 0)
  (dolist (a-inpin inpins)
    (setq inpin (symbol-value a-inpin))
    (when (eq (first inpin) 1)
          (setq out-value 1)(return) )
    (when (eq (first inpin) 0)
          (setq flag 'undef) ) )
  (when (and (/= out-value 1)(eq flag 'undef))
        (setq out-value 0))
  (when (and (/= out-value 1)(eq flag nil))
        (setq out-value -1))
  (setq old-out-value
        (symbol-value (car outpins)))
  (unless (eq old-out-value out-value)
    (setf (first (symbol-value (car outpins)))
          out-value) ))
```

図 5　2 値論理 or

```
(defun $3or (inpins outpins &aux inpin flag
              old-out-value out-value)
  (setq flag nil)
  (setq out-value -1)
  (dolist (a-inpin inpins)
    (setq inpin (symbol-value a-inpin))
    (when (eq (first inpin) -1)
          (setq flag 'undef)
          (setq out-value -1)
    (return) )
    (when (eq (first inpin) 2)
          (setq flag 'two) )
    (when (eq (first inpin) 1)
          (setq out-value 1) ) )
  (when (eq flag 'two)
        (setq out-value 2) )
  (when (and (eq flag nil)
             (/= out-value 1) )
        (setq out-value 0) )
  (setq old-out-value
        (first (symbol-value (car outpins))))
  (unless (eq old-out-value out-value)
    (setf (first (symbol-value (car outpins)))
          out-value) ))
```

図 6　3 値論理 or

```
(defun conv-internal (a-ext)
  (cond ((eq a-ext 1) 1)
        ((eq a-ext 0) -1)
        ((eq a-ext 'x) 0)
        (t (error-report))))
```

図 7　2 値での入力表現から内部表現への変換

```
(defun conv-internal (a-ext)
  (cond ((eq a-ext 0) 0)
        ((eq a-ext 1) 1)
        ((eq a-ext 2) 2)
        ((eq a-ext 'x) -1)
        (t (error-report))))
```

図 8　3 値での入力表現から内部表現への変換

```
(defun $add (inpins outpins &aux inpin
             flag old-out-value out-value)
  (setq flag nil)
  (setq out-value 0)
  (dolist (a-inpin inpins)
    (setq inpin (symbol-value a-inpin))
    (when (eq (first inpin) -1)
          (setq flag 'undef)
          (setq out-value -1)
          (return) )
    (setq out-value (+ out-value (first inpin)))
    (when (> out-value 2)
          (setq out-value (- out-value 3)) ) )
  (when (eq flag 'undef)
        (setq out-value -1) )
  (setq old-out-value
        (first (symbol-value (car outpins))))
  (unless (eq old-out-value out-value)
    (setf (first (symbol-value (car outpins)))
          out-value) ) )
```

図 9　3 値論理 add

```
(defun $carry (inpins outpins &aux inpin flag
               old-out-value out-value state)
  (setq flag nil)
  (setq out-value 0)
  (setq state 0)
  (dolist (a-inpin inpins)
    (setq inpin (symbol-value a-inpin))
    (when (eq (first inpin) -1)
          (setq flag 'undef)
          (setq out-value -1)
          (return) )
    (setq state (+ state (first inpin)))
    (when (> state 2)
          (setq out-value 1) ) )
  (when (eq flag 'undef)
        (setq out-value -1) )
  (setq old-out-value
        (first (symbol-value (car outpins))))
  (unless (eq old-out-value out-value)
    (setf (first (symbol-value (car outpins)))
          out-value) ))
```

図 10　3 値論理 carry

している．内部表現から出力表現への変換設定も同様に行えば良い．

3 値論理シミュレータの動作確認の 1 つとして，4 桁加算器のネットリストを作成し，そのシミュレーションを行うこととした．4 桁加算器を構成するにあたり，論理関数定義部に 1 桁加算関数 add（図 9）と桁上げ演算関数 carry（図 10）を追加定義した．図 11 は add と carry を基に単純に 4 桁加算器を構成したとき回路図である．この回路を記述したネットリストファイルが図 12 である．ピン属性定義ファイルは図 13 の通りである．

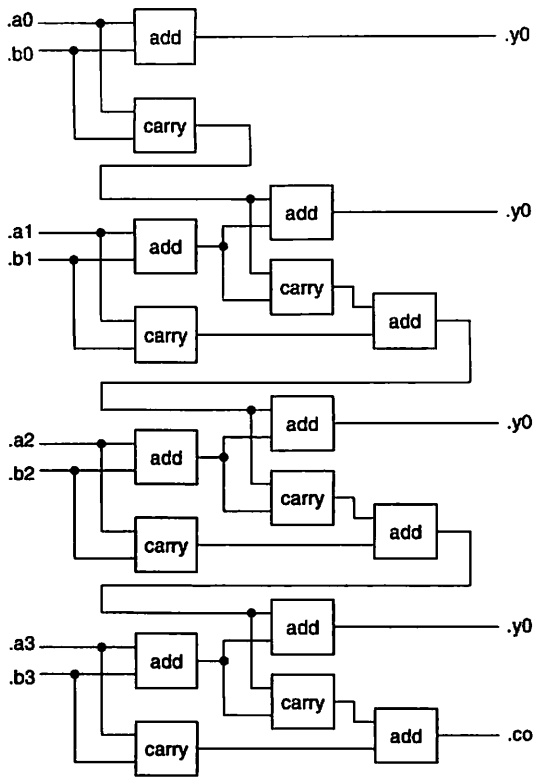

図 11　3 値論理 4 桁加算器の回路図

入力ピン.a0～.a3 及び.b0～.b3 に対して図 14 のテストシーケンスファイルで記述した信号を加え，出力ピン.y0～.y3 及び.co を観測するシミュレーションを行った．シミュレーションの結果を図 15 に示す．動作確認の結果，シミュレータが正確に処理していることを確認した．

5. ま と め

今回我々は任意の値表現を取り扱うことが可能である値独立なシミュレータカーネルを作成した．このカーネルを用いて，3 値論理関数の定義と値表現の設定を行い，3 値論理シミュレータを実装した．これにより，3 値論理ゲートを用いたシミュレーションを容易に行うことが可能となった．今後，4 値以上の多値論理に基づく回路のシミュレーションが必要な際にも，このカーネルを利用することにより，論理関数の定義と値表現の設定のみで短期間にシミュレータを構築することが可能であると言える．

文　献

[1] 樋口龍雄，亀山充隆，"多値情報処理"，昭晃堂，東京，1989.

[2] T. Kerins, W. P. Marnane, E. M. Popovic and P. S. L. M. Barreto, "Efficient Hardware for the Tate Pairing Calculation in Characteristic Three", Cryptographic Hardware and Embedded Systems–CHES 2005, LNCS 3659, Springer, pp.412-426, 2005.

[3] 日比野靖，白勢政明，"標数 3 の体での XTR"，信学技報，Vol. 105, No. 51, ISEC2005-3, 2005.

[4] 白勢政明，日比野靖，"CMOS トランスファーゲートによる三値論理回路とその構成法"，多値技報 Vol. MVL-05, No. 1, 2005.


