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

2006－SLDM－127（4）
2006／11／28

算術演算回路の形式的検証手法とその評価

渡邉 裕樹† 本間 尚文† 青木 孝文† 樋口 龍雄††

† 東北大学 大学院情報科学研究科 〒980-8579 仙台市青葉区荒巻字青葉 6-6-05
†† 東北工業大学 工学部 電子工学科 〒982-8577 仙台市太白区八木山香澄町 35 番 1 号
E-mail: †{watanabe,homma}@aoki.ecei.tohoku.ac.jp

あらまし 本稿では，筆者らの提案する算術アルゴリズム記述言語 ARITH を用いた算術演算回路の形式的設計について述べる．ARITH では，多進数系や冗長数系を含む多様な数系に基づく算術アルゴリズムを整数方程式により記述可能である．ARITH によって記述された算術演算回路は，数式処理を用いた検証によってその機能の正当性が検証される．本稿では，提案する検証手法と従来の*BMD に基づく形式的検証手法を比較し，両者の利点を組み合わせることで効率的な検証が可能となることを示す．

キーワード 算術アルゴリズム，形式的検証，データパス，ハードウェア記述言語

Formal Verification Method for Arithmetic Circuits and Its Evaluation

Yuki WATANABE†, Naofumi HOMMA†, Takafumi AOKI†, and Tatsuo HIGUCHI††

† Graduate School of Information Sciences Tohoku University Aoba-yama 6-6-05, Sendai, 980-8579 Japan
†† Department of Electronic Engineering, Faculty of Engineering, Tohoku Institute of Technology
Kasumi-cho 35-1, Yagiyama, Taihaku-ku, Sendai, 982-8577 Japan
E-mail: †{watanabe,homma}@aoki.ecei.tohoku.ac.jp

Abstract This paper presents the formal design of arithmetic circuits based on arithmetic description language called ARITH. By using ARITH, we can describe a wide variety of arithmetic algorithms including those using unconventional number systems. The functionality of arithmetic algorithms in ARITH can be formally verified using formula manipulation methods. In this paper, we compare the proposed formula-based method with the conventional *BMD-based method, and demonstrate that the combination of the two methods enables to verify arithmetic circuits in an efficient way.

Key words arithmetic algorithms, formal verification, datapaths, hardware description language

1. はじめに

身の回りのあらゆる機器に VLSI システムが搭載されるユビキタス社会においては，システムの性能を左右するデータパス（算術演算回路）を用途に応じて適切に設計する必要がある．算術演算回路の性能は，デバイスレベルや論理レベルの最適化のみならず，算術演算のハードウェアアルゴリズム（算術アルゴリズム）に大きく依存する．そのため，これまでに数多くの有用な算術アルゴリズムが考案されてきた [1], [2]．従来の 2 進数系にとらわれない，多進数系や冗長数系に基づく算術アルゴリズムも考案されており，それらが高い性能を発揮することも示されている [3]．

一方で，現在の VLSI 設計技術は論理回路の合成手法を基本として発展しており，データパスの高性能化に必須となる算術アルゴリズムの設計に対して十分な設計環境が整っていない．従来のハードウェア記述言語（HDL: Hardware Description Language）では，論理式によってデータパスを記述するため，算術式を基本とする算術アルゴリズムを直接記述することができない．また，2 進数系以外の算術アルゴリズムを HDL で記述するためには，論理信号に基づいた低水準の記述が必要となる．

また，算術演算回路の設計では，その機能検証も深刻な問題となる．現在広く用いられている論理シミュレーションによる機能検証では，入力信号に対して考えうるテストパターンの数が指数関数的に増加するため，完全な検証は事実上不可能である．これに対して，回路の正しさを数学的に証明する検証手法（形式的検証手法）が数多く研究されている [4]．算術演算回路の形式的検証としては，BDD（Binary Decision Diagram）や*BMD（multiplicative Binary Moment Diagram）といったグラフ構造の等価性判定を用いた検証手法 [5], [6] がよく知られている．これらの手法を用いることで，ある程度の規模の算

術演算回路については完全な検証が可能となった．また，大規模な算術演算回路を検証する場合は，いくつかの階層に分けて回路を表現し，それぞれの階層間で等価性判定による検証を行う．しかし，従来の設計手法では，各階層の演算器機能が明らかでないことが多く，階層化のためには各算術アルゴリズムの機能を明示的に表現・記述することが不可欠となる．

