2004－SLDM－117 (41)
2004／12／2

# 算術アルゴリズム記述言語を用いた乗算器モジュールジェネレータの構築

石田 一哉† 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 宮城県仙台市青葉区荒巻字青葉05
†† 東北工業大学 工学部 電子工学科 〒982-8577 宮城県仙台市太白区八木山香澄町35番1号
E-mail: †{ishida,homma}@aoki.ecei.tohoku.ac.jp

**あらまし** 本稿では，算術アルゴリズム記述言語 ARITH を用いた算術演算回路の設計手法について述べる．ARITH を用いることで，(i) 算術アルゴリズムの形式的な記述と (ii) 記述された算術アルゴリズムの形式的な検証，(iii) 算術アルゴリズムの HDL 記述への変換が可能になる．その応用として，ARITH を用いた乗算器モジュールジェネレータについて述べる．構築した乗算器モジュールジェネレータは，250 種類を越える乗算アルゴリズムの取り扱いが可能であり，生成した乗算器モジュールの機能をアルゴリズムレベルで完全に保証するという特長を有する．

**キーワード** 算術アルゴリズム，ハードウェア記述言語，形式的検証，モジュールジェネレータ，データパス，並列乗算器

# Development of a Multiplier Module Generator Using Arithmetic Description Language

Kazuya ISHIDA†, Naofumi HOMMA†, Takafumi AOKI†, and Tatsuo HIGUCHI††

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

**Abstract** This paper presents a design method for arithmetic circuits using an arithmetic description language: ARITH. The use of ARITH makes possible (i) formal description of arithmetic algorithms, (ii) formal verification of described arithmetic algorithms, and (iii) translation of arithmetic algorithms to equivalent HDL codes. This paper also presents an application of ARITH to a multiplier module generator. The developed generator can handle over 250 types of multiplication algorithms, and produce the multiplier modules whose functions are completely verified at the algorithmic level.

**Key words** arithmetic algorithms, hardware description language, formal verification, module generator, datapaths, parallel multipliers

## 1. まえがき

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，システムの性能を大きく左右するデータパスを用途に応じて適切に設計する必要がある．データパスの大部分を占める算術演算回路の性能は，デバイスレベルや論理レベルでの最適化のみならず，算術演算のアルゴリズムに大きく左右される．そのため，これまでに多くの有用な算術アルゴリズムが提案されてきた [1] [2] [3]．

一方，VLSI システムの設計において，ハードウェア記述言語（Hardware Description Language: HDL）を用いたハイレベル設計手法が普及している．HDL では論理式によって回路を記述するため，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はできない．HDL によって高度な算術アルゴリズムを記述するためには 2 値論理信号に基づいた低水準の構造記述が必要となる．また，一般に論理シミュレーションや論理ベースの形式的検証手法を用いて多入力・多出力の算術演算回路の機能検証を行うことは困難である．算術演算回路に対する論理ベースの検証が困難であるという問題は，ハードウェア／ソフトウェア協調設計への移行とともに近年急速に普及してい

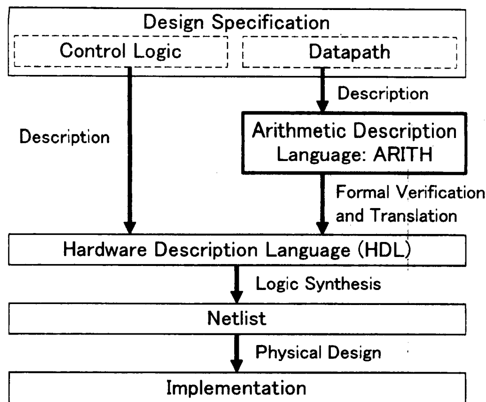



図1 ARITH を導入したシステム設計フロー

る SystemC などの記述言語にもあてはまる．

このような問題を解決するために，我々はこれまでに算術アルゴリズム記述言語 ARITH と，その処理系を用いた新しいデータパス設計手法を提案している[4]. ARITH を用いることで，算術アルゴリズムの形式的な記述と検証が可能となる．さらに，ARITH のソースコードから等価な HDL のソースコードを自動生成することによって，ARITH によるデータパス設計を従来の HDL による回路設計フローに統合することができる(図1).

