PAPER Special Section on VLSI Design and CAD Algorithms

Formal Design of Arithmetic Circuits Based on Arithmetic Description Language

Naofumi HOMMA[†], ***Member***, **Yuki WATANABE**[†], ***Nonmember***, **Takafumi AOKI**[†], ***Member***, ***and*** **Tatsuo HIGUCHI**[††], ***Fellow***

SUMMARY This paper presents a formal design of arithmetic circuits using an arithmetic description language called ARITH. The key idea in ARITH is to describe arithmetic algorithms directly with high-level mathematical objects (i.e., number representation systems and arithmetic operations/formulae). Using ARITH, we can provide formal description of arithmetic algorithms including those using unconventional number systems. In addition, the described arithmetic algorithms can be formally verified by equivalence checking with formula manipulations. The verified ARITH descriptions are easily translated into the equivalent HDL descriptions. In this paper, we also present an application of ARITH to an arithmetic module generator, which supports a variety of hardware algorithms for 2-operand adders, multi-operand adders, multipliers, constant-coefficient multipliers and multiply accumulators. The language processing system of ARITH incorporated in the generator verifies the correctness of ARITH descriptions in a formal method. As a result, we can obtain highly-reliable arithmetic modules whose functions are completely verified at the algorithm level.

key words: *datapaths, formal design, arithmetic circuits, hardware algorithms, hardware description language, module generator*

1. Introduction

Arithmetic circuits are of major importance in today's computing and signal processing systems. Numerous algorithms for arithmetic computation have been developed and implemented since the early days of digital computers, and newer ones are still being proposed [1], [2]. In addition to the standard binary arithmetic algorithms, we can introduce non-binary arithmetic algorithms for enhancing the performance of arithmetic circuits. These include high-radix number systems, redundant number systems and other dedicated data structures designed for specific applications [3].

Most of the arithmetic algorithms are devised by researchers who had trained in a particular way to understand the basic arithmetic fundamentals. Currently, we do not have a unified framework for manipulating arithmetic circuit structures in a systematic way. The conventional Hardware Description Languages (HDLs) cannot handle high-level arithmetic data structures, arithmetic operations and formulae with various number systems (including user-defined unconventional number systems). Even the state-of-the-art design environment can provide only limited capability to create arithmetic circuit structures. This sometimes requires us to describe structural details of the arithmetic circuits at the lowest level of abstraction.

Addressing this problem, this paper presents a new approach to designing arithmetic circuits using an arithmetic description language called "ARITH" (see [4] for earlier discussions on this topic). The key idea in ARITH is to describe arithmetic algorithms with integer equations. The underlying observation here is that most hardware algorithms for addition, subtraction and multiplication can be naturally represented by a set of mathematical objects such as integer equations. The use of ARITH makes possible (i) formal description of arithmetic algorithms including those using unconventional number systems, (ii) formal verification of described arithmetic algorithms, and (iii) translation of arithmetic algorithms to the equivalent HDL descriptions.

Early researches on arithmetic algorithm description are primarily based on the standard binary number system. Reference [5], for example, presents a hardware description language, called ACV language, to describe arithmetic circuits in a hierarchical fashion, and is closely related to our approach. The reported language, however, is designed so as to build multiplicative Binary Moment Diagrams (*BMDs). It seems difficult to handle arbitrary number systems. On the other hand, ARITH is dedicated for describing arithmetic algorithms based on various positional number systems.

This paper also presents an application of ARITH to an arithmetic module generator. By using ARITH, we can develop an arithmetic algorithm library containing a wide variety of arithmetic algorithms in a unified manner. The ARITH-based library is used for the generation. The language processing system of ARITH, which is incorporated in the generator, formally verifies the correctness of ARITH descriptions using formula manipulations as well as the conventional techniques. The verified ARITH descriptions are finally translated into the equivalent HDL descriptions.

Manuscript received March 10, 2006.
Manuscript revised June 13, 2006.
Final manuscript received August 1, 2006.

[†]The authors are with the Department of Computer and Mathematical Sciences, Graduate School of Information Sciences, Tohoku University, Sendai 980-8579 Japan.

[††]The author is with the Department of Electronics, Tohoku Institute of Technology, Sendai 982-8577 Japan.

The proposed generator supports various types of hardware algorithms for 2-operand adders, multi-operand adders, parallel multipliers, constant-coefficient multipliers and multiply accumulators. Some of the hardware algorithms are based on unconventional number systems such as Signed-Digit (SD) number system [6]. For example, we have Booth encoder with radix-4 SD number system, redundant binary addition tree, and constant-coefficient multipliers with Signed-Weight number system [7]. Even if such unconventional number systems are used in the hardware algorithms, we can obtain highly-reliable arithmetic modules whose functions are completely verified at the algorithm level. In this paper, we demonstrate the capability of the ARITH-based generator through some experimental results.

2. Arithmetic Description Language: ARITH

2.1 Formal description of arithmetic algorithms

ARITH is a dedicated language for describing computer arithmetic algorithms based on weighted number systems. In ARITH, we can employ high-level mathematical objects (i.e., number representation systems and arithmetic operations/formulae) for describing arithmetic algorithms. We assume here that arithmetic circuits implement arithmetic functions which should be dealt with in the (integer) arithmetic domain rather than the (Boolean) logic domain.

ARITH description consists of two blocks: `typedef` blocks and `module` blocks. The `typedef` block is used to define arithmetic data types, i.e., the number representation systems. The `module` block includes functions of arithmetic algorithms and internal structures. Every integer variable in the `module` block is associated with a weighted number system defined by the `typedef` block.

We first describe the `typedef` block. The weighted number system [1] defined in the `typedef` block is associated with the tuple $\langle \boldsymbol{W}, \boldsymbol{D} \rangle$, where $\boldsymbol{W}$ is the *weight vector* and $\boldsymbol{D}$ is the *digit set vector*, respectively. More precisely, $\boldsymbol{W}$ and $\boldsymbol{D}$ are defined as follows:

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

where h is the most significant digit, and l ($\leq h$) is the least significant digit. Each digit set is represented as an *arithmetic interval*. An *arithmetic interval* is defined as a set of integers:

$$\begin{aligned} &[min, max, step] \\ &\triangleq \{u \in \mathbf{Z} \mid (min \leq u) \ \wedge \ (u \leq max) \\ &\qquad \wedge \ \exists j (j \in \mathbf{Z}_{0+} \ \wedge \ u = min + step \cdot j)\}, \end{aligned} \quad (2)$$

where $\mathbf{Z}$ is the set of integers, $\mathbf{Z}_{0+}$ is the set of non-negative integers, and the integer constants min, max, and $step$ satisfy $min \leq max$ and $step \geq 0$.

```
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
```

Fig. 1 Radix-2 signed-digit number system with digit set {-1,0,1}.