本研究グループでは，上記の問題を解決するため，算術アルゴリズム記述言語 ARITH とその処理系（ARITH 処理系）を用いた算術演算回路の設計手法を提案している [7], [8]．提案する手法では，加算，減算および乗算といった主要な演算の多くが基本的な整数演算の組み合わせのみで表現できることに着目し，数系と整数方程式によって算術アルゴリズムを記述する．ARITH は，(i) 非 2 進数系を含む任意の重み数系に基づく算術アルゴリズムを形式的に記述可能，(ii) 算術アルゴリズムの機能を数式処理によって形式的に検証可能，(iii) 機能保証された算術アルゴリズムを従来のハードウェア記述言語に変換可能などの特徴を有する．

本稿では，ARITH を用いた算術アルゴリズムの形式的検証手法について述べる．まず，ARITH によって記述された算術アルゴリズムの機能を数式処理を用いて形式的に検証できることを示す．次に，従来の*BMD 等価性判定を用いた形式的検証手法が ARITH 記述の検証にも適用可能であることを示し，数式処理を用いた検証手法との差異を考察する．さらに，数式処理と*BMD 等価性判定の利点を組み合わせることで，算術アルゴリズムの検証時間を大幅に削減できることを示す．

2. 算術アルゴリズム記述言語 ARITH

本節では，算術アルゴリズム記述言語 ARITH の概要を述べる．ARITH の詳細は [8] や本研究グループの Web ページ [9] を参照されたい．

2.1 算術アルゴリズムの形式的表現

ARITH では，算術アルゴリズムを整数方程式を用いて表現する．例えば，冗長 2 進加算器の機能は以下の式で表現される．

$$S = X + Y. \tag{1}$$

ARITH において，整数方程式は整数信号と算術演算子からなり，整数信号は特定の整数値をとる．例えば式 (1) の整数信号 S, X, Y は，それぞれ $\{-2,-1,0,1,2\}$, $\{-1,0,1\}$, $\{-1,0,1\}$ の整数値をとる．整数信号のとりうる値は，数系によって決定される．ARITH で扱う数系は，重みベクトル $\boldsymbol{W}$ と桁集合ベクトル $\boldsymbol{D}$ で表される任意の重み数系 [1] $\langle \boldsymbol{W}, \boldsymbol{D}\rangle$ である．$\boldsymbol{W}$ および $\boldsymbol{D}$ は次式で与えられる．

$$\begin{aligned} \boldsymbol{W} &\triangleq \langle w_h, w_{h-1}, \cdots, w_i, \cdots, w_{l+1}, w_l\rangle, \\ \boldsymbol{D} &\triangleq \langle D_h, D_{h-1}, \cdots, D_i, \cdots, D_{l+1}, D_l\rangle. \end{aligned} \tag{2}$$

ここで，h と l は，桁の最上位と最下位を表す整数であり，$h \geq l$ である．h と l の 2 項組 $\langle h, l\rangle$ を数系のレンジ制約と呼ぶ．w_i は，i $(l \leq i \leq h)$ 桁目の重みを表す整数であり，D_i は，i 桁目の桁のとりうる整数の集合 (桁集合) である．

以上の表記法を用いることで，任意の重み数系を定義することができる．例えば，桁集合 $\{-1,0,1\}$ の冗長 2 進数系 $SD2,1$ は以下の式で定義される．

$$\begin{aligned} \boldsymbol{W_{SD2,1}} &\triangleq \langle 2^h, 2^{h-1}, \cdots, 2^{l+1}, 2^l\rangle, \\ \boldsymbol{D_{SD2,1}} &\triangleq \langle \{-1,0,1\}, \{-1,0,1\}, \cdots, \\ &\quad \{-1,0,1\}, \{-1,0,1\}\rangle. \end{aligned} \tag{3}$$

整数信号とは，レンジ制約 $\langle h, l\rangle$ の与えられた数系 $\langle \boldsymbol{W}, \boldsymbol{D}\rangle$ を属性として持つ整数の変数である．例えば，式 (1) の S は $\langle h, l\rangle = \langle 1, 0\rangle$ の制約を与えた数系 $SD2,1$ に基づく整数信号であり，X, Y は $\langle h, l\rangle = \langle 0, 0\rangle$ の制約を与えた数系 $SD2,1$ に基づく整数信号である．