```
(adder4
    (i .a0)(i .a1)(i .a2)(i .a3)
    (i .b0)(i .b1)(i .b2)(i .b3)
    (o .y0)(o .y1)(o .y2)(o .y3)(o .co)
    (c add01 ad01)
    (c add01 ad11)
    (c add01 ad12)
    (c add01 ad13)
    (c add01 ad21)
    (c add01 ad22)
    (c add01 ad23)
    (c add01 ad31)
    (c add01 ad32)
    (c add01 ad33)
    (c car01 ca01)
    (c car01 ca11)
    (c car01 ca12)
    (c car01 ca21)
    (c car01 ca22)
    (c car01 ca31)
    (c car01 ca32)
    (n .a0 ad01.a ca01.a)
    (n .b0 ad01.b ca01.b)
    (n .a1 ad11.a ca11.a)
    (n .b1 ad11.b ca11.b)
    (n ca01.y ad12.a ca12.a)
    (n ad11.y ad12.b ca12.b)
    (n ca11.y ad13.a)
    (n ca12.y ad13.b)
    (n .a2 ad21.a ca21.a)
    (n .b2 ad21.b ca21.b)
    (n ad13.y ad22.a ca22.a)
    (n ad21.y ad22.b ca22.b)
    (n ca21.y ad23.a)
    (n ca22.y ad23.b)
    (n .a3 ad31.a ca31.a)
    (n .b3 ad31.b ca31.b)
    (n ad23.y ad32.a ca32.a)
    (n ad31.y ad32.b ca32.b)
    (n ca31.y ad33.a)
    (n ca32.y ad33.b)
    (n  ad01.y .y0)
    (n  ad12.y .y1)
    (n  ad22.y .y2)
    (n  ad32.y .y3)
    (n  ad33.y .co) )
```

図 12　3 値 4 桁加算器のネットリスト

```
pinprop
   (add01 $add ((.a) (.b)) ((.y)) )
   (car01 $carry ((.a) (.b)) ((.y)) ) )
```

図 13　ピン属性定義

付　録

1. LISP による実装上の考慮

シミュレータカーネル中の current や next のリストを作る

```
(e 1000)
(i .a0 .a1 .a2 .a3 .b0 .b1 .b2 .b3)
(o .y0 .y1 .y2 .y3 .co)
(0000  0 0 0 0 0 0 0 0)
(0100  1 1 1 1 0 0 0 0)
(0200  1 1 1 1 1 1 1 1)
(0300  1 1 1 1 1 1 1 2)
(0400  1 1 1 1 2 2 2 2)
(0500  2 2 2 2 2 2 2 2)
(0600  0 0 0 0 2 2 2 2)
```

図 14　テストシーケンス

```
      ........ .....
      aaaabbbb yyyyc
clock 01230123 0123o

00000 00000000 XXXXX
00005 00000000 0XXXX
00022 00000000 00XXX
00050 00000000 000XX
00078 00000000 0000X
00089 00000000 00000
00100 11110000 00000
00109 11110000 10000
00150 11110000 11110
00200 11111111 11110
00205 11111111 21110
00222 11111111 22220
00300 11111112 22220
00322 11111112 22200
00334 11111112 22201
00400 11112222 22201
00405 11112222 02201
00422 11112222 00001
00450 11112222 01001
00475 11112222 01111
00500 22222222 01111
00509 22222222 11111
00546 22222222 12221
00600 00002222 12221
00605 00002222 22221
00616 00002222 22220
00622 00002222 22020
00633 00002222 22220
01000 00002222 22220
```

図 15　3 値 4 桁加算器のシミュレーション結果

際，LISP 特有の cons を用いて作ると，セルの消費が毎回行われ，シミュレーションの途中でガーベジ・コレクション (屑セルの回収) が発生してしまう．これを防ぐため，インスタンス対応に予めセルを確保し，それをインスタンス自体に登録している．

インスタンス中の inst-cell は，car 部をインスタンス自身へのポインタ，cdr 部をリストの次の要素へのポインタとしている．current あるいは next のリストをたどることにより，インスタンスを次々と取り出すことが可能である．また、current や next のリストを解放した後も，リストを構成するセルは屑になることなく各インスタンスが保有した状態を保ち，再びリストを作成する際に再利用される．