IP/SOC 2005

Session : Specific Modeling

A Multiplier Module Generator Based on Arithmetic Description Language

Naofumi Homma†, Yuki Watanabe, Kazuya Ishida, and Takafumi Aoki
Graduate School of Information Sciences
Tohoku University, Sendai 980-8579, Japan
homma@aoki.ecei.tohoku.ac.jp
†The author is also with PRESTO, JST.

Tatsuo Higuchi
Department of Electronics
Tohoku Institute of Technology
Sendai 982-8577, Japan

Abstract :

This paper presents an arithmetic module generator based on an arithmetic description language called ARITH. 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 equivalent HDL codes. We employ ARITH to develop an arithmetic algorithm library included in the generator. The developed generator supports 352 types of hardware algorithms for parallel multiplication and produces highly-reliable multiplier modules whose functions are completely verified at the algorithm level.

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 being proposed all the time [1], [2]. 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 systematic framework for manipulating arithmetic circuit structures based on 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.

Addressing this problem, we propose a new approach to designing arithmetic circuits using an arithmetic description language called "ARITH" (see [3] for earlier discussions on this topic). The key idea in ARITH is to describe arithmetic algorithms with 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 equivalent HDL codes. Examples of number systems that can be handled by ARITH include Redundant-Binary (RB) number system [4], Signed-Digit (SD) number systems [5], Generalized Signed-Digit (GSD) number systems [6], Positive-Digit (PD) number systems [7] and Binary Carry-Save number system [8].

This paper presents an application of ARITH to an arithmetic module generator. ARITH is used as a data format for the proposed generator. By using ARITH, we can develop the arithmetic algorithm library in a unified manner, and produce highly-reliable arithmetic modules whose functions are completely verified in a formal method. The language processing system of ARITH incorporated in the generator verifies the correctness of ARITH descriptions using formula manipulations as well as the conventional manipulations such as *BMDs [9] and BDDs [10]. The hybrid approach is effective especially for verifying ARITH descriptions in a short time. The generator can translate the verified ARITH description into the equivalent HDL description.

In this paper, we focus on a multiplier module generator (MMG) based on ARITH. The product specification considered here is a parallel multiplier consisting of Partial Product Generator (PPG), a partial product accumulator (PPA), and a final stage adder (FSA). Various hardware algorithms of PPG, PPA and FSA are implemented in ARITH for the arithmetic algorithm library. The developed MMG system [**?**] supports 352 types of hardware algorithms for parallel multiplication and produces the corresponding HDL (Verilog HDL and VHDL) codes containing the explicit gate-level netlists. The generated HDL code can be downloaded from our website [**?**].

2. Basic concept of ARITH

2.1. Formal description of arithmetic algorithms in ARITH

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. The underlying observation here is 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.

We first describe the `typedef` block in details. The weighted number system [1] defined in the `typedef` block is specified by 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:

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

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:

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

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

Using the above notation, we can define various number systems including unconventional number systems. For example, the two's complement (TC) and the radix-2 signed-digit (SD2) number system are given as follows:

- TC

$$\begin{aligned} \boldsymbol{W}_{\boldsymbol{TC}} &\triangleq \langle -2^h, 2^{h-1}, \cdots, 2^i, \cdots, 2^{l+1}, 2^l \rangle, \\ \boldsymbol{D}_{\boldsymbol{TC}} &\triangleq \langle \{0,1\}, \cdots, \{0,1\} \rangle. \end{aligned} \tag{3}$$

- SD2

$$\begin{aligned} \boldsymbol{W}_{SD2} &\triangleq \langle 2^h, 2^{h-1}, \cdots, 2^i, \cdots, 2^{l+1}, 2^l \rangle, \\ \boldsymbol{D}_{SD2} &\triangleq \langle \{-1,0,1\}, \cdots, \{-1,0,1\} \rangle. \end{aligned} \tag{4}$$

Figure 1 shows a `typedef` block for the SD2 number system, where

`SD2.high`: the most significant digit,
`SD2.low`: the least significant digit,
`SD2{i}.weight`: the weight at the ith digit set,
`SD2{i}.min`: the min. integer at the ith digit set,
`SD2{i}.max`: the max. integer at the ith digit set,
`SD2{i}.step`: the step at the ith digit set.

```
1:  typedef SD2;
2:    for(i, SD2.low, SD2.high) begin
3:      SD2{i}.weight = Power(2, i);
4:      SD2{i}.min = -1;
5:      SD2{i}.max = 1;
6:      SD2{i}.step = 1;
7:    end
8:  endtypedef
```

Figure 1. Radix-2 signed-digit number system.

```
 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 PP[];
13:      wire SD2 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_ENCODE U0 (B,X);
23:      PPG          U1 (PP, B, Y);
24:      ACCUMULATE   U2 (F,PP);
25:      SD2TC        U3 (P,F);
26:    end
27:  endmodule
```

Figure 2. Top module of an 8-bit binary signed-digit multiplier.

