Arithmetic Module Generator Based on Arithmetic Description Language

Yuki Watanabe, Naofumi Homma, and Takafumi Aoki
Graduate School of Information Sciences
Tohoku University
Sendai 980-8579, Japan
watanabe@aoki.ecei.tohoku.ac.jp

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

Abstract—This paper presents a new approach to designing arithmetic circuits using 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. In this paper, we propose an application of ARITH to an arithmetic module generator, which supports various hardware algorithms for 2-operand adders, multi-operand adders, multipliers, constant-coefficient multipliers and multiply accumulators. From the generator, we can obtain the highly-reliable arithmetic modules whose functions are completely verified at the algorithm level.

I. 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]. 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],[5] 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 equivalent HDL descriptions.

This paper proposes an application of ARITH to an arithmetic module generator. By using ARITH, we can develop an arithmetic algorithm library in the generator. The ARITH-based library makes possible to generate a variety of arithmetic modules in a systematic way. 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 such as BMDs [6] and BDDs [7]. The verified ARITH descriptions are finally translated into the equivalent HDL descriptions.

The developed generator [8] supports various hardware algorithms for 2-operand adders, multi-operand adders, parallel multipliers, constant-coefficient multipliers and multiply accumulators. For example, we have 352 types of hardware algorithms for parallel multipliers. Given a design specification, the generator can produce the corresponding HDL (Verilog HDL and VHDL) codes containing the explicit gate-level netlists. These performances are comparable or sometimes superior to those of modules obtained from the conventional logic synthesis tools.

II. Arithmetic Description Language: ARITH

This section summarizes the description and verification of arithmetic algorithms in ARITH (see [5] for more details).

A. 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. 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.

The weighted number system \([1]\) defined in the typedef block is associated with the tuple \((W, D)\), where \(W\) is the weight vector and \(D\) is the digit set vector, respectively. More precisely, \(W\) and \(D\) are defined as follows:

\[
W \triangleq (w_h, w_{h-1}, \ldots, w_l, w_l), \\
D \triangleq (D_h, D_{h-1}, \ldots, D_l+1, D_l),
\]

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:

\[
[min, max, step] \\
\triangleq \{ u \in \mathbb{Z} | (min \leq u) \land (u \leq max) \land (\exists j \in \mathbb{Z}_{0+}, u = min + step \cdot j) \},
\]

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

Figure 1 shows a typedef block for the unsigned binary (UB) number system. The weight, \(min, max,\) and \(step\) for the \(i\)th digit set are defined at lines 3-6, respectively. Using the notation, we can define various number systems including unconventional number systems such as Redundant-Binary (RB) number system and Signed-Digit (SD) number systems.

On the other hand, the module block is used to describe an arithmetic algorithm in a hierarchical fashion. As an example, let us consider a 4-bit unsigned array multiplier. Figure 2 represents the module block at the top level of the hierarchy. The module block includes declarative statements of I/O interface (at lines 2-8), functional assertion (at line 9) and structural description (at lines 10-24). As shown in Fig. 2, the functional assertion is represented as an integer equation, and the structural description is given as a combination of sub-modules at a lower-level of abstraction. Figure 3 shows an example of hierarchy diagrams for the array multiplier.

**B. Formal Verification in ARITH system**

The ARITH description can be formally verified by the language processing system of ARITH (ARITH code verifier). The formal verification of arithmetic circuits is usually performed by word-level DDs or "BMDs [6], [9]. We can apply the conventional verification techniques to ARITH descriptions. On the other hand, we have a possibility for verifying ARITH descriptions with formula manipula-
tions. The ARITH code verifier can perform the equivalence checking with formula manipulations in addition to the conventional techniques. The hybrid verification approach is useful especially for arithmetic algorithms in ARITH.

In the following, we briefly describe the equivalence checking with 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 given structural description. Finally, we solve the system of integer equations obtained from the above two steps for the input/output signals. If the obtained solution is equal to the given functional assertion, the formula evaluation returns "true". In this evaluation, we employ Gaussian Elimination method and Groebner-bases method [10] for solving the system of linear and non-linear equations, respectively.