Using the above notation, we can define various number systems including unconventional number systems. For example, unsigned binary number system (UB), two's complement representation (TC), radix-2 signed-digit number system with digit set {-1,0,1} (SD2,1) and radix-4 signed-digit number system with digit set {-2,-1,0,1,2} (SD4,2) are given as follows:

- UB

$$\begin{array}{rcl} \boldsymbol{W_{UB}} & \triangleq & \langle 2^h, 2^{h-1}, \cdots, 2^i, \cdots, 2^{l+1}, 2^l \rangle, \\ \boldsymbol{D_{UB}} & \triangleq & \langle \{0,1\}, \cdots, \{0,1\} \rangle. \end{array} \quad (3)$$

- TC

$$\begin{array}{rcl} \boldsymbol{W_{TC}} & \triangleq & \langle -2^h, 2^{h-1}, \cdots, 2^i, \cdots, 2^{l+1}, 2^l \rangle, \\ \boldsymbol{D_{TC}} & \triangleq & \langle \{0,1\}, \cdots, \{0,1\} \rangle. \end{array} \quad (4)$$

- SD2,1

$$\begin{array}{rcl} \boldsymbol{W_{SD2,1}} & \triangleq & \langle 2^h, 2^{h-1}, \cdots, 2^i, \cdots, 2^{l+1}, 2^l \rangle, \\ \boldsymbol{D_{SD2,1}} & \triangleq & \langle \{-1,0,1\}, \cdots, \{-1,0,1\} \rangle. \end{array} \quad (5)$$

- SD4,2

$$\begin{array}{rcl} \boldsymbol{W_{SD4,2}} & \triangleq & \langle 4^h, 4^{h-1}, \cdots, 4^i, \cdots, 4^{l+1}, 4^l \rangle, \\ \boldsymbol{D_{SD4,2}} & \triangleq & \langle \{-2,-1,0,1,2\}, \cdots, \\ & & \qquad \{-2,-1,0,1,2\} \rangle. \end{array} \quad (6)$$

Figure 1 shows a `typedef` block for SD2,1, where
`SD2_1.high`: the most significant digit,
`SD2_1.low`: the least significant digit,
`SD2_1{i}.weight`: the weight at the ith digit set,
`SD2_1{i}.min`: the min. integer at the ith digit set,
`SD2_1{i}.max`: the max. integer at the ith digit set,
`SD2_1{i}.step`: the step at the ith digit set.
At lines 2-7, we define SD2,1 from `SD2_1.low` digit to `SD2_1.high` digit.

On the other hand, the `module` block is used to describe an arithmetic algorithm in a hierarchical fashion. As an example, let us consider an 8-bit SD2,1 multiplier as shown in Fig. 2. Each component (e.g., "Accumulator" in (a), "PPG1" in (b), and each solid square in (c)) can be described as a module. The modules in Figs. 2 (a) and (b) correspond to the shaded parts in Figs. 2

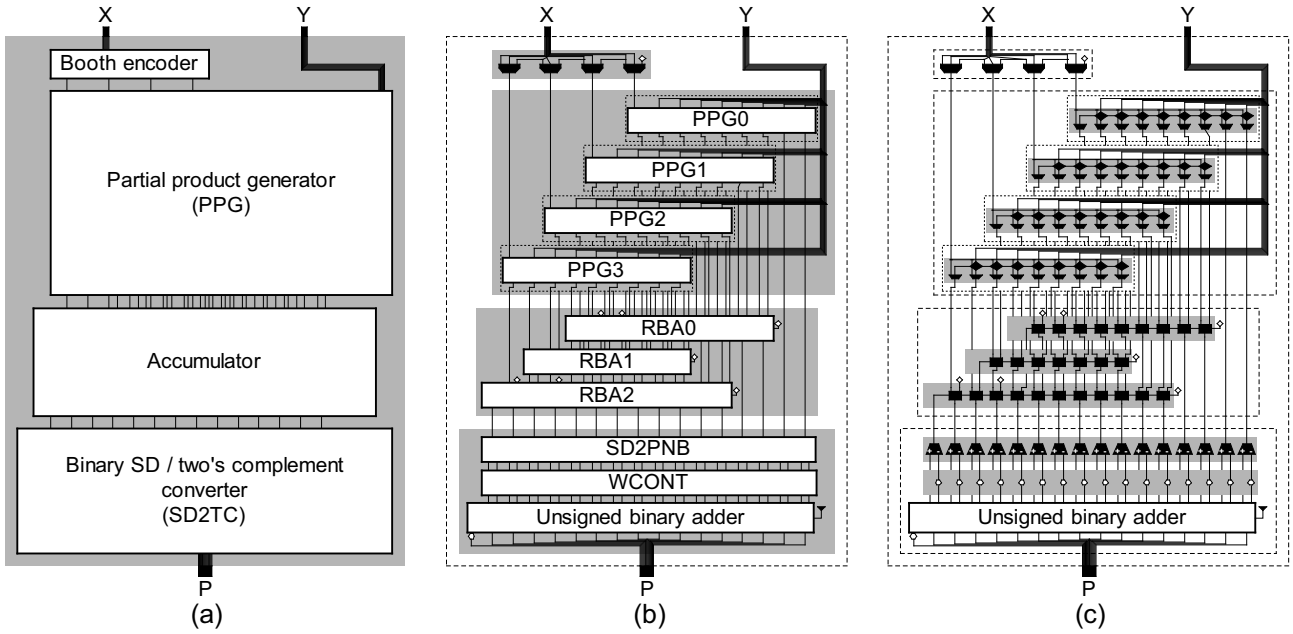


Fig. 2 8-bit SD2,1 multiplier at various levels of abstraction.