本稿では，ARITH を用いたデータパス設計の概要について述べる．その例として並列乗算器をとり上げ，その構造が階層的に記述可能であること，記述された回路の機能が数式処理により形式的に検証可能であることを示す．次に，ARITH を用いた乗算器モジュールジェネレータについて述べる．機能検証に ARITH を用いることによって検証時間が大幅に削減されることを示し，ARITH を用いたデータパス設計手法の有効性を示す．さらに，乗算器モジュールジェネレータから生成される乗算器の性能を示す．

## 2. 算術アルゴリズム記述言語 ARITH を用いたデータパス設計

ARITH は算術アルゴリズムの記述に特化した設計言語であり，データパスの効率的な設計を支援する．本章では，ARITH の記述対象となる算術アルゴリズムの形式的表現について述べた後，ARITH の記述例を示す．また，ARITH の言語処理系によって行われる機能検証についても述べる．

### 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数系と整数方程式により形式的に表現する．

ARITH で扱う数系は，重みベクトル $W$ と桁集合ベクトル $D$ の二項組 $\langle \boldsymbol{W}, \boldsymbol{D} \rangle$ によって与えられる任意の重み数系である．$\boldsymbol{W}$ および $\boldsymbol{D}$ は次式で与えられる．

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

```
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:  endtypedef
```

図2 符号無し2進数の記述例

ここで，$h$ と $l$ は，桁の最上位と最下位を表す整数であり，$h \geqq l$ 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，$i$ 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．例えば，符号無し2進数（UB）を表す二項組 $\langle \boldsymbol{W}_{UB}, \boldsymbol{D}_{UB} \rangle$ は以下のように与えられる．

$$\begin{aligned} \boldsymbol{W_{UB}} &\triangleq \langle 2^h, 2^{h-1}, \cdots, 2^i, \cdots, 2^{l+1}, 2^l \rangle \\ \boldsymbol{D_{UB}} &\triangleq \langle \{0,1\}, \cdots, \{0,1\} \rangle \end{aligned} \quad (2)$$

一方，整数方程式は，整数信号および整数定数を加算 +，減算 −，乗算 × で結合して得られる整数式から成る方程式である．ARITH が扱う信号は全て，数系 $\langle \boldsymbol{W}, \boldsymbol{D} \rangle$ とレンジ制約 $\langle h, l \rangle$ を属性として持つ整数信号である．また，整数信号内の特定の桁を取り出したものを桁信号と呼ぶ．ここで，整数信号 $X$ とその桁信号 $x_i (l \leqq i \leqq h)$ の関係は次式で与えられる．

$$X = x_h + x_{h-1} + \cdots + x_i + \cdots + x_{l+1} + x_l \quad (3)$$

ARITH における整数の集合は，以下に定義する算術区間によって表現される．$min$, $max$ を整数とし，$step$ を正の整数としたとき，算術区間 $[min, max, step]$ は次式で与えられる．

$$\begin{aligned} &[min, max, step] \\ &\triangleq \{u \in \mathbf{Z} \mid (min \leq u) \ \wedge \ (u \leq max) \\ &\qquad \wedge \ (\exists j \in \mathbf{Z}_{0+} \bullet u = min + step \cdot j)\} \end{aligned} \quad (4)$$