- **Range evaluation:**
  Given a module, checks whether hardware implementation 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 range evaluation returns "true". This means that the given module provides sufficient output dynamic range in order to cover the input dynamic range.

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

III. DESIGN OF ARITHMETIC MODULE GENERATOR BASED ON ARITH

This section describes an application of ARITH to an arithmetic module generator (AMG). The product specifications considered here are 2-operand addition, multi-operand addition, multiplication, constant-coefficient multiplication, and multiply accumulation.

A. System framework

Figure 4 is a block diagram of AMG, which consists of (i) ARITH code generator, (ii) ARITH code verifier, and (iii) ARITH/HDL converter as follows:

- **ARITH code generator:**
  Generates ARITH codes according to the design specification given by designers. The arithmetic algorithm library is used for the generation.

- **ARITH code verifier:**
  Verifies the generated ARITH codes using the equivalence checking with formula manipulations. The BDD-based equivalence checking [7] is performed only for 2-operand adders.

- **ARITH/HDL converter:**
  Converts the verified ARITH codes into the equivalent HDL codes. This is just a syntactical conversion without intelligence.

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

Figure 5 shows the system framework for our web service. From the web interface, a design specification is determined in terms of (i) target function, (ii) hardware algorithms, (iii) operand length, and (iv) number representation system for operands. AMG then generates the 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.

B. 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. In the following, we briefly describe the hardware algorithms that can be handled by AMG (see our AMG website [8] and [1][2] for more details).

B.1 2-operand addition algorithms

AMG has 11 types of 2-operand addition algorithms. Let us first describe Ripple carry adder, Carry look-ahead adder, Block carry look-ahead adder, and Ripple block carry look-ahead adder as follows:

- Ripple carry adder:
  - Is the most straightforward implementation for \( n \)-bit 2-operand addition, which requires \( n \) full adders (FAs).
  - The carry-out of the \( i \)th FA is connected to the carry-in of the \( (i+1) \)th FA. In the general case, the worst-case delay is \( n\Delta FA \), where \( \Delta FA \) denotes the operation time of an FA.

- Carry look-ahead adder:
  - Is based on an attempt to generate all incoming carries in parallel and avoid waiting until the correct carry propagates from the stage of the adder where it has been generated. It is estimated that the total delay is \( 5\Delta G \), where \( \Delta G \) is the delay of a single gate, regardless of \( n \), the number of bits in each operand. However, for a large value of \( n \), say, \( n = 32 \), an extremely large number of gates is needed and, more importantly, gates with a very large fan-in are required.

- Ripple-block carry look-ahead adder:
  - Reduces the fan-in and fan-out difficulties inherent in carry look-ahead adders. An RCLA consists of \( m \)-bit blocks arranged in such a way that carries within blocks are generated by carry look-ahead but carries between blocks are rippled. The block size \( m \) is fixed to 4 in the generator. The RCLA design is based on the multiple levels of carry look-ahead. If there are five or more blocks in an RCLA, four blocks are grouped into a single superblock, with the second level of look-ahead applied to the superblocks.

- Block carry look-ahead adder:
  - Is another way to design a practical carry look-ahead adder. The idea is to reverse the basic design principle of RCLA, that is, to ripple carries within blocks but to generate carries between blocks by look-ahead.

Secondly, we describe parallel prefix adders such as Kogge-Stone adder, Brent-Kung adder, and Han-Carlson adder. These adders are constructed out of fundamental carry operators denoted by \( \circ \) as follows: \( (G', P') \circ (G'', P'') = (G'' + G' \cdot P'', P' \cdot P') \), where \( P'' \) and \( P' \) indicate the propagated carries, \( G'' \) and \( G' \) indicate the generated carries, the operators “+” and “\( \cdot \)” indicate the logical OR and AND, respectively.

- Kogge-Stone adder:
  - Is a full binary tree with minimum fun-out and minimum logic depth in the parallel prefix adders. That results in a fast adder but with a large area.

- Brent-Kung adder:
  - Is an extreme design with maximum logic depth and minimum area in the parallel prefix adders.

- Han-Carlson adder:
  - Is a hybrid design combining stages from Brent-Kung adder and Kogge-Stone adder.