At lines 2-7, we define the SD2 number system from `SD2.low` digit to `SD2.high` digit.

On the other hand, the `module` blocks include declarative statements of module I/O interface, functional assertion, and structural description. As an example, let us consider the hierarchical description of an 8-bit SD2 multiplier. Figure 2 represents a `module` block of `SD_MULT` at the top of the hierarchy. 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 its digit signals are associated with a specific number system defined by the `typedef` block.

Figure 3 shows the schematics of the SD2 multiplier at different levels of abstraction. Each component (e.g., "Accumulator" in (a), "PPG1" in (b), and a solid square in (c)) can be described as a module in ARITH. The modules in Figs. 3 (a) and (b) correspond to the shaded parts in Figs. 3 (b) and (c), respectively. The internal structure of each module is described by using

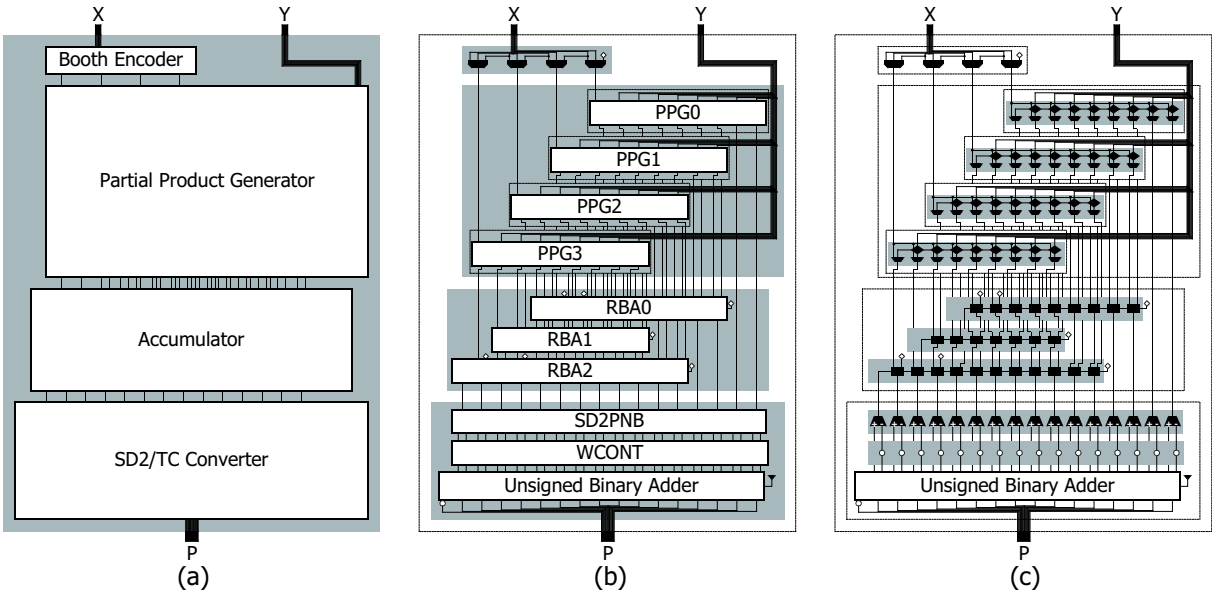

Figure 3. 8-bit binary signed-digit multiplier at various levels of abstraction.

the sub-modules on the corresponding shaded part.

Let us explain the `SD_MULT` module for more details.

- 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. At lines 11-21, the internal signals based on the radix-2/radix-4 Signed-Digit number systems (`SD2`/`SD4`) are declared similarly to the above input/output signals, where `PP[]` represents that `PP` is an array of integer signals. At line 16, `PP` is defined as 4 integer signals. The internal structure is given by the sub-modules `BOOTH_ENCODE` ("Booth Encoder" in Fig. 3), `PPG` ("Partial Product Generator" in Fig. 3), `ACCUMULATE` ("Accumulator" in Fig. 3), and `SD2TC` ("SD2/TC Converter" in Fig. 3) 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 in ARITH at the lower level of abstraction. This is based on the principle that an arithmetic circuit can be divided into simpler sub-circuits which implement arithmetic functions.