ここで，$\mathbf{Z}$ は整数の集合，$\mathbf{Z}_{0+}$ は正の整数の集合である．

### 2.2 ARITH 記述

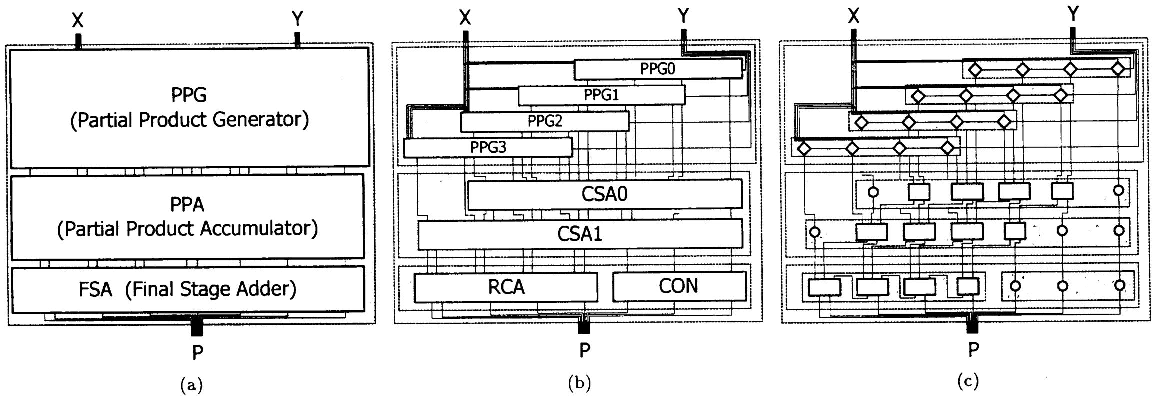
ARITH 記述は，数系を記述するための **typedef** ブロックと，整数方程式を用いて算術アルゴリズムの機能や内部構造を記述するための **module** ブロックから構成される．

**typedef** ブロックでは，整数信号の型となる数系を定義する．式 (2) の符号無し2進数に対応する **typedef** ブロックを図2に示す．**typedef** 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．

| | |
|---|---|
| UB.high | ：最上位桁（レンジ制約の $h$） |
| UB.low | ：最下位桁（レンジ制約の $l$） |
| UB{i}.weight | ：$i$ 桁目の重み |
| UB{i}.min | ：$i$ 桁目の桁集合の $min$ |
| UB{i}.max | ：$i$ 桁目の桁集合の $max$ |
| UB{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 UB.low 桁目から UB.high 桁目までの重みベクトルと桁集合ベクトルが定義される．

一方，**module** ブロックでは，整数方程式と **typedef** ブロッ




(a) (b) (c)

図 3　様々な抽象度で表現した 4 ビット配列型乗算器

```
 1:  module MULTIPLIER(P, X, Y);
 2:    output UB P;
 3:    input  UB X, Y;
 4:    constraint begin
 5:      P.high = 7; P.low = 0;
 6:      X.high = 3; X.low = 0;
 7:      Y.high = 3; Y.low = 0;
 8:    end
 9:    assertion P = X * Y;
10:    structure begin
11:      wire UB PP0, PP1, PP2, PP3;
12:      wire UB S1, S2;
13:      constraint begin
14:        PP0.high = 3; PP0.low = 0;
15:        PP1.high = 4; PP1.low = 1;
16:        PP2.high = 5; PP2.low = 2;
17:        PP3.high = 6; PP3.low = 3;
18:        S1.high = 6; S1.low = 3;
19:        S2.high = 6; S2.low = 0;
20:      end
21:      PPG U0 (PP0, PP1, PP2, PP3, X, Y);
22:      PPA U1 (S1, S2, PP0, PP1, PP2, PP3);
23:      FSA U2 (P, S1, S2);
24:    end
25:  endmodule
```

図 4　4 ビット配列型乗算器の記述例

クで定義された数系とを用いて算術アルゴリズムの機能と構造を記述する．module ブロックは，モジュール名およびポートリスト，入出力信号，機能表明，内部構造の記述からなる．図 3 は，配列型乗算器を様々な抽象度で表現したものであり，左側の図ほど抽象度が高い．白抜きの図形は算術演算機能を持ったモジュールを表し，灰色に塗り潰された部分はひとつ上位のモジュールに対応する．以下では，この 4 ビット配列型乗算器の記述例（図 4）に沿って module ブロックについて説明する．