SD_MULT (P, X, Y)
function:
  assertion: $P = X \times Y$
  attributes: $P.att = \langle\langle -2^{16}, 2^{15}, \cdots, 2^1, 2^0\rangle, \langle\{0,1\}, \{0,1\}, \cdots, \{0,1\}, \{0,1\}\rangle\rangle$
  $X.att = \langle\langle -2^7, 2^6, \cdots, 2^1, 2^0\rangle, \langle\{0,1\}, \{0,1\}, \cdots, \{0,1\}, \{0,1\}\rangle\rangle$
  $Y.att = \langle\langle -2^7, 2^6, \cdots, 2^1, 2^0\rangle, \langle\{0,1\}, \{0,1\}, \cdots, \{0,1\}, \{0,1\}\rangle\rangle$
structure:
  components: BOOTH_ENCODER (B, X)
  PPG (PP, B, Y)
  ACCUMULATOR (F, PP)
  SD2TC (P, F)
  attributes: $B.att = \langle\langle 4^3, 4^2, 4^1, 4^0\rangle, \langle\{-2,-1,0,1,2\}, \{-2,-1,0,1,2\}, \{-2,-1,0,1,2\}, \{-2,-1,0,1,2\}\rangle\rangle$
  $PP[0].att = \langle\langle 2^8, 2^7, \cdots, 2^1, 2^0\rangle, \langle\{-1,0,1\}, \{-1,0,1\}, \cdots, \{-1,0,1\}, \{-1,0,1\}\rangle\rangle$
  $PP[1].att = \langle\langle 2^{10}, 2^9, \cdots, 2^3, 2^2\rangle, \langle\{-1,0,1\}, \{-1,0,1\}, \cdots, \{-1,0,1\}, \{-1,0,1\}\rangle\rangle$
  $PP[2].att = \langle\langle 2^{12}, 2^{11}, \cdots, 2^5, 2^4\rangle, \langle\{-1,0,1\}, \{-1,0,1\}, \cdots, \{-1,0,1\}, \{-1,0,1\}\rangle\rangle$
  $PP[3].att = \langle\langle 2^{14}, 2^{13}, \cdots, 2^7, 2^6\rangle, \langle\{-1,0,1\}, \{-1,0,1\}, \cdots, \{-1,0,1\}, \{-1,0,1\}\rangle\rangle$
  $F.att = \langle\langle 2^{15}, 2^{14}, \cdots, 2^1, 2^0\rangle, \langle\{-1,0,1\}, \{-1,0,1\}, \cdots, \{-1,0,1\}, \{-1,0,1\}\rangle\rangle$

Fig. 3 Formal description of 8-bit SD2,1 multiplier in Fig. 2 (a).

(b) and (c), respectively. The internal structure of each module is described by using the sub-modules on the corresponding shaded part.

Figure 3 shows a formal description of the SD2,1 multiplier at the top of the hierarchy (Fig. 2 (a)). The formal description consists of two parts: "function" and "structure". The function part describes functional assertions with integer equations. The structure part describes components that make up the function. In both parts, the attributes of integer variables (e.g., *P.att*), are specified with weighted number systems. The above description forms the basis of the `module` block.

Figure 4 represents a `module` block corresponding to Fig. 3. The `module` block includes declarative statements of I/O interface (at lines 2-8), functional assertion (at line 9) and structural description (at lines 10-26). Basic signals used in the ARITH description (i.e., `X`, `Y` and `P`) are "integer signals." Every integer signal, say `X`, consists of "digit signals" `X{7}`, `X{6}`, `X{5}`, `X{4}`, `X{3}`, `X{2}`, `X{1}`, and `X{0}`. The integer signal and digit signals are associated with a specific number system defined by the `typedef` block.

Let us explain the `SD_MULT` module in detail.

- Statements of I/O interface (at lines 2-8):
  The input/output signals `X`, `Y`, and `P` are declared with `TC` number representation at lines 2-3. The corresponding digit ranges are determined by using `high` and `low` at lines 4-8.
- Functional assertion (at line 9):
  The function is described as an integer equation `P = X * Y`, where the left-hand side indicates output, and the right-hand side indicates input.
- Structural description (at lines 10-26):
  The internal structure is described by using sub-modules and internal signals. The internal signals

```
 1:  module SD_MULT(P, X, Y);
 2:    output TC P;
 3:    input TC X, Y;
 4:    constraint begin
 5:      P.high = 16; P.low = 0;
 6:      X.high = 7; X.low = 0;
 7:      Y.high = 7; Y.low = 0;
 8:    end
 9:    assertion P = X * Y;
10:    structure begin
11:      wire SD4_2 B;
12:      wire SD2_1 PP[];
13:      wire SD2_1 F;
14:      constraint begin
15:        B.high = 3; B.low = 0;
16:        PP.high = 3; PP.low = 0;
17:        for (i, 0, 3) begin
18:          PP[i].high=i*2+8; PP[i].low=i*2;
19:        end
20:        F.high = 15; F.low = 0;
21:      end
22:      BOOTH_ENCODER U0 (B,X);
23:      PPG           U1 (PP, B, Y);
24:      ACCUMULATOR   U2 (F,PP);
25:      SD2TC         U3 (P,F);
26:    end
27:  endmodule
```

Fig. 4 Top module of an 8-bit SD2,1 multiplier.

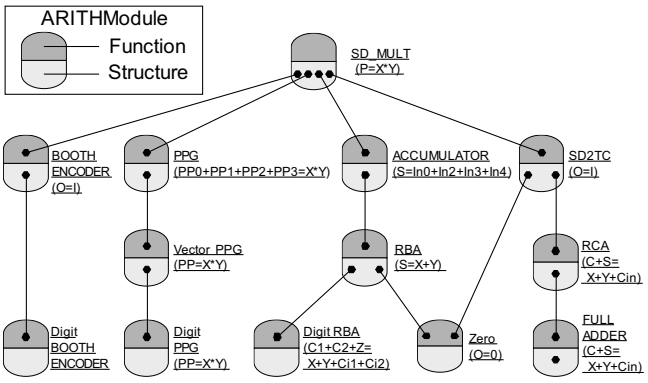


Fig. 5 Hierarchy diagram for the SD2,1 multiplier as shown in Fig. 2.

based on `SD2_1` and `SD4_2` number systems are declared similarly to input/output signals at lines 11-21, where `PP[]` represents that `PP` is an array of integer signals. At line 16, it is determined that `PP` contains 4 integer signals. The `SD_MULT` module assumes the use of sub-modules `BOOTH_ENCODER`, `PPG`, `ACCUMULATOR`, and `SD2TC` at lines 22-25.

As shown in this example, we describe an arithmetic algorithm in a hierarchical fashion. Each module is composed of sub-modules that can be described independently in ARITH. This is based on the principle that an arithmetic circuit can be divided into simpler sub-circuits, which themselves compute arithmetic functions. Figure 5 shows a hierarchy diagram for the SD2,1 multiplier.

```
 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 S#L = X#L^Y#L^Z#L;
14:      assign C#L = X#L&Y#L|(X#L|Y#L)&Z#L;
15:    end
16:  endmodule
```

Fig. 6 Module including logical expressions.

We also describe arbitrary logical expressions in ARITH. Figure 6 shows an example of modules including logical expressions. An encoding type `<enc_type>` is declared for input/output signals at lines 2-3. The encoding type is defined in the `typedef` block. In this example, we assume that the input/output signals are given by a single bit L. Therefore, the internal structure is described as lines 13-14, where `X#L`, `Y#L`, `Z#L`, `S#L` and `C#L` indicate the logic signals of `X`, `Y`, `Z`, `S` and `C`, respectively.