2.2. Formal Verification in ARITH system

The ARITH description can be formally verified by the language processing system of ARITH (ARITH system). The formal verification of arithmetic circuits is usually performed by word-level DDs and *BMDs [9], [11]. We can apply the conventional verification techniques to ARITH descriptions. On the other hand, we have a possibility for verifying ARITH descriptions with formula manipulations. The ARITH system has the equivalence checker using formula manipulations in addition to the conventional techniques. Thus, we can reduce the verification time of arithmetic algorithms in ARITH.

In the following, let us briefly describe the verification method based on formula manipulations. The proposed verification method consists of "formula evaluation" and "range evaluation" as follows:

- Formula evaluation:
  Given a module, checks whether its structural description matches its functional assertion. We first obtain the integer equations representing the relationship between integer signals and their digit signals. Second, we extract the set of functional assertions from the sub-modules and rename their integer signals according to the structural description of the given module. Finally, we consider the integer equations obtained from the above two steps as a system of equations, and solve it for the input/output integer signals. If the obtained solution is equal to an integer equation of the functional assertion, the formula evaluation returns "true".

- Range evaluation:
  Given a module, checks whether hardware imple-

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

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

Figure 4. Hardware algorithms supported by MMG.

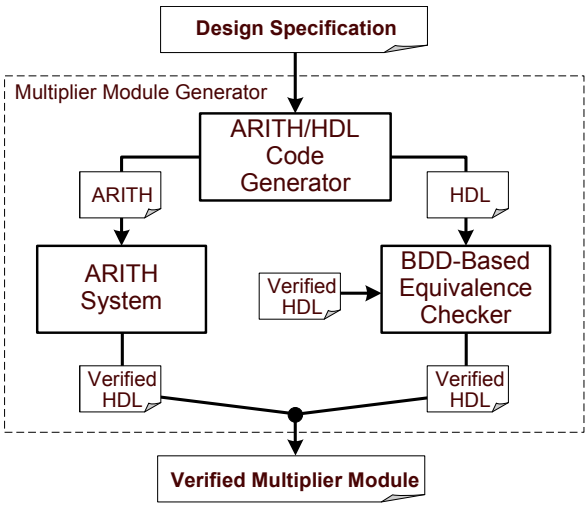

Figure 5. MMG system flow.

mentation is possible under the range constraints of input/output signals. The range constraints are examined on arithmetic intervals. From the range of input/output signals, we evaluate the arithmetic intervals of the functional assertion. If the output arithmetic interval subsumes the input arithmetic interval, the evaluation returns "true". This means that the given module provides sufficient output dynamic range in order to cover the input dynamic range.

We can prove that ARITH description holds correct arithmetic circuit structures if and only if both formula evaluation and range evaluation return true.

3. Design of a multiplier module generator based on ARITH

This section describes an application of ARITH to a multiplier module generator (MMG). In this system, ARITH is used for describing arithmetic algorithms including those using unconventional number systems. The arithmetic algorithm library based on ARITH makes possible to produce reliable multiplier modules in a systematic way.

We consider a parallel multiplier 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.

According to the above specification, we can determine a multiplication algorithm in terms of (i) input word length, (ii) number representation system for operands: signed or unsigned binary, and (iii) hardware algorithms for PPG, PPA, and FSA. Figure 4

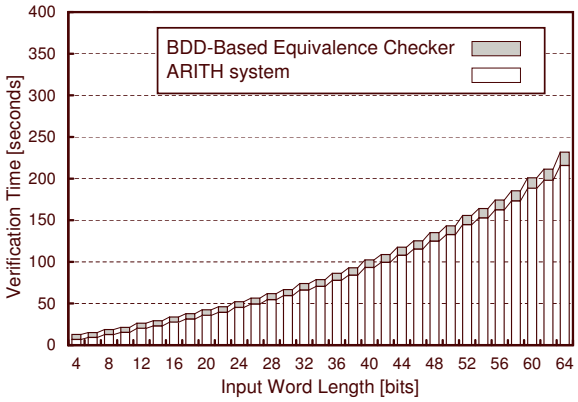

Figure 6. Verification time of MMG.

shows a list of the number representation systems and hardware algorithms supported by MMG. In this paper, let us omit the detailed explanation of the hardware algorithms due to the page limitation (see our website [**?**] and [1], [2] for more details).

3.1. System framework

Figure 5 is a block diagram of the proposed MMG, which consists of (i) ARITH/HDL code generator, (ii) ARITH system, and (iii) BDD-based equivalence checker as follows:

- ARITH/HDL code generator:
  Generates ARITH and HDL codes according to the product specification given by designers. The current version of MMG provides PPGs and PPAs in ARITH and FSAs in HDL at this stage since BDD-based equivalence checker can be effective for FSAs. Note here that all the hardware algorithms for PPA, PPG and FSA can be described in ARITH.