- モジュール名およびポートリストの記述（1 行目）：

モジュール名（MULTIPLIER）とポートリスト（P, X, Y）が記述される．このモジュールをより抽象度の高い算術アルゴリズムの構成要素として用いる際のインタフェースとなる．

- 入出力信号の記述（2–8 行目）：

入力信号（X, Y）及び出力信号（P）の宣言と，各信号に対する数系とレンジ制約が記述される．2–3 行目では，入出力信号として用いられる整数信号の識別子と数系を宣言し，4–8 行目では high と low を用いてレンジ制約を与えている．

- 機能表明の記述（9 行目）：

整数方程式（P = X * Y）によってモジュールの機能が記述される．機能表明においては，左辺が出力を，右辺は入力を表す．

- 内部構造の記述（10–24 行目）：

内部信号の宣言と制約，下位モジュールの呼び出しが記述される．11–20 行目では内部信号の宣言と制約が記述されている．記述方法は input/output の代わりに wire を用いることを除き，入出力信号の宣言と同じである．21–23 行目では 3 つの下位モジュール PPG, PPA, FSA で内部構造を記述している．

図 5 は，MULTIPLIER の下位モジュールの記述例である．これら下位モジュールも MULTIPLIER 同様に，内部構造はさらに抽象度の低いモジュールを呼び出すことによって記述される．なお，54–55 行目に見られる中括弧{}は，整数信号から桁信号を取り出すことを意味する．ARITH では，このように抽象度の異なる算術演算を結び付けることにより算術アルゴリズムを記述する．ただし，図 3(c) のようにそれ以上抽象度の低い算術演算が無い場合には，内部構造を論理式によって記述する．

### 2.3　ARITH 処理系による機能検証

本節では，ARITH の言語処理系（ARITH 処理系）によって行われる機能検証について述べる．開発した ARITH 処理系は，算術アルゴリズムの正しさを形式的に検証する機能に加えて，ARITH 記述を等価な HDL 記述に変換する機能を有する．ARITH 処理系を用いることで，HDL を用いた従来の回路設計フローに ARITH を統合することが可能となる．

ARITH の記述に対する機能検証は，「数式評価」と「レンジ評価」からなる．数式評価は，構造の記述が機能表明を満たすかどうかを評価する．一方，レンジ評価は，入出力信号のレンジ制約のもとでハードウェアとして実現可能かどうかを評価する．これらの評価は全てのモジュールに対して行われる．以下では，図 4 の MULTIPLIER モジュールを例に各評価を説明する．