The ARITH grammar is given by the denotational semantics based on meta-notation [8]. Denotational definitions are available for the automatic construction of the compilers. Figures 7 and 8 are examples of the syntactic domain and semantic domain, respectively. Both domains are formally defined by using the abstract grammar, where $A \stackrel{\triangle}{=} B$, $A*$, $A|B$, $a : A; b : B$ denote a production rule, a list, a choice and an aggregate, respectively. The semantics of ARITH description can be represented with mathematical objects such as sets and functions. The full abstract grammar is given on our website [9].

2.2 Formal verification in ARITH

Functional verification of arithmetic circuits remains to be difficult although recent verification technology can handle designs with millions of gates. In general, it requires time-consuming circuit simulation due to the large number of input/output variables. It is almost impossible to simulate 100% functionality if the number of input variables increases. The formal verification techniques have gained large attention to ensure 100% functional correctness of a design instead of simulating

```
File   △=  Block*
Block  △=  Module | Typedef
Module △=  moduleId :  ModuleId;
               argumentList :  ArgumentList;
               moduleBody :  ModuleBody
...
```

Fig. 7 Definition of syntactic domain.

```
TopData   △=  moduleDataMap :  ModuleDataMap ;
                  typeDataMap :  TypeDataMap
ModuleDataMap  △=  ModuleDataElement*
ModuleDataElement  △=
                  moduleName :  ModuleName ;
                  moduleData :  ModuleData
...
```

Fig. 8 Definition of semantic domain.

some vectors.

The ARITH description can be formally verified by the language processing system of ARITH (ARITH code verifier). It is known that word-level DDs or *BMDs [10], [5] are useful for the formal verification of arithmetic circuits. We can apply the conventional verification techniques to ARITH descriptions. In addition, we have a definite possibility for verifying ARITH descriptions with formula manipulations. The ARITH code verifier can perform the equivalence checking with formula manipulations as well as the conventional techniques.

In the following, we describe the equivalence checking with formula manipulations. The basic idea is to check for every module (i) whether its structural description matches its functional assertion, and (ii) whether hardware implementation is possible under the range constraints of input/output signals. The proposed verification method consists of "formula evaluation" and "range evaluation" corresponding to the above two tasks, respectively. We assume that ARITH description holds correct arithmetic circuit structures if and only if both evaluations return true.

Let us explain these evaluations using `SD_MULT` in Fig. 4.

- Formula evaluation
  The formula evaluation returns "true" if the integer equations of internal structures are equivalent to those of functional assertion. The formula evaluation procedure is as follows:

  1. Obtain the integer equations representing the relationship between integer signals and their digit signals. Thus, we have

$$\begin{cases} P = P\{16\} + P\{15\} + \cdots + P\{1\} + P\{0\}, \\ X = X\{7\} + X\{6\} + \cdots + X\{1\} + X\{0\}, \\ Y = Y\{7\} + Y\{6\} + \cdots + Y\{1\} + Y\{0\}, \\ B = B\{3\} + B\{2\} + B\{1\} + B\{0\}, \\ PP = PP[3] + PP[2] + PP[1] + PP[0], \\ F = F\{15\} + F\{14\} + \cdots + F\{1\} + F\{0\}, \end{cases} \tag{7}$$

     where P indicates the output signal; X and Y indicate the input signals; B, PP and F indicate the internal signals.
  2. Extract the set of functional assertions from submodules and rename the integer signals according to the given structural description. Thus, we have

$$\begin{cases} B = X, \\ PP[3] + PP[2] + PP[1] + PP[0] = B * Y, \\ F = PP[3] + PP[2] + PP[1] + PP[0], \\ P = F. \end{cases} \tag{8}$$

  3. Solve the system of integer equations obtained from the above two steps for the input/output signals. We employ Gaussian Elimination method and Groebner-bases method [11] for solving the system of linear and non-linear equations, respectively. Thus, we have

$$P = X * Y. \tag{9}$$

  If the obtained solution is equal to the given functional assertion (i.e., $P = X * Y$), the formula evaluation returns "true".

- Range evaluation
  The range evaluation returns "true" if the range constraints of input/output signals are compatible with the functional assertion of the given module. The range constraints are examined on arithmetic intervals. The range evaluation procedure is as follows:

  1. Calculate the arithmetic interval of left-hand side of the functional assertion. In Fig. 4, the arithmetic interval of `P` (i.e., output) is evaluated as $[-65536, 65535, 1]$.
  2. Calculate the arithmetic interval of right-hand side of the functional assertion. In Fig. 4, the arithmetic interval of `X*Y` (i.e., input) is evaluated as $[-16256, 16384, 1]$.

  If the output arithmetic interval subsumes the input arithmetic interval, the range evaluation returns "true". This means that the given module provides sufficient output dynamic range in order to cover the input dynamic range.

2.3 Translating ARITH codes to HDL codes

ARITH descriptions can be converted into the equivalent HDL (VHDL or Verilog HDL) descriptions. This is

```
 1:  module SD_MULT (P, X, Y);
 2:     output [16:0] P;
 3:     input [7:0]   X, Y;
 4:     wire [3:0]   Bs, Bd1, Bd0;
 5:     wire [8:0]   PP0p, PP0n;
 6:     wire [10:2]   PP1p, PP1n;
 7:     wire [12:4]   PP2p, PP2n;
 8:     wire [14:6]   PP3p, PP3n;
 9:     wire [15:0]   Fp, Fn;
10:     BOOTH_ENCODE U0(Bs,Bd1,Bd0,X);
11:     PPG U1(PP0p,PP0n,PP1p,PP1n,PP2p,PP2n,
                PP3p,PP3n,Bs,Bd1,Bd0,Y);
12:     ACCUMULATE U2(Fp,Fn,PP0p,PP0n,PP1p,PP1n,
                      PP2p,PP2n,PP3p,PP3n);
13:     SD2TC U3(P,Fp,Fn);
14:  endmodule
```

Fig. 9 Verilog HDL code corresponding to Fig. 4.