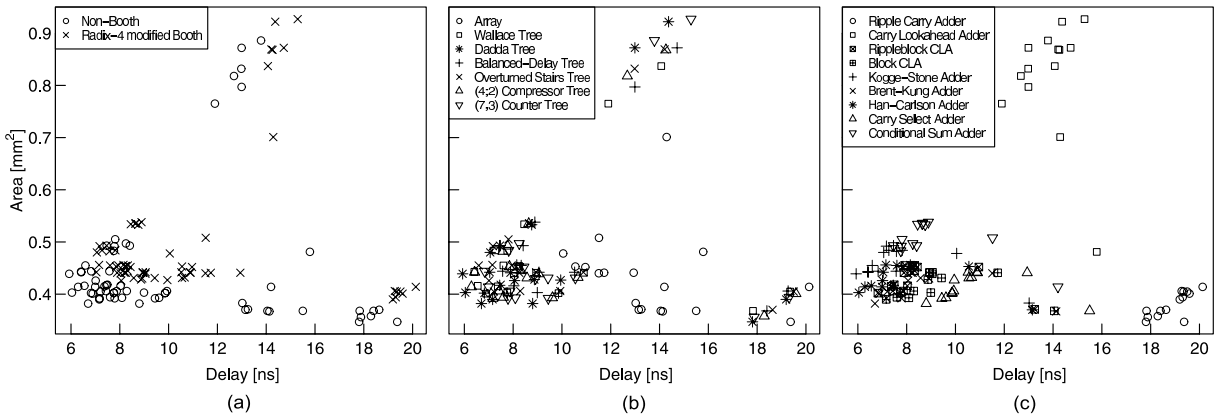

Figure 7. Performance of 32×32 unsigned multipliers for HITACHI 0.18μm process: (a) PPG grouping, (b) PPA grouping, (c) FSA grouping.

- ARITH system:
  Verifies the generated ARITH codes using formal verification techniques as described in the previous section. The verified ARITH codes are translated into the equivalent HDL codes.

- BDD-based equivalence checker:
  Verifies the functional equivalence of the generated HDL code and a reference HDL code by constructing their BDDs [10]. The reference HDL code is derived from the verified ARITH code of a ripple carry adder in advance. The ripple carry adder is easily verified by ARITH system.

Combining the results of ARITH system and BDD-based equivalence checker, MMG obtains the HDL codes verified completely at the algorithm level.

3.2. Experimental designs

In order to examine the verification time of MMG, we have designed a set of parallel multipliers whose operand lengths are ranging from 4 to 64 digits at every 2 digit. Figure 6 illustrates the computation time of MMG on a SUN Blade 2000 with 900MHz UltraSPARC III and 2GB memory. The result shows that MMG performs a complete verification of 64×64 parallel multipliers within 250 seconds. We can confirm here that the proposed verification technique can be useful for ARITH descriptions.

In the following, we evaluate all the types of parallel multipliers generated from MMG. The obtained HDL codes can be synthesized using Synopsys Design Compiler with the compile option "-only_design_rule -boundary_optimization." We employ the Kyoto University's standard-cell libraries targeted for HITACHI 0.18μm process (Typical condition) [**?**], [12]. The delay information is calculated according to the result of place-and-route using Synopsys Milkyway/Apollo. Information on the circuit area is also obtained from the result of place-and-route.

Figure 7 shows all the types of 32×32 unsigned multipliers generated from MMG. The vertical axis indicates the circuit area, and the horizontal axis indicates the circuit delay. Figure 8 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. Figure 9 illustrates the non-booth Array multipliers grouped by FSA for various operand lengths. These results indicate that the proposed MMG system can generate various multipliers faithfully according to the design specifications.

Our website [**?**] provides the detailed comparisons of the typical multipliers as shown in Fig. 10. We can compare the performance of multipliers in terms of (i) hardware algorithms for PPG, PPA, and FSA, (ii) input word lengths, and (iii) target cell libraries.

4. Conclusion

In this paper, we have proposed a multiplier module generator based on an arithmetic description language called ARITH. By using ARITH, we can develop the arithmetic algorithm library in a unified manner. The proposed generator on the website [**?**] supports 352 types of hardware algorithms for parallel multiplication and produces highly-reliable multiplier modules whose functions are completely verified in a formal method. Further investigations are being conducted to develop datapath module generators based on ARITH for DSP systems and public key cryptosystems.