We then describe Conditional sum adder and Carry select adder. The basic idea is to generate two sets of outputs for a given group of operand bits, say, \( k \) bits. Each set includes \( k \) sum bits and an outgoing carry. One set assumes that the eventual incoming carry will be zero, while the other assumes that it will be one. Once the incoming carry is known, we need only to select the correct set of outputs (out of the two sets) without waiting for the carry to further propagate through the \( k \) positions.

- Conditional sum adder:
  - Divides the given \( n \)-bit operands into two groups of size \( n/2 \) bits each. Each of these can be further divided into two groups of \( n/4 \) bits each. This process can, in principle, be continued until a group of size 1 is reached. The above idea is applied to each of groups separately.

- Carry select adder:
  - Employs the above idea so that the size of the \( k \)th group can equalize the delay of the ripple-carry within the group and that of the carry-select chain from the 1st group to the \( k \)th group. In this generator, the group sizes follow the simple arithmetic progression \( 1,1,2,3,\ldots \).

Finally, we describe carry-skip adders such as Fixed-block-size carry-skip adder and Variable-block-size carry-skip adder. A carry-skip adder reduces the carry-propagation time by skipping over groups of consecutive adder stages, and is usually comparable in speed to the carry look-ahead technique, but it requires less chip area and consumes less power.

- Fixed-block-size carry skip adder:
  - Has a fixed block size \( k \) to minimize the time for the longest carry-propagation chain. The optimal block size \( k_{opt} \) follows: \( k_{opt} = \sqrt{n} \), where \( n \) denotes the operand length.
### Table I

<table>
<thead>
<tr>
<th>Basic component</th>
<th>Level of design optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Word-level design</td>
</tr>
<tr>
<td>(3,2) counter</td>
<td>Array</td>
</tr>
<tr>
<td>(4:2) compressor</td>
<td>(4:2) compressor tree</td>
</tr>
<tr>
<td>(7,3) counter</td>
<td>RB adder</td>
</tr>
</tbody>
</table>

- Variable-block-size carry skip adder:
  Includes $L$ blocks with sizes $k_1, k_2, \ldots, k_L = 1, 2, 3, \ldots, 3, 2, 1$ in this generator. This reduces the ripple-carry delay through these blocks.

#### B.2 Multi-operand addition algorithms

AMG has 8 types of multi-input 2-output addition algorithms. Table I 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.

We first describe Array, Wallace tree, Balanced-delay tree, and Overturned-stairs tree consisting of word-level (3,2) counters (carry-save adders: CSAs). Assuming that the number of operands is $N$, we compare the four algorithms in terms of the number of adder stages and wiring tracks. The wiring track indicates a long wire between adjacent bit-slices. The three tree structures differ in the number of required wiring tracks; these in turn, affect the layout area.

- **Array:**
  Is a straightforward way to accumulate multiple operands using a series of CSAs. The $N$-operand array consists of $N - 2$ carry-save adders.

- **Wallace tree:**
  Is known for the optimal computation time, when adding multiple operands to two outputs using CSAs. The Wallace tree guarantees the lowest overall delay $O(\log N)$ but requires the largest number of wiring tracks $O(\log N)$ in this category.

- **Balanced-delay tree:**
  Requires the smallest number of wiring tracks ($= 2$) but has the highest overall delay $O(\sqrt{N})$ compared with Wallace tree and Overturned-stairs tree.

- **Overturned-stairs tree:**
  Requires smaller number of wiring tracks ($= 3$) than Wallace tree, and has lower overall delay $O(\sqrt{N})$ than Balanced delay tree.

- **Dadda tree:**
  Is based on bit-level (3,2) counters. To reduce the hardware complexity, we allow the use of (2,2) counters in addition to (3,2) counters. Given the matrix of multiple operands, the number of bits in each column is reduced to minimize the number of (3,2) and (2,2) counters.

- **(7,3) counter tree:**
  Is based on bit-level (7,3) counters. To reduce the hardware complexity, we allow the use of (6,3), (5,3), (4,3), (3,2), and (2,2) counters in addition to (7,3) counters. We employ Dadda’s strategy for constructing them.