- **数式評価**

数式評価では，内部構造から得られる整数方程式と機能表明に記述された整数方程式とが等価であるかどうかを調べる．MULTIPLIER モジュールの場合，内部構造を表す連立方程式は

```
 1:  module PPG(PP0, PP1, PP2, PP3, IN1, IN2);
 2:    output UB PP0, PP1, PP2, PP3;
 3:    input UB IN1, IN2;
 4:    constraint begin
 5:      PP0.high = 3; PP0.low = 0;
 6:      PP1.high = 4; PP1.low = 1;
 7:      PP2.high = 5; PP2.low = 2;
 8:      PP3.high = 6; PP3.low = 3;
 9:      IN1.high = 3; IN1.low = 0;
10:      IN2.high = 3; IN2.low = 0;
11:    end
12:    assertion PP0 + PP1 + PP2 + PP3 = IN1 * IN2;
13:    structure begin
14:      PPG0 U0 (PP0, IN1, IN2{0});
15:      PPG1 U1 (PP1, IN1, IN2{1});
16:      PPG2 U2 (PP2, IN1, IN2{2});
17:      PPG3 U3 (PP3, IN1, IN2{3});
18:    end
19:  endmodule
20:
21:  module PPA(S1, S2, PP0, PP1, PP2, PP3);
22:    output UB S1, S2;
23:    input UB PP0, PP1, PP2, PP3;
24:    constraint begin
25:      S1.high = 6; S1.low = 3;
26:      S2.high = 6; S2.low = 0;
27:      PP0.high = 3; PP0.low = 0;
28:      PP1.high = 4; PP1.low = 1;
29:      PP2.high = 5; PP2.low = 2;
30:      PP3.high = 6; PP3.low = 3;
31:    end
32:    assertion S1 + S2 = PP0 + PP1 + PP2 + PP3;
33:    structure begin
34:      wire UB IC0, IS0;
35:      constraint begin
36:        IC.high = 5; IC.low = 2;
37:        IS.high = 5; IS.low = 0;
38:      end
39:      CSA0 U0 (IC, IS, PP0, PP1, PP2);
40:      CSA1 U1 (S1, S2, IS, IC, PP3);
41:    end
42:  endmodule
43:
44:  module FSA(S, X, Y);
45:    output UB S;
46:    input UB X, Y;
47:    constraint begin
48:      S.high = 7; S.low = 0;
49:      X.high = 6; X.low = 3;
50:      Y.high = 6; Y.low = 0;
51:    end
52:    assertion S = X + Y;
53:    structure begin
54:      RCA U0 (S{7:3}, X{6:3}, Y{6:3});
55:      CON U1 (S{2:0}, Y{2:0});
56:    end
57:  endmodule
```

図 5　4 ビット配列型乗算器の下位モジュールの記述例

$$\begin{cases} PP0+PP1+PP2+PP3 = X \times Y \\ S1+S2 = PP0+PP1+PP2+PP3 \\ P = S1+S2 \end{cases} \quad (5)$$

であり，この連立方程式から数式処理によって内部信号 $PP0$, $PP1$, $PP2$, $PP3$, $S1$, $S2$ を消去すると，整数方程式

$$P = X \times Y \quad (6)$$

が導出される．MULTIPLIER モジュールでは，式 (6) は機能表明の整数方程式（図 4: 9 行目）と等価であるため構造の記述が機能表明を満たしていると言える．数式処理において内部信号が消去できない場合や，導出された整数方程式が機能表明と一致しない場合は，記述された算術アルゴリズムが誤っていることがわかる．

- **レンジ評価**

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．MULTIPLIER モジュールの場合，左辺の値域 $R_l$，右辺の値域 $R_r$ はそれぞれ

$$R_l \supseteq [0, 255, 1] \quad (7)$$

$$R_r \subseteq [0, 225, 1] \quad (8)$$

と近似される．よって，$R_l \supseteq R_r$ を満足する．

上記のように，ARITH で記述されたモジュールを検証するためには，内部構造を構成する下位モジュールが正しいと仮定し，それらの機能表明を評価に利用する．その結果，各モジュールを独立に検証することが可能となり，全ての階層を結合して行う従来の検証手法に比べて大幅に検証時間を削減することができる．

## 3. 乗算器モジュールジェネレータ

本章では，ARITH に基づくデータパス設計手法を応用した乗算器モジュールジェネレータについて述べる．構築した乗算器モジュールジェネレータは入力信号幅，数系ならびに乗算アルゴリズムを仕様として指定すると，それに対応する乗算器の構造を HDL 記述として生成する．ここで，乗算アルゴリズムの機能検証に ARITH 記述と ARITH 処理系を用いることにより乗算器の機能を完全に保証することが可能になる．以下では，提案する乗算器モジュールジェネレータの構成を示す．次に，得られる多様な乗算器の性能を示す．