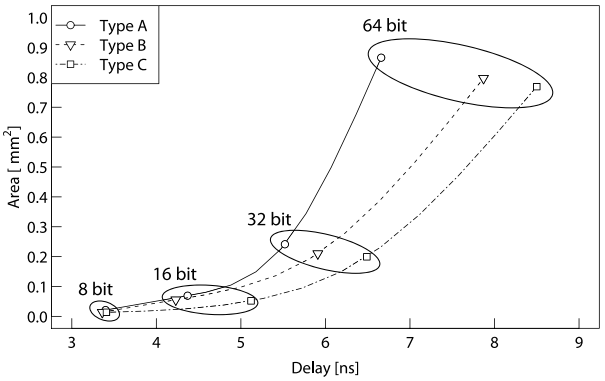

Figure 8. Comparison of three types of unsigned multipliers for various operand lengths.

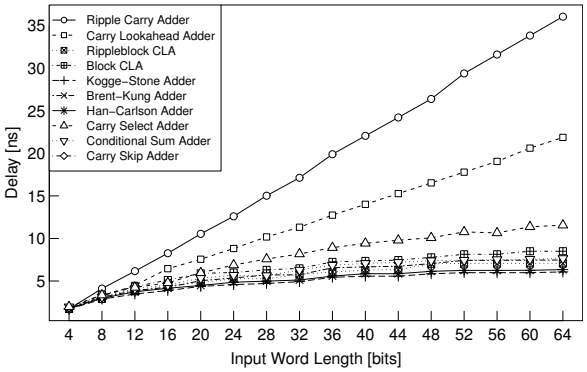

Figure 9. Comparison of non-booth Array multipliers for various operand lengths.

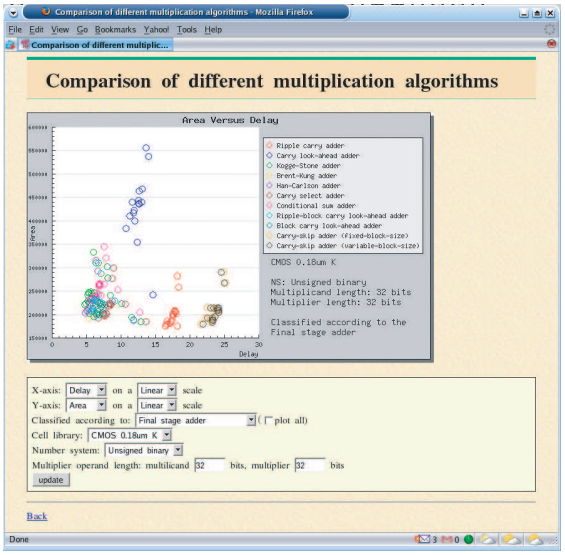

Figure 10. Performance comparisons on website [12].

Acknowledgment

This work is supported by VLSI Design and Education Center(VDEC), the University of Tokyo in collaboration with Synopsys, Inc. and Hitachi Ltd.

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] 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, May 2004.

[4] N. Takagi, H. Yasuura, and S. Yajima,"High-speed VLSI multiplication algorithm with a redundant binary addition tree," *IEEE Trans. Computers*, Vol. 34, No. 9, pp. 789 – 796, September 1985.

[5] A. Avizienis,"Signed-digit number representations for fast parallel arithmetic," *IRE Trans. Electronic Computers*, Vol. 10, pp. 389 – 400, September 1961.

[6] B. Parhami,"Generalized signed-digit number systems: a unifying framework for redundant number representations," *IEEE Trans. Computers*, Vol. 39, No. 1, pp. 89 – 98, January 1990.

[7] S. Kawahito, M. Ishida, T. Nakamura, M. Kameyama, and T. Higuchi,"High-speed area-efficient multiplier design using multiple-valued current-mode circuits," *IEEE Trans. Computers*, Vol. 43, No. 1, pp. 34 – 42, January 1994.

[8] N. Takagi,"Multiple-valued-digit number representations in arithmetic circuit algorithms," *Proc. 32nd IEEE Int. Symp. Multiple-Valued Logic*, pp. 224 – 235, May 2002.

[9] 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.

[10] R. E. Bryant,"Graph-based algorithms for boolean function manipularion," *IEEE Trans. Computers*, Vol. C-35, No. 8, pp. 677 – 691, August 1986.

[11] Multiplier Module Generator based on ARITH. http://www.aoki.ecei.tohoku.ac.jp/arith/mg/

[12] Y.-A. Chen and E. R. Bryant,"ACV: An arithmetic circuit verifier," *Proc. of International Conference on Computer-Aided Design*, 1996.

[13] VLSI Design and Education Center (VDEC) website. http://www.vdec.u-tokyo.ac.jp/English/

[14] 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, January 2003.