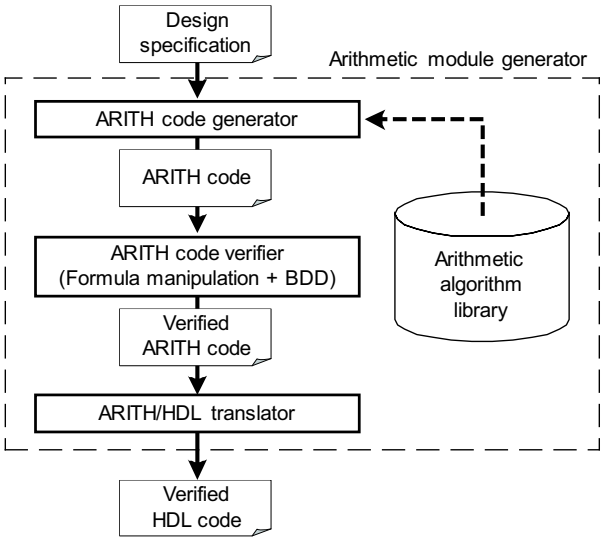


Fig. 10 AMG system flow.

just a syntactical conversion without intelligence. Figure 9 shows a Verilog HDL code corresponding to the ARITH code in Fig. 4. The integer variables are encoded into binary variables. For example, an integer variable `B` with SD4,2 (at line 11 in Fig.4) is encoded into three binary variables `Bs`, `Bd1` and `Bd0` (at line 4 in Fig.9). Note that The encoding method can be given depending on the target technologies. We confirm here that ARITH description has a higher level of readability compared with the conventional HDL description.

3. Arithmetic module generator based on ARITH

This section describes an application of ARITH to an arithmetic module generator (AMG). We employ ARITH for describing arithmetic algorithms including those using unconventional number systems in a unified manner. The product specifications considered here are 2-operand addition, multi-operand addition, multiplication, constant-coefficient multiplication, and multiply accumulation. The major advantage of the generated modules over the conventional IPs is its capability to provide a formally verified function even if the hardware algorithm contains unconventional number systems.

3.1 System framework

Figure 10 is a block diagram of AMG, which consists of (i) ARITH code generator, (ii) ARITH code verifier, and (iii) ARITH/HDL translator as follows:

- ARITH code generator:
  Generates ARITH codes according to the design specification given by designers. In the generation, parameterized algorithms are retrieved from the arithmetic algorithm library.
- ARITH code verifier:
  Verifies the generated ARITH codes using the equivalence checking with formula manipulations. The BDD-based equivalence checking [12] is performed only for 2-operand adders.
- ARITH/HDL translator:
  Translates the verified ARITH codes into the equivalent HDL codes. This can be done simply by the one-to-one mapping.

As a result, AMG obtains the HDL codes verified formally at the algorithm level. The generated ARITH codes are registered into the arithmetic algorithm library after every successful verification, and retrieved them when the same specification is requested.

3.2 Hardware algorithms

AMG supports various hardware algorithms for 2-operand adders and multi-operand adders. These hardware algorithms are also used to generate multipliers, constant-coefficient multipliers and multiply accumulators. The operand length in AMG is basically covered from 4 bits to 64 bits. In the following, we briefly describe the hardware algorithms that can be handled by AMG (see our AMG website [13] and [1],[2] for more details).

3.2.1 Two-operand addition algorithms

AMG has 11 types of 2-operand addition algorithms. Table 1 classifies all the algorithms into 5 classes: Ripple carry, Carry lookahead, Parallel prefix, Carry select, and Carry skip.

3.2.2 Multi-operand addition algorithms

AMG has 8 types of multi-input 2-output addition al-

Table 1 Two-operand addition algorithms

Class	Algorithm
Ripple carry	Ripple carry adder
Carry lookahead	Carry lookahead adder (CLA) Ripple-block CLA Block CLA
Parallel prefix	Kogge-Stone adder Brent-Kung adder Han-Carlson adder
Carry select	Carry select adder Conditional sum adder
Carry skip	Fixed-block-size carry skip adder Variable-block-size carry skip adder

Table 2 Multi-operand addition algorithms

Component	Level of design optimization	
	Word-level design	Bit-level optimized design
(3,2) counter	Array Wallace tree Overturned-stairs tree Balanced delay tree	Dadda tree
(4;2) compressor	(4;2) compressor tree	
(7,3) counter		(7,3) counter tree
RB adder	RB addition tree	

Table 3 Verification time of AMG

Arithmetic module	Verification time [s]		
	16 bits	32 bits	64 bits
Two-operand adder	0.75	1.82	6.09
Multi-operand adder	65.40	84.00	111.10
Multiplier	26.50	53.48	127.70
Multiply accumulator	25.62	54.30	158.90

gorithms, where the number of operands is from 4 to 64. Table 2 shows hardware algorithms that can be handled by the generator, where the bit-level optimized design indicates that the matrix of operands is reorganized to minimize the number of basic components. These basic components include (3,2) counter, (4;2) compressor, (7,3) counter, and redundant-binary (RB) adder.

3.2.3 Multiplication algorithms

AMG provides parallel multipliers consisting of Partial Product Generator (PPG), Partial Product Accumulator (PPA), and Final Stage Adder (FSA). The PPG stage first generates partial products from the multiplicand and multiplier in parallel. The PPA stage then performs multi-operand addition for all the generated partial products and produces their sum in carry-save form. Finally, the carry-save form is converted to the corresponding binary output at FSA.

Figure 11 shows hardware algorithms for PPG, PPA, and FSA. Note here that PPAs and FSAs correspond to multi-operand adders and 2-operand adders. In addition, we have 2 types of PPGs in AMG. In total, AMG supports 352 types of hardware algorithms for parallel multiplication.

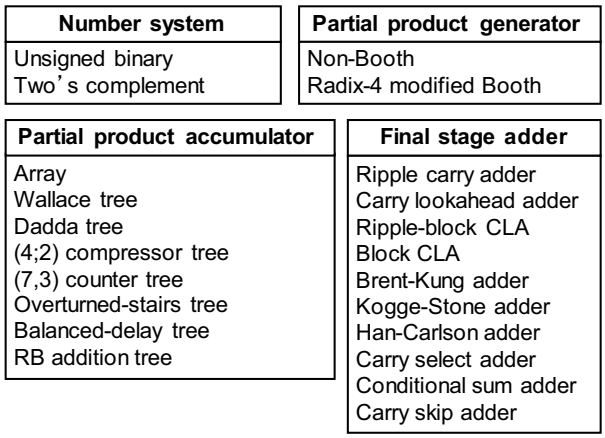


Fig. 11 Hardware algorithms for parallel multipliers and multiply accumulators.

3.2.4 Constant-coefficient multiplication algorithms

AMG provides constant-coefficient multipliers in the form: $p = Rx$, where R is an integer coefficient, and x and p are the integer input and output. The hardware algorithms for constant-coefficient multiplication are based on multi-input 1-output addition algorithms (i.e., combinations of PPAs and FSAs). There are many possible choices for the multiplier structure for a specific coefficient R. The complexity of multiplier structures significantly varies with the coefficient value R.