### 3.1 乗算器モジュールジェネレータの構成

乗算器モジュールジェネレータでは，全ての乗算器は図 3(a) のように部分積生成器（PPG）・部分積加算器（PPA）・最終段加算器（FSA）から構成される．まず，PPG において 2 つの入力信号を桁ごとに乗算し，複数の部分積を生成する．次に，PPA で複数の部分積を桁上げ保存加算器を用いて 2 つに削減する．最後に FSA を用いてこの 2 つ部分積を加算する．この FSA の出力が乗算器の出力となる．入力信号の数系が 2 の補数の場合には，上記の構造に加えて，2 の補数を符号無し 2 進数に変換する構造が PPG の内部に，その変換に伴い生じたバイアス成分をキャンセルするための構造が FSA の後段にそれぞれ加えられる．

構築した乗算器モジュールジェネレータの構成を図 6 に示す．乗算器モジュールジェネレータは (i)ARITH/HDL コードジェネレータと (ii)ARITH 処理系，(iii)BDD 等価性チェッカーによって構成される．

- ARITH/HDL コードジェネレータ

与えられた設計仕様に従い ARITH 記述と HDL 記述を生成する．現在のバージョンでは，検証を効率よく行うために，FSA の HDL 記述を生成し，PPG，PPA および全体の構造の ARITH 記述を生成する．FSA のような 2 入力加算器は，入

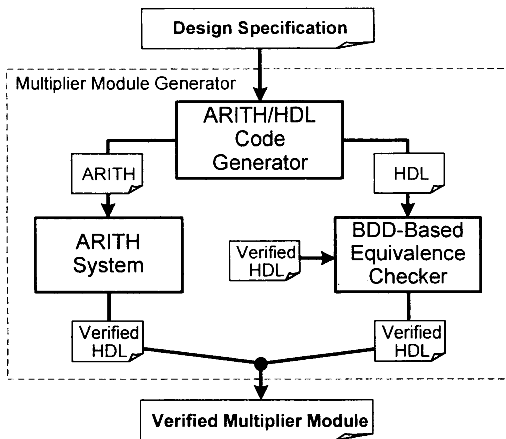



図 6 乗算器モジュールジェネレータの構成

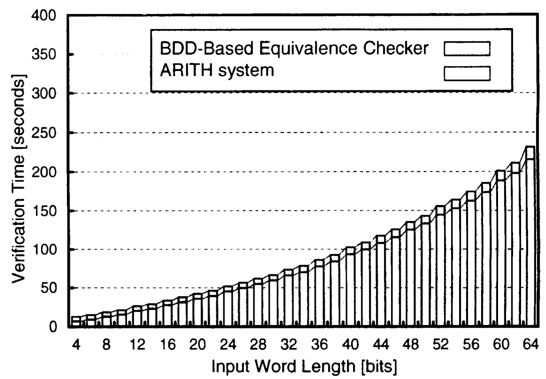



図 7 乗算器モジュールジェネレータにおける検証時間
設計仕様：数系は 2 の補数, PPG は Radix-4 Modified Booth, PPA は (4;2) Compressor Tree, FSA は Brent-Kung Adder
測定環境：SUN Blade 2000 (UltraSPARC 900MHz, メモリ 2GB)

力信号幅 $n$ に対して $O(n)$ の大きさで BDD [5] を構築可能なため，ARITH による検証よりも BDD の等価性判定を用いる方が効率よく検証できる．ここで ARITH 記述のみを用いても全ての乗算アルゴリズムが記述・検証可能であることに注意されたい．

- ARITH 処理系

生成された ARITH 記述を数式評価とレンジ評価によって形式的に検証し，検証された ARITH 記述と等価な HDL 記述を出力する．

- BDD 等価性チェッカー

生成された HDL 記述（FSA）の論理機能を BDD の等価性判定によって検証する．参照する論理機能には，算術式でシンプルに記述することが可能な Ripple Carry Adder の ARITH 記述を ARITH 処理系によってあらかじめ検証したものを用いる．

| Number System | Partial Product Generator |
|---|---|
| Unsigned Binary<br>Two's Complement | Non-Booth<br>Radix-4 Modified Booth |

| Partial Product Accumulator | Final Stage Adder |
|---|---|
| Array<br>Wallace Tree<br>Dadda Tree<br>(4;2) Compressor Tree<br>(7,3) Counter Tree<br>Overturned-Stairs Tree<br>Balanced-Delay Tree | Ripple Carry Adder<br>Carry Lookahead Adder<br>Ripple-Block CLA<br>Block CLA<br>Brent-Kung Adder<br>Kogge-Stone Adder<br>Han-Carlson Adder<br>Carry Select Adder<br>Conditional Sum Adder |