また ARITH では，整数信号のある一桁の信号を取り出した桁信号を扱うことができる．数系 $\langle \boldsymbol{W}, \boldsymbol{D}\rangle$ に基づく整数信号 X の i $(l \leq i \leq h)$ 桁目の桁信号 x_i は，重み w_i と桁集合 D_i の 2 項組 $\langle w_i, D_i\rangle$ を属性として持つ．ここで，整数信号 X とその桁信号 x_i の関係は，次式で与えられる．

$$X = x_h + x_{h-1} + \cdots + x_i + \cdots + x_{l+1} + x_l. \tag{4}$$

整数の集合は算術区間によって表現される．算術区間 $[min, max, step]$ は次の式で与えられる整数の集合を表す．

$$\begin{aligned} &[min, max, step] \\ &\triangleq \{u \in \mathbf{Z} \mid (min \leq u) \ \wedge \ (u \leq max) \\ &\qquad \wedge \ \exists k (k \in \mathbf{Z}_{0+} \ \wedge \ u = min + step \cdot k)\}. \end{aligned} \tag{5}$$

ここで，$\mathbf{Z}_{0+}$ は非負整数である．また，min, max, $step$ は，$min \leq max$, $step \geq 0$ とする．算術区間は自由に加算・乗算することができる．例えば，式 (1) の整数信号 S, X, Y の算術区間はそれぞれ $[-2,2,1]$, $[-1,1,1]$, $[-1,1,1]$ で表現される．また，式 (1) の右辺 $X+Y$ の算術区間は X と Y の算術区間の加算により，$[-2,2,1]$ と導くことができる．

ARITH では，比較や分岐など任意の論理演算を含む算術アルゴリズムも論理信号と論理式を用いて表現することができる．論理信号を用いる場合，整数信号の論理信号への符号化定義関数: $\mathbf{B}^n \rightarrow \mathbf{Z}$ を与える必要がある．n 本の論理信号 $L_{1,i}, \cdots, L_{n,i}$ による，桁集合の要素 d_i の符号化は符号化定義関数 E_i を用いて以下のように表現する．

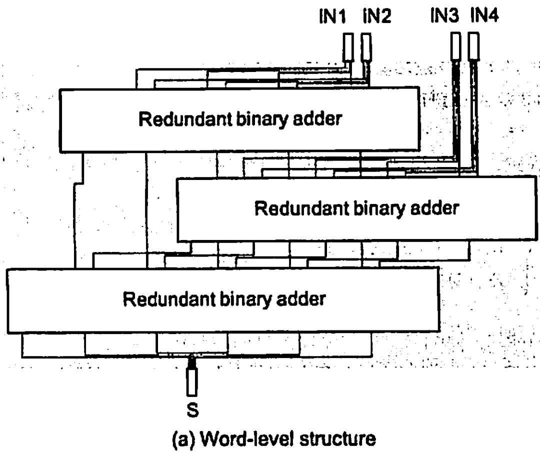
$$d_i = E_i(L_{1,i}, \cdots, L_{n,i}). \tag{6}$$

以上のように，提案する ARITH では厳密に定義された数系を用いることで，任意の重み数系に基づく算術アルゴリズムを形式的に表現することが可能となる．

2.2 ARITH を用いた算術アルゴリズムの設計

図 1 の冗長 2 進加算木を例に ARITH を用いた設計について概説する．図 1 (a) はワードレベルの加算木構造を表し，図 1 (b) は桁レベルの加算木構造を表す．ARITH 記述は，数系を記述するための数系定義ブロックと，算術アルゴリズムの機能や内部構造を記述するためのモジュール記述ブロックから構成される．

数系定義ブロックでは，整数信号の型となる数系を定義する．



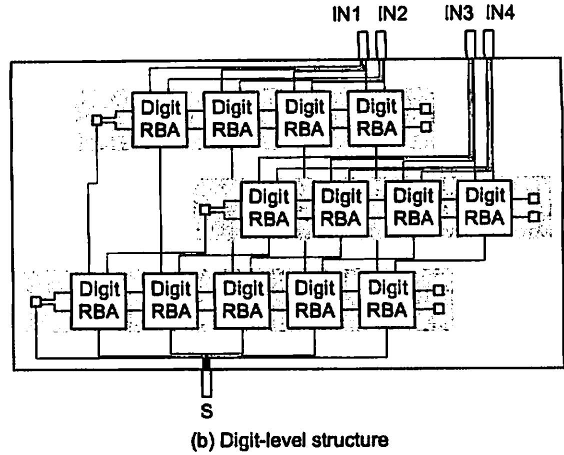


図 1　冗長 2 進加算木の構造

```
1:  typedef SD2_1;
2:    for(i, SD2_1.low, SD2_1.high) begin
3:      SD2_1{i}.weight = Power(2, i);
4:      SD2_1{i}.min = -1;
5:      SD2_1{i}.max = 1;
6:      SD2_1{i}.step = 1;
7:    end
8:  endtypedef
```