We consider here the use of special number representation called Signed-Weight (SW) number system [7], which is useful for constructing compact PPAs. At present, the combination of CSD (Canonic Signed-Digit) coefficient encoding technique [14] with the SW-based PPAs seems to provide the practical hardware implementation of fast constant-coefficient multipliers. As a result, AMG supports such hardware algorithms for constant-coefficient multiplication, where the range of R is from -2^{31} to $2^{31} - 1$.

3.2.5 Multiply accumulation algorithms

AMG provides multiply accumulators in the form: $p = \sum_{i=0}^{N} x_i \times y_i$, where x_i and y_i are the integer inputs/constants, p is the integer output, and $N(\in \{1, 2\})$ is the integer constant. The operand length is from 4 bits to 32 bits. A multiply accumulator is generated by a combination of hardware algorithms for multipliers and constant-coefficient multipliers. All the partial products from PPGs are accumulated in carry-save form by a single PPA. The carry-save form is converted to the corresponding binary output by an FSA.

3.3 Experimental designs

To evaluate the verification time of AMG, we have

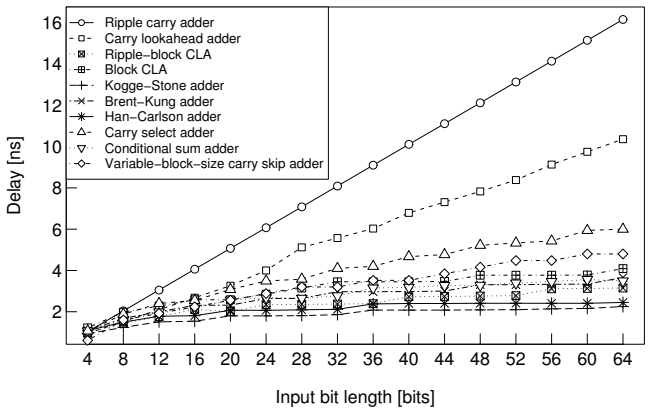


Fig. 12 Latency of two-operand adders for various operand lengths.

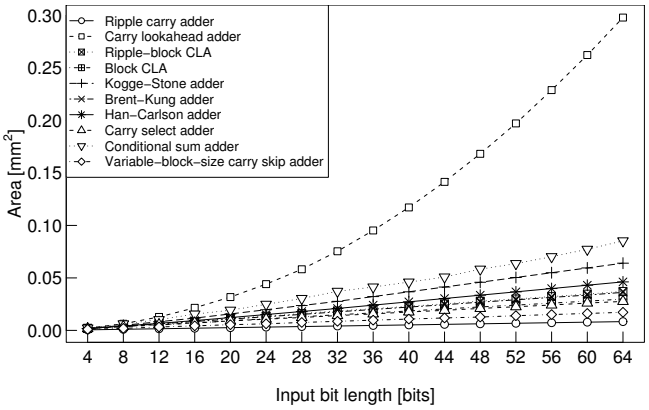


Fig. 13 Area of two-operand adders for various operand lengths.

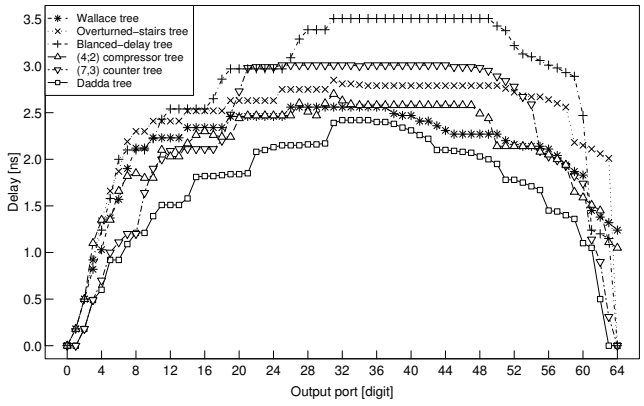


Fig. 14 Output arrival profile of multi-operand adders.

designed the four types of arithmetic modules whose operand lengths are 16, 32, and 64 bits. Table 3 illustrates the verification time of AMG on Intel Pentium 4 CPU 2.80GHz and 2GB memory, where "Two-operand adder" indicates the Kogge-Stone adder, "Multi-operand adder" indicates the 64-operand Wallace tree, "Multiplier" indicates the carry lookahead

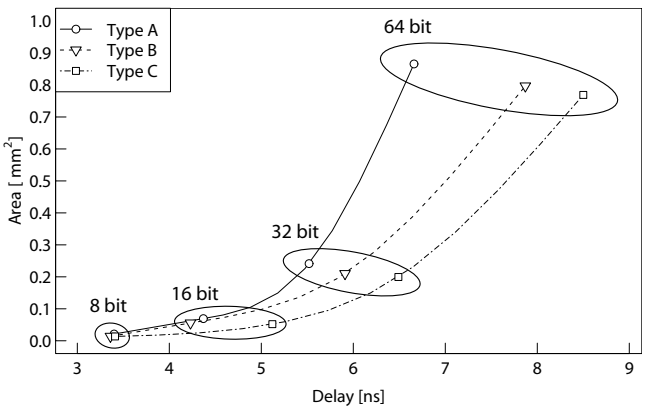


Fig. 15 Comparison of three types of unsigned multipliers.

adder and RB addition tree with radix-4 modified Booth encoder, and "Multiply accumulator" indicates the ripple carry adder and overturned-stairs tree with radix-4 modified Booth encoder. Note that the "Multiply accumulator" has a function given by $p = x_0 \times y_0 + x_1 + x_2$. The result shows that AMG performs a complete verification of the 64-bit multiply accumulator at most 160 seconds.

In the following, let us evaluate the performance of arithmetic modules generated from AMG. The modules have been synthesized using Synopsys Design Compiler with the compile option "-only_design_rule -boundary_optimization." For the synthesis, we employed the Kyoto University's standard-cell library targeted for HITACHI 0.18 μm process (Typical condition) [15], [16]. The delay/area information was calculated according to the delay/area model given by the cell library. Note that the estimated circuit delay of a standard full adder is 0.27ns.

The evaluation results are summarized as shown in Figs. 12-15. Figures 12-13 illustrate the performance of two-operand adders for various operand lengths. Figure 14 shows the output arrival profile of 32-bit 32-operand adders, where the horizontal axis indicates the output bit position, and the vertical axis indicates the circuit delay. Figure 15 compares three types of unsigned multipliers for various operand lengths, where Type A indicates the Kogge-Stone adder and Dadda tree architecture with radix-4 Booth encoding, Type B indicates the Han-Carlson adder and Balanced-delay tree architecture, and Type C indicates the Block CLA and (4;2) compressor tree architecture. The above results suggests that AMG can generate the arithmetic modules faithfully according to the design specifications.