図 8 生成可能な乗算アルゴリズム

最後に，ARITH 処理系と BDD 等価性チェッカーから出力された HDL を合わせることで，完全に機能の保証された並列乗算器の HDL コードが得られる．

乗算器モジュールジェネレータの内部で行われる検証の効率を評価するために，乗算器の入力信号幅を 4 ビットから 64 ビットまで 2 ビット毎に変更し，各検証に要する時間を測定した．その結果を図 7 に示す．乗算器モジュールジェネレータでは 64 ビットの乗算器でも数分で検証を終えることが確認できる．一方，BDD 等価性判定を乗算器全体に適用することを試みた場合，14 ビット付近で BDD の大きさが爆発してしまうため，それ以上大きな乗算器を検証することは不可能であった．

### 3.2 生成された乗算器の性能

図 8 に現在の乗算器モジュールジェネレータで生成可能な算術アルゴリズムと数系の一覧を示す．乗算器モジュールジェネレータを利用する際は，乗算器の入力信号幅に加えて，このリストから各部分のアルゴリズムを設計仕様として選択する．図 8 に示したアルゴリズムの組合せにより，252 種類の並列乗算器を生成することができる．

図 9 は，32 ビット符号無し 2 進数を入力とする全ての乗算器の性能をプロットしたものである．横軸が遅延時間，縦軸が面積を示し，(a)・(b)・(c) はそれぞれ PPG・PPA・FSA の種類で分類したものである．生成された乗算器の遅延と面積は Synopsys 社の Design Compiler と Apollo によるスタンダードセルライブラリへのテクノロジマッピングと配置配線を行った結果から得られた値である．ここで，アルゴリズムの違いによる性能の違いを分かりやすくするために，Design Compiler のコンパイルオプションには `-only_design_rule` を用いた．また，使用したスタンダードセルライブラリは，VDEC から提供されている Hitachi 0.18$\mu m$ プロセス用 京都大学作成版である．図 9 より，実装されるアルゴリズムに応じて乗算器は多様な性能を示すことを確認できる．

図 9(a) において Non-Booth の方が Radix-4 modified Booth よりも全体的に遅延時間が小さくなっていることに注目されたい．一般的には Booth エンコーダを用いることで，部分積の数が減少し，遅延時間の小さい乗算器が設計できることが知られている．これとは逆の結果を図 9(a) は示している．その理由は，利用したスタンダードセルライブラリにあると考えられる．今回用いたセルライブラリには非常に遅延時間の小さい全