図 2　冗長 2 進数系の ARITH 記述例

式 (3) で定義される $SD2,1$ 進数の数系定義ブロックを図 2 に示す．数系定義ブロック内で用いられる予約語の意味は以下のとおりである．

SD2_1.high	: 最上位桁（レンジ制約の h）
SD2_1.low	: 最下位桁（レンジ制約の l）
SD2_1{i}.weight	: i 桁目の重み
SD2_1{i}.min	: i 桁目の桁集合の min
SD2_1{i}.max	: i 桁目の桁集合の max
SD2_1{i}.step	: i 桁目の桁集合の $step$

図 2 では，2 − 7 行目の記述によって SD2_1.low 桁目から SD2_1.high 桁目までの重みベクトルと桁集合ベクトルを定義している．

一方，モジュール記述ブロックでは，整数方程式と数系定義ブロックで定義された数系を用いて算術アルゴリズムの機能と構造を記述する．図 3 に図 1(a) に対応する 4 入力冗長 2 進加算木の記述例を示す．モジュール記述ブロックは，モジュール名およびポートリスト（1 行目），入出力信号の宣言と制約（2–13 行目），機能表明（14 行目），内部構造記述（15–25 行目）から構成される．図 3 のように，機能表明は整数方程式で与え，内部構造はより抽象度の低い下位モジュールの組み合わせで記述する．各下位モジュールについても同様に機能表明と内部構造を記述することで階層的な設計を行う．

下位モジュールの組み合わせで表現できない最下位モジュールや，比較や条件分岐などの論理演算を含むモジュールは，論理信号と論理式によって記述する．論理信号を扱うときは，整数信号の符号化を数系定義ブロック内に記述する．図 4 に，符

```
1:  module RBtree(Z, IN0, IN1, IN2, IN3);
2:    output SD2_1 Z;
3:    input SD2_1 IN0;
4:    input SD2_1 IN1;
5:    input SD2_1 IN2;
6:    input SD2_1 IN3;
7:    constraint begin
8:      Z.high = 5; Z.low = 0;
9:      IN0.high = 3; IN0.low = 0;
10:     IN1.high = 3; IN1.low = 0;
11:     IN2.high = 3; IN2.low = 0;
12:     IN3.high = 3; IN3.low = 0;
13:   end
14:   assertion Z = IN0 + IN1 + IN2 + IN3;
15:   structure begin
16:     wire SD2_1 W0;
17:     wire SD2_1 W1;
18:     constraint begin
19:       W0.high = 4; W0.low = 0;
20:       W1.high = 4; W1.low = 0;
21:     end
22:     RBA_3_0 U0(W0, IN0, IN1);
23:     RBA_3_0 U1(W1, IN2, IN3);
24:     RBA_4_0 U2(Z, W0, W1);
25:   end
26: endmodule
```

図 3　冗長 2 進加算木の ARITH 記述例

号なし 2 進数系の ARITH 記述を示す．2–7 行目の数系定義に加え，8–16 行目で符号化を定義している．ここでは，論理信号の宣言（9 行目）と制約（10–12 行目），符号化の定義式（13–15 行目）を記述している．

論理演算を含む算術アルゴリズムの例として全加算器の ARITH 記述を図 5 に示す．全加算器の機能は整数方程式を用いて $C+S=X+Y+Z$（11 行目）のように記述されるが，その内部構造は論理式を用いて 13–14 行目のように記述される．ここで，C#L, S#L, X#L, Y#L, Z#L はそれぞれ整数信号 C, S, X, Y, Z に対応する論理信号である．