Figure 16 shows 32×32 unsigned multipliers generated from AMG, where the horizontal axes indicate the circuit delay, and vertical axes indicate the circuit area. Table 4 shows the numerical results of arithmetic modules obtained from AMG. The types "Small" and "Fast" are the best circuits under area optimization and delay optimization, respectively. From Fig. 16, we

Table 4 Performance evaluation

Arithmetic Unit	Type	Area [μm^2]	Delay [ns]	Area×Delay [μm^2×ns]
64-bit two-operand adder	Small	8355.84	16.16	135030.37
	Fast	64035.84	2.25	144080.65
32-bit multiplier	Small	175241.06	18.12	3175368.04
	Fast	220328.69	4.74	1044357.98
32-bit constant-coefficient multiplier ($p = 299792458 \times x$)	Small	41587.01	15.34	637944.76
	Fast	81982.87	4.39	359904.79
32-bit multiply accumulator ($p = x_0 \times y_0 + x_1 + x_2$)	Small	183566.00	18.68	3429012.88
	Fast	232947.00	4.76	1108827.72

can confirm that the performance of multipliers heavily depends on hardware algorithm. Thus, AMG can produce and evaluate a variety of the hardware algorithms including those using unconventional number systems in a unified manner. Note here that we have different distributions in the case of other target technologies.

The other evaluation results can be available on our website [13] which would be helpful for prospective designers as a reference.

4. Conclusion

In this paper, we have presented a dedicated language for describing arithmetic algorithms called ARITH. The use of ARITH makes possible the formal description and verification of arithmetic algorithms including those using unconventional number representation systems. For arithmetic circuit designers, ARITH may provide a unified framework of design optimization bridging the gap between algorithm-level design and circuit-level design. The extension of ARITH to generic arithmetic circuits is being left for future study.

This paper have also proposed an arithmetic module generator based on ARITH. The proposed generator supports various hardware algorithms for 2-operand adders, multi-operand adders, multipliers, constant-coefficient multipliers and multiply accumulators. In addition, the generated modules can be completely verified in a formal method. Further investigations are being conducted to develop advanced module generators based on ARITH for DSP systems and public key cryptosystems.

The proposed ARITH-based generator would be available from our website [13] on a trial basis. Figure 17 shows the system framework for our web service. We first specify (i) target function, (ii) hardware algorithms, (iii) operand length, and (iv) number representation system for operands from the web interface. The generator then generates an arithmetic module according to the specification. The performance evaluator operates after the successful generation. Finally, the generated module and its performance data are provided through the web interface. The AMG database registers and retrieves design specifications, generated modules in HDLs, and performance data.

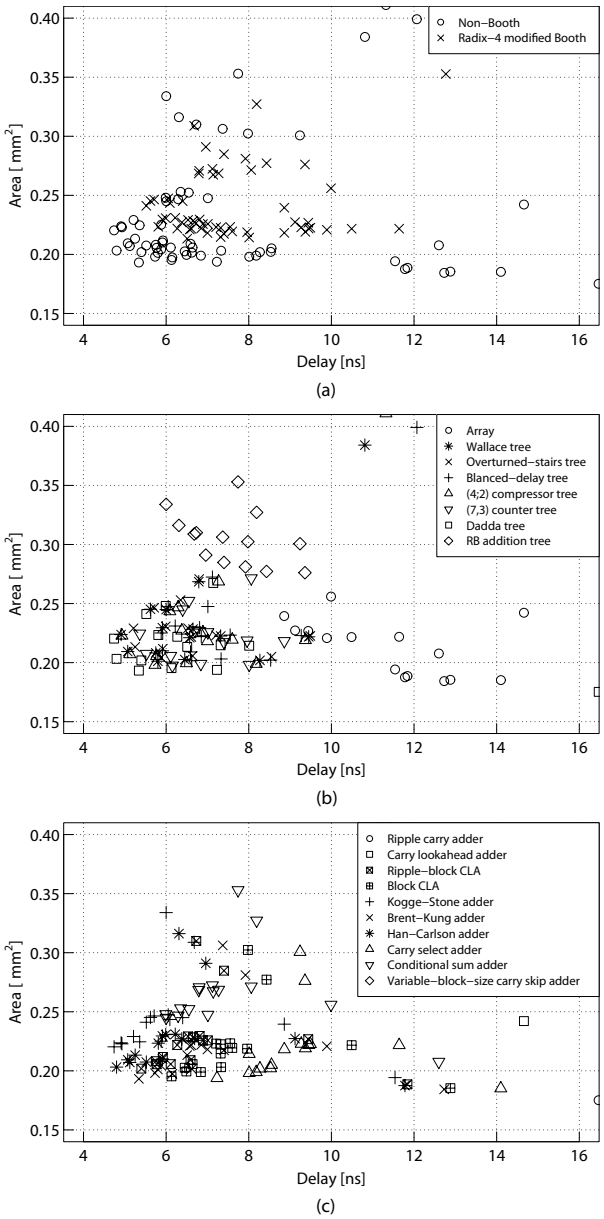


Fig. 16 Performance of 32 bit unsigned binary multipliers for HITACHI 0.18μm process : (a) PPG grouping, (b) PPA grouping, (c) FSA grouping.

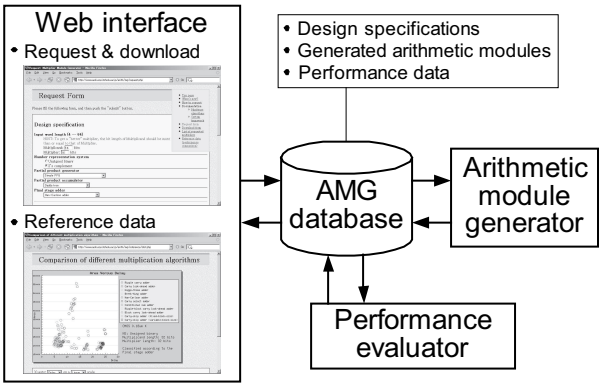
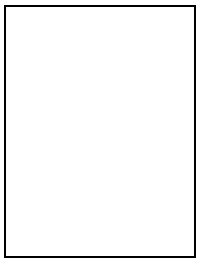


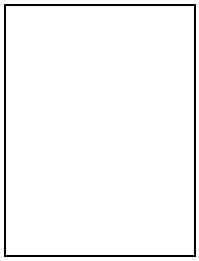
Fig. 17 System framework for web service.