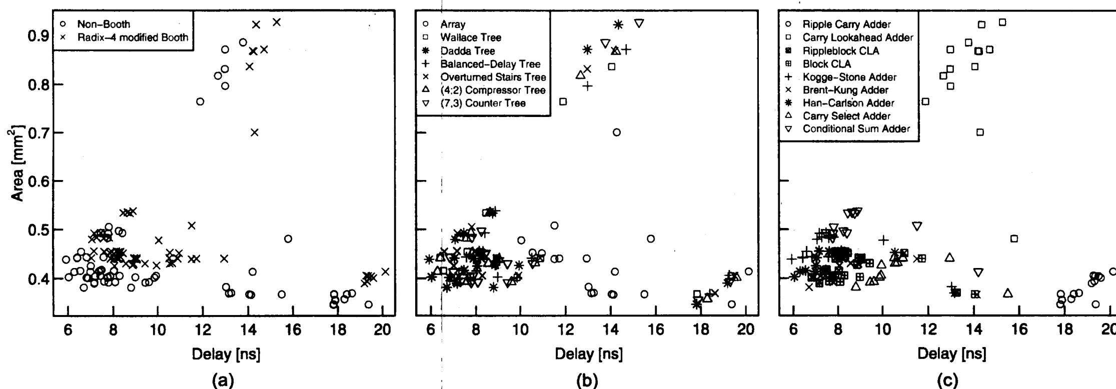



図 9　生成された 32 ビット符号無し 2 進数乗算器の性能分布
(a) PPG による分類，(b) PPA による分類，(c) FSA による分類

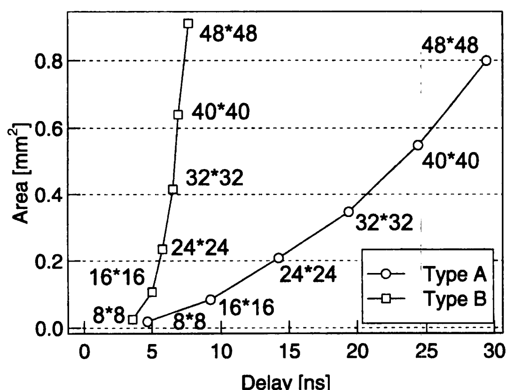



図 10　入力信号幅の変化による性能の推移
Type A: Non-Booth，Array，Ripple Carray Adder
Type B: Non-Booth，Wallace Tree，Han-Carlson Adder

加算器のセルが含まれている．そのため，多段の論理ゲートにより構成される Booth のアルゴリズムを用いた乗算器よりも，全加算器を多く用いた乗算器の方が遅延が小さく高速になったものと考えられる．

図 10 は，2 種類の乗算アルゴリズムについて，入力信号幅を変えた時の性能をプロットしたものである．Type A は PPG，PPA，FSA それぞれに Non-Booth，Array，Ripple Carry Adder を用いた乗算器，Type B は Non-Booth，Wallace Tree，Han-Carlson Adder を用いた乗算器である．Type A は，PPA，FSA ともに遅延時間 $O(n)$ のアルゴリズムを用いているのに対し，Type B は PPA，FSA ともに遅延時間 $O(\log n)$ のアルゴリズムを用いている．面積については，Type A が PPA，FSA ともに $O(n)$ のアルゴリズムを用いているのに対し，Type B は PPA が $O(n)$，FSA が $O(n \log n)$ のアルゴリズムを用いている．図 10 に示した結果はこれらのアルゴリズムの特徴を反映しており，乗算器モジュールジェネレータにアルゴリズムが正しく実装されていることがわかる．

## 4.　まとめ・今後の展望

本稿では，算術アルゴリズム記述言語 ARITH を用いたデータパス設計の概要について述べ，その応用例である ARITH に基づく乗算器モジュールジェネレータについて述べた．データパスの設計に ARITH を用いることで，機能検証の時間を大幅に削減できることを示すとともに，乗算器モジュールジェネレータによって多様な性能をもった乗算器が生成されることを示した．3 章で述べた乗算器モジュールジェネレータと，生成された乗算器の性能については，本研究グループのウェブページ（http://www.aoki.ecei.tohoku.ac.jp/arith/）にて公開している．

今後は，ARITH において除算や浮動小数点演算の取り扱いが可能となるように言語仕様の拡張を検討していく．

## 文　　献


[1] I. Koren: "Computer Arithmetic Algorithms", A K Peters (2001).

[2] B. Parhami: "Computer Arithmetic: Algorithms and Hardware Designs", Oxford University Press (2000).

[3] A. R. Omondi: "Computer Arithmetic Systems: Algorithms, Architecture and Implementations", Prentice Hall (1994).

[4] 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).

[5] R. Bryant: "Graph-based algorithms for boolean function manipulation", IEEE Trans. Computers, **C-35**, 8, pp. 677–691 (1986).