3. ARITH に基づく算術演算回路の形式的検証

ARITH で記述された算術アルゴリズムの機能は，ARITH 処理系によって検証される．一般に，算術演算回路の機能検証手法としては，論理シミュレーションの他に，ワードレベルの

```
 1:  typedef UB;
 2:    for (i, UB.low, UB.high) begin
 3:      UB{i}.weight = Power(2,i);
 4:      UB{i}.min = 0;
 5:      UB{i}.max = 1;
 6:      UB{i}.step = 1;
 7:    end
 8:    encoding enc_type begin
 9:      logic L[];
10:      constraint begin
11:        L.high = UB.high; L.low = UB.low;
12:      end
13:      for (i, UB.low, UB.high) begin
14:        UB{i} = UB{i}.weight * L[i];
15:      end
16:    end
17:  endtypedef
```

図 4　符号化定義の ARITH 記述

```
 1:  module FULL_ADDER (C, S, X, Y, Z);
 2:    output UB<enc_type> C, S;
 3:    input  UB<enc_type> X, Y, Z;
 4:    constraint begin
 5:      C.high = 1; C.low = 1;
 6:      S.high = 0; S.low = 0;
 7:      X.high = 0; X.low = 0;
 8:      Y.high = 0; Y.low = 0;
 9:      Z.high = 0; Z.low = 0;
10:    end
11:    assertion C + S = X + Y + Z;
12:    structure begin
13:      assign C#L = X#L&Y#L|Z#L&(X#L|Y#L);
14:      assign S#L = X#L^Y#L^Z#L;
15:    end
16:  endmodule
```

図 5　論理演算を含む算術アルゴリズムの ARITH 記述

決定木や*BMD [5], [6] などのグラフ構造を用いた形式的検証手法が知られている．ARITH 記述もまた，*BMD を用いて検証することができる．一方で，ARITH 記述は数系と整数方程式を基本とするため，特別なデータ構造を用いない，数式処理による形式的な検証が可能となる．いずれの手法も，表明された機能と実装された内部構造が一致するかを調べる等価性判定である．以下では，提案する数式処理による検証手法と，*BMD 等価性判定を用いた検証手法について述べ，その検証時間を評価する．

3.1　数式処理を用いた機能検証

提案する形式的検証手法は，「数式評価」と「レンジ評価」から構成される．

数式評価では，内部構造が機能表明と等価であるかどうかを数式処理によって評価する．下位モジュールの機能表明から抽出した連立方程式を解くことによって得られる整数方程式が，評価対象のモジュールの機能表明と一致する場合のみ，モジュールの機能が満たされる．連立方程式の解法には，ガウスの消去法やグレブナー基底を用いた消去法などの代数的手法を利用することができる．

レンジ評価では，整数方程式の両辺の値域を区間演算によって求め，左辺（出力側）の値域が右辺（入力側）の値域を含んでいるかどうかを調べる．区間演算では，出力側の値域は下界へ近似され，入力側の値域は上界へ近似される．出力側の値域が入力側の値域を包含しない場合は，入力に対する演算結果を出力信号で表現できない場合が存在するため，ハードウェア上で実現できない．

[例 1] 冗長 2 進加算木の検証

図 3 の ARITH 記述に対する検証を示す．まず，数式評価では，下位モジュールの機能表明を抽出することで，次の連立方程式を得る．

$$\begin{cases} W0 = IN0 + IN1, \\ W1 = IN2 + IN3, \\ Z = W0 + W1. \end{cases} \tag{7}$$

この連立方程式を Z, $IN0$, $IN1$, $IN2$, $IN3$ について解くことによって，機能表明 $Z = IN0 + IN1 + IN2 + IN3$ が導出されるため，モジュールの機能が代数的に満たされることが保証される．レンジ評価では，機能表明の左辺の値域，右辺の値域を評価するとそれぞれ，$[-63, 63, 1]$，$[-60, 60, 1]$ と近似される．左辺の値域が右辺の値域を包含するため，このモジュールはハードウェア上で実現することができる．

[例 2] 全加算器の検証

図 5 の ARITH 記述に対する検証を示す．まず，数式評価では，下位モジュールの機能表明を抽出することで，次の連立方程式を得る．

$$\begin{cases} C_L &= X_L \,\&\, Y_L \mid Z_L(X_L \mid Y_L), \\ S_L &= X_L \hat{}\, Y_L \hat{}\, Z_L. \end{cases} \tag{8}$$

ここで，連立方程式を解くために，論理式を疑似論理式に変換する．

$$\begin{cases} C_L &= X_LY_L + (X_L + Y_L - X_LY_L)Z_L \\ &\quad -X_LY_L(X_L + Y_L - X_LY_L)Z_L, \\ S_L &= X_L + Y_L - 2X_LY_L + Z_L \\ &\quad -2(X_L + Y_L - 2X_LY_L)Z_L. \end{cases} \tag{9}$$