References

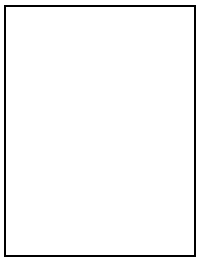
[1] I. Koren, Computer arithmetic algorithms 2nd Edition, A K Peters, 2001.
[2] B. Parhami, Computer Arithmetic: Algorithms and Hardware Designs, Oxford University Press, 2000.
[3] T. Aoki and T. Higuchi, "Beyond-binary arithmetic — Algorithms and VLSI implementations —," Interdisciplinary Information Sciences, vol.6, no.1, pp.75–98, March 2000.
[4] N. Homma, K. Ishida, T. Aoki, and T. Higuchi, "Arithmetic description language and its application to parallel multiplier design," Proc. of the Workshop on Synthesis And System Integration of Mixed Information Technologies, pp.319–326, Oct. 2004.
[5] Y.A. Chen and E.R. Bryant, "ACV: An arithmetic circuit verifier," Proc. of International Conference on Computer-Aided Design, 1996.
[6] A. Avizienis, "Signed-digit number representations for fast parallel arithmetic," IRE Trans. Electronic Computers, vol.10, pp.389 – 400, Sept. 1961.
[7] T. Aoki, Y. Sawada, and T. Higuchi, "Signed-weight arithmetic and its application to a field-programmable digital filter architecture," IEICE Trans. Electronics, vol.E82-C, no.9, pp.1687–1698, Sept. 1999.
[8] B. Meyer, Introduction to the Theory of Programming Languages, Interactive Software Engineering, 1991.
[9] Arithmetic Description Language : ARITH. http://www.aoki.ecei.tohoku.ac.jp/arith/.
[10] 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.
[11] D.A. Cox, J.B. Little, and D. O'Shea, Ideals, Varieties, and Algorithms, 2nd ed., Springer-Verlag, NY, 1996. 536 pages.
[12] R.E. Bryant, "Graph-based algorithms for boolean function manipularion," IEEE Trans. Computers, vol.C-35, no.8, pp.677 – 691, Aug. 1986.
[13] Arithmetic Module Generator based on ARITH. http://www.aoki.ecei.tohoku.ac.jp/arith/mg/.
[14] K. Hwang, Computer arithmetic: principles, architecture, and design, John Wiley & Sons, 1979.
[15] VLSI Design and Education Center (VDEC) website. http://www.vdec.u tokyo.ac.jp/English/.
[16] M. Hashimoto, K. Fujimori, and H. Onodera, "Standard cell libraries with various driving strength cells for 0.13, 0.18, and 0.35 μm technologies," Proc. of Asia and South Pacific Design Automation Conference 2003, pp.589 – 590, Jan. 2003.



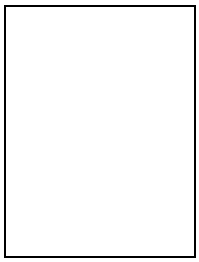
Naofumi Homma received the B.E. degree in information engineering, and the M.S. and Ph.D. degrees in information sciences from Tohoku University, Sendai, Japan, in 1997, 1999 and 2001, respectively. He is currently a Research Associate of the Graduate School of Information Sciences at Tohoku University. For 1999-2001, he was a Research Fellow of the Japan Society for the Promotion of Science. For 2002-2006, he also joined the Japan Science and Technology Agency (JST) as a researcher for the PRESTO project. His research interests include computer arithmetic, algorithms for high-performance VLSI computing, and cryptographic hardware. Dr. Homma received the IP Award at the 2005 LSI IP Design Award.



Yuki Watanabe received the B.E. degree in information engineering and the M.S. degree in information sciences from Tohoku University, Sendai, Japan, in 2004 and 2006, respectively. He is currently working towards the Ph.D. degree at Tohoku University. His research interests include computer arithmetic, algorithms for high-performance VLSI computing. Mr. Watanabe received the IP Award at the 2005 LSI IP Design Award.



Takafumi Aoki received the B.E., M.E., and D.E. degrees in electronic engineering from Tohoku University, Sendai, Japan, in 1988, 1990, and 1992, respectively. He is currently a Professor of the Graduate School of Information Sciences at Tohoku University. For 1997–1999, he also joined the PRESTO project, Japan Science and Technology Corporation (JST). His research interests include theoretical aspects of computation, VLSI computing structures for signal and image processing, multiple-valued logic, and biomolecular computing. Dr. Aoki received the Outstanding Paper Award at the 1990, 2000, 2001 and 2006 IEEE International Symposiums on Multiple-Valued Logic, the Outstanding Transactions Paper Award from the Institute of Electronics, Information and Communication Engineers (IEICE) of Japan in 1989 and 1997, the IEE Ambrose Fleming Premium Award in 1994, the IEICE Inose Award in 1997, the IEE Mountbatten Premium Award in 1999, the Best Paper Award at the 1999 IEEE International Symposium on Intelligent Signal Processing and Communication Systems, and the IP Award at the 2005 LSI IP Design Award.



Tatsuo Higuchi received the B.E., M.E., and D.E. degrees in electronic engineering from Tohoku University, Sendai, Japan, in 1962, 1964, and 1969, respectively. He is currently a Professor at Tohoku Institute of Technology and a Emeritus Professor at Tohoku University. From 1980 to 1993, he was a Professor in the Department of Electronic Engineering at Tohoku University. He was a Professor from 1994 to 2003, and was Dean from 1994 to 1998 in the Graduate School of Information Sciences at Tohoku University. His general research interests include the design of 1-D and multi-D digital filters, linear time-varying system theory, fractals and chaos in digital signal processing, VLSI computing structures for signal and image processing, multiple-valued ICs, multiwave opto-electronic ICs, and biomolecular computing. Dr. Higuchi received the Outstanding Paper Awards at the 1985, 1986, 1988, 1990, 2000, 2001 and 2006 IEEE International Symposiums on Multiple-Valued Logic, the Certificate of Appreciation in 2003 and the Long Service Award in 2004 on Multiple Valued Logic, the Outstanding Transactions Paper Award from the Society of Instrument and Control Engineers (SICE) of Japan in 1984, the Technically Excellent Award from SICE in 1986, the Outstanding Book Award from SICE in 1996, the Outstanding Transactions Paper Award from the Institute of Electronics, Information and Communication Engineers (IEICE) of Japan in 1990 and 1997, the Inose Award from IEICE in 1997, the Technically Excellent Award from the Robotics Society of Japan in 1990, the IEE Ambrose Fleming Premium Award in 1994, the Outstanding Book Award from the Japanese Society for Engineering Education in 1997, the Award for Persons of scientific and technological merits (Commendation by the minister of state for Science and Technology), the IEE Mountbatten Premium Award in 1999, the Best Paper Award at the 1999 IEEE International Symposium on Intelligent Signal Processing and Communication Systems, and the IP Award at the 2005 LSI IP Design Award. He also received the IEEE Third Millennium Medal in 2000. He is a Life Fellow of IEEE and a Fellow of IEICE and SICE.