式 (9) は，論理信号に対してべき等律（$C_L{}^2 = C_L$，$S_L{}^2 = S_L$，$X_L{}^2 = X_L$，$Y_L{}^2 = Y_L$，$Z_L{}^2 = Z_L$）が成り立つという条件の下で式 (8) と等価な機能を持つ．また，図 4 より整数信号の符号化は以下の式で与えられる．

$$\begin{cases} C &= 2 \times C_L, \\ S &= 1 \times S_L, \\ X &= 1 \times X_L, \\ Y &= 1 \times Y_L, \\ Z &= 1 \times Z_L. \end{cases} \tag{10}$$

式 (9)，(10) からなる連立方程式を解くことによって，整数方程式 $C + S = X + Y + Z$ が導出されるため，モジュールの機能が代数的に満たされることが保証される．レンジ評価では，機能表明の左辺の値域，右辺の値域を評価するとそれぞれ，$[0, 3, 1]$，$[0, 3, 1]$ と近似される．左辺の値域が右辺の値域を包含するため，このモジュールはハードウェア上で実現可能となる．

数式処理では，以上のように，内部構造を表す連立方程式から入力信号以外を消去することによって機能表明の方程式が導

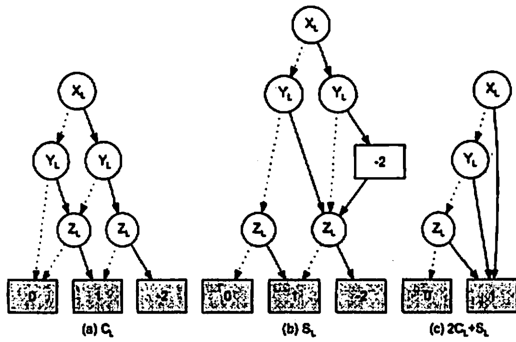


図6 *BMDによる全加算器の表現

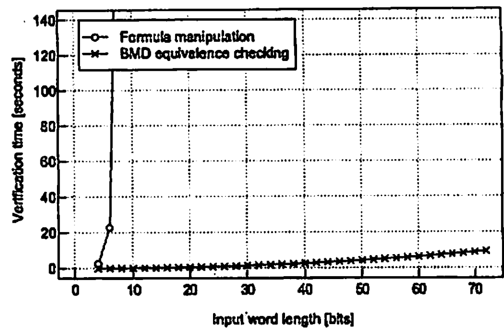


図7 高速加算モジュール（Kogge-Stone adder）の検証時間

けるかどうかを評価する．数式処理による検証時間は，連立方程式の解法にガウスの消去法を用いるか，グレブナー基底による消去法を用いるかによって異なる．以下では，数式処理による計算時間について考察する．

消去対象の信号について各方程式が線形のとき，ガウスの消去法を用いることができる．算術アルゴリズムの検証時間 T は，整数信号の数 s と方程式の数 m，比例記号 $\propto$ を用いて次のように表せる．

$$T \propto s^2 \times m. \tag{11}$$

ここで，算術アルゴリズムへの入力信号幅を n とすると，各モジュールへの入力整数信号の数および下位モジュール呼び出しの数は，n に比例するため，s と m は $O(n)$ 程度になる．したがって，ガウスの消去法に必要な計算量は，$O(n^3)$ 程度となる．このように，線形方程式の場合は算術アルゴリズムの検証を多項式時間で終えることができる．

一方，非線形な連立方程式の場合は，ガウスの消去法を適用できないため，グレブナー基底を用いた消去法[10]を用いる．特に，論理演算は非線形な演算であるため，グレブナー基底を用いた消去法を必要とする．算術アルゴリズムの検証に利用する場合は，内部構造を表す連立方程式からグレブナー基底を導出し，求めた基底によって機能表明を表す方程式を簡約化する．簡約化の結果が0になった場合，機能表明の方程式が内部構造の方程式系に属することがわかる．これは，機能表明が内部構造によって導出可能であることを意味する．任意の多項式の組からグレブナー基底を求めるためのアルゴリズム（ブッフバーガーのアルゴリズム）が提案されており，そのアルゴリズムが必ず停止することも証明されている．そのため，非線形な演算を含む任意の算術アルゴリズムをグレブナー基底により検証することができる．しかしながら，与えられた多項式の組によっては，グレブナー基底の導出に膨大な計算時間と記憶スペースが必要となる．

3.2 *BMD等価性判定を用いた機能検証

*BMD等価性判定を用いた機能検証では，数式処理を用いた機能検証と同様に，機能表明の式と内部構造の式が一致するかどうかを調べる．このとき，各式を表す*BMDを構築し，グラフの同形判定により等価性を判定する．例えば全加算器の機能検証の場合，図5の13行目と14行目の式からは，それぞれ図6 (a) と図6 (b) の*BMDが得られる．これらを用いて機能表明の左辺（整数信号の符号化後は $2C_L + S_L$）を表す*BMDを構築すると，図6 (c) が得られる．これは，機能表明の右辺（整数信号の符号化後は $X_L + Y_L + Z_L$）の*BMDと等しい．よって，内部構造が機能表明と一致することがわかる．

以下では，*BMD等価性判定による検証時間について考察する．*BMD等価性判定処理では，内部構造を表す連立方程式から順次*BMDを構築することによって，最終的に出力信号を表す*BMDを構築する．*BMD構築アルゴリズムでは，同形のグラフは必ず同じメモリ上に格納されるため，同形判定は定数時間で行うことできる．したがって，*BMD等価性判定の計算量 T は，*BMD構築に必要な時間 $T_{construct}$，方程式の数 m を用いて次のように表せる．

$$T \propto T_{construct} \times m. \tag{12}$$

ここで $T_{construct}$ は，演算対象の*BMDのグラフサイズを g，方程式の項数を t とすると，次のように表すことができる．

$$T_{construct} \propto g^2 \times t. \tag{13}$$

ここで，算術アルゴリズムへの入力信号幅を n とすると，各モジュールへの入力整数信号の数 s，入力整数信号を表現するための論理信号の数 w，下位モジュール呼び出しの数（= 方程式の数 m），および項数 t はそれぞれ $O(n)$ 程度である．このとき，$s \times w$ に比例する*BMDのグラフサイズ g は $O(n^2)$ 程度になる．従って，*BMD等価性判定全体の計算量 T は $O(n^6)$ 程度となる．

*BMD等価性判定を用いた検証では，整数信号を必ず符号化する必要があるため，整数方程式のみで表現されるモジュールの検証効率は数式処理に比べて悪い．一方，論理演算を効率的に表現することができるため，論理演算を含むモジュールの検証効率は数式処理よりも良い．

3.3 検証時間の測定

いくつかの算術アルゴリズムについて，数式処理による検証時間と*BMD等価性判定による検証時間を測定した．数式処理にはMathematica 5.2を使用した．

図7は高速加算モジュール（Kogge-Stone adder）の検証時間である．Kogge-Stone adderは大規模な論理式で構造が表

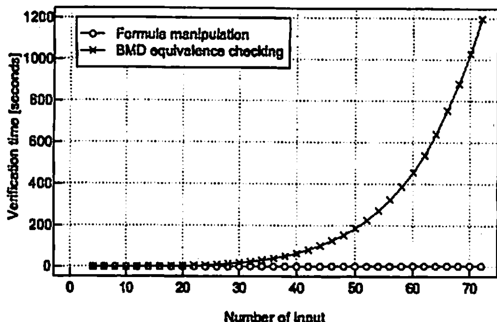


図 8　配列型多入力加算モジュールの検証時間

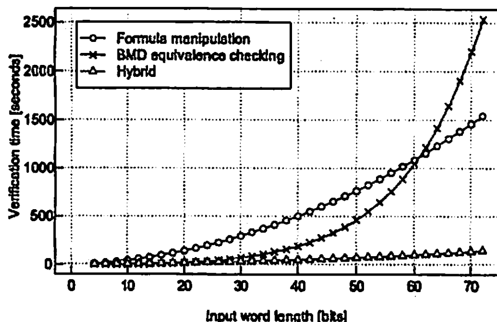


図 9　配列型乗算器の検証時間

現される．そのため，数式処理では計算コストの大きいグレブナー基底を用いた消去法を行う必要があり，小規模な加算器しか検証できていない．一方で，*BMD 等価性判定では大規模な加算器も効率的に検証することができる．

図 8 は n ビット n 入力加算器モジュールの検証時間である．数式処理ではガウスの消去法が適用できるため，効率的な検証が行える．一方，*BMD 等価性判定は，64 ビット程度の加算器でも現実的な時間で検証を行うことができるが，整数信号を論理信号まで展開するため，数式処理に比べて大幅に検証時間を用する．

図 9 は符号なし 2 進数配列型乗算器の検証時間である．最終段加算器には Ripple carry adder を用いた．図 9 の Hybrid は，論理演算を含むモジュールのみを*BMD 等価性判定によって検証し，それ以外を数式処理によって検証したときの計算時間である．数式処理と*BMD 等価性判定を併用することで検証時間が大幅に削減されることを確認できる．

図 10 は，さまざまな構成の積和演算器の検証時間を入力信号幅毎にプロットしたものである．検証対象の積和演算器は，筆者らが構築した演算器モジュールジェネレータ [8], [9] から生成された 100 種類の異なる構成であり，冗長数系を含む多様な数系に基づく算術アルゴリズムを実現している．機能検証には，数式処理と*BMD 等価性判定を組み合わせた手法を用いた．この結果から，ARITH を用いた設計においては，大規模な演算器の機能を現実的な時間で完全に検証できることがわかる．

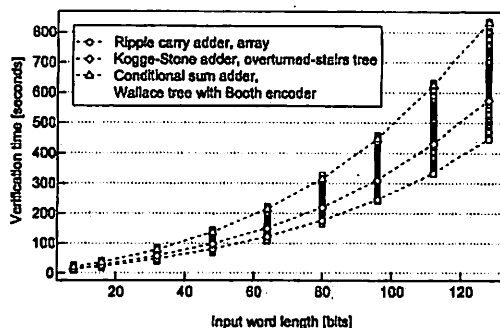


図 10　積和演算器 ($S = X * Y + W + Z$) の検証時間

4. ま と め

本稿では，算術アルゴリズム記述言語 ARITH に基づく算術演算回路の設計・検証手法について述べた．ARITH で記述された算術アルゴリズムの機能は数式処理を用いて形式的に検証できることを示した．また，ARITH 記述の検証に*BMD 等価性判定を用いた形式的検証手法が適用可能であることを述べ，提案する数式処理を用いた形式的検証手法と比較した．数式処理と*BMD 等価性判定との違いは，整数変数を論理変数に符号化することなく扱うことができるか否かにある．一般に整数演算の組み合わせで構築される算術アルゴリズムの検証には整数変数を扱うことのできる数式処理が適している．一方で，論理演算を含む算術アルゴリズムには，*BMD 等価性判定が適している．本稿では，算術演算と論理演算の検証にそれぞれ数式処理と*BMD 等価性判定を用いることで，積和演算等の代表的な演算器の検証時間を大幅に削減できることを示した．

文　献

[1] I. Koren: " Computer arithmetic algorithms 2nd Edition", A K Peters (2001).
[2] B. Parhami: "Computer Arithmetic: Algorithms and Hardware Designs", Oxford University Press (2000).
[3] T. Aoki and T. Higuchi: "Beyond-binary arithmetic — Algorithms and VLSI implementations —", Interdisciplinary Information Sciences, 6, 1, pp. 75–98 (2000).
[4] R. Drechsler Ed.: "Advanced Formal Verification", Kluwer Academic Publishers (2004).
[5] E. R. Bryant and Y.-A. Chen: "Verification of arithmetic circuits with binary moment diagrams", Proc. of 32nd Design Automation Conference, pp. 535 – 541 (1995).
[6] Y.-A. Chen and R. E. Bryant: "ACV: an arithmetic circuit verifier", ICCAD '96: Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design, Washington, DC, USA, IEEE Computer Society, pp. 361–365 (1996).
[7] K. Ishida, N. Homma, T. Aoki and T. Higuchi: "Design and verification of parallel multipliers using arithmetic description language: ARITH", Proc. 34th IEEE Int. Symp. Multiple-Valued Logic, pp. 334 – 339 (2004).
[8] N. Homma, Y. Watanabe, T. Aoki and T. Higuchi: "Formal design of arithmetic circuits based on arithmetic description language", IEICE Trans. Fundamentals, E89-A, 12 (2006).
[9] Arithmetic Description Language : ARITH. http://www.aoki.ecei.tohoku.ac.jp/arith/
[10] D. A. Cox, J. B. Little and D. O'Shea: "Ideals, Varieties, and Algorithms", Springer-Verlag, NY, 2nd edition (1996).