- **(4;2) compressor tree:**
  Is composed of word-level (4;2) compressors. That results in a more regular structure than an ordinary CSA tree made of (3,2) counters because the multiple operands are added up in the form of a binary tree.

- **RB addition tree:**
  Is composed of word-level RB adders. That results in a more regular structure than an ordinary CSA tree made of (3,2) counters. The RB addition tree is closely related to (4;2) compressor tree. Note here that the RB number should be encoded into a vector of binary digit in the binary-logic implementation. In AMG, we employ a minimum length encoding based on positive-negative representation.

#### B.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 multiple-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 6 shows hardware algorithms for PPG, PPA, and FSA. Note here that PPAs and FSAs correspond to multiple-operand adders and 2-operand adders. In addition, we have 2 types of PPGs in AMG as follows:

- **Non-Booth**: Generates partial products from the logical AND of the multiplicand with the multiplier.
- **Radix-4 modified Booth**: Employs radix-4 modified Booth recoding to generate partial products. The Booth recoding of the multiplier reduces the number of partial products and hence has a possibility of reducing the amount of hardware involved and the execution time.

In total, AMG supports 352 types of hardware algorithms for parallel multiplication.

### B.4 Constant-coefficient multiplication algorithms

AMG provides constant-coefficient multipliers in the form: \( y = Rx \), where \( R \) is an integer coefficient, and \( x \) and \( y \) 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 \( R \), and the complexity of the multiplier structure significantly varies with the coefficient value \( R \).

We consider here the use of special number representation called Signed-Weight (SW) number system [11], which is useful for constructing compact PPAs. At present, the combination of CSD (Canonic Signed-Digit) coefficient encoding technique [12] with the SW-based PPAs seems to provide the practical hardware implementation of fast constant-coefficient multipliers. As a result, AMG supports 154 types of such hardware algorithms for constant-coefficient multiplication, where the range of \( R \) is from \(-2^{31}\) to \(2^{31} - 1\).

### B.5 Multiply accumulation algorithms

---

Fig. 7. Latency of 2-operand adders for various operand lengths.

Fig. 8. Area of 2-operand adders for various operand lengths.

Fig. 9. Output arrival profile of multi-operand adders.

Fig. 10. Comparison of three types of unsigned multipliers.
TABLE II
Verification time of AMG

<table>
<thead>
<tr>
<th>Arithmetic unit</th>
<th>Verification time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8 bits</td>
</tr>
<tr>
<td>2-operand adder</td>
<td>0.46</td>
</tr>
<tr>
<td>Multi-operand adder</td>
<td>3.90</td>
</tr>
<tr>
<td>Multiplier</td>
<td>6.28</td>
</tr>
<tr>
<td>Multiply accumulator</td>
<td>7.03</td>
</tr>
</tbody>
</table>

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 \( N(\geq 1) \) is the integer constant. A multiply accumulator is generated by a combination of hardware algorithms for multipliers and constant-coefficient multipliers. All the partial products from the PPGs are accumulated in carry-save form by the single PPA. The carry-save form is converted to the corresponding binary output by the FSA.

C. Experimental designs

To evaluate the verification time of AMG, we have designed the four types of arithmetic modules whose operand lengths are 8, 16, 32, and 64 bits. Table II illustrates the verification time of AMG on Intel Pentium 4 CPU 2.80GHz and 2GB memory, where “2-operand adder” indicates the Kogge-Stone adder, “Multi-operand adder” indicates the Wallace tree, “Multiplier” indicates the Ripple-carry adder and Array architecture, and “Multiply accumulator” indicates the Han-Carlson adder and Dadda architecture having a function: \( P = X \times Y + Z + W \). The result shows that AMG performs a complete verification of the 64-bit multiply accumulator at most 200 seconds.

In the following, let us evaluate the performance of arithmetic modules generated from AMG. The modules can be synthesized using Synopsys Design Compiler with the compile option “-only design rule -boundary optimization.” For the synthesis, we employ the Kyoto University’s standard-cell library targeted for HITACHI 0.18 \( \mu \)m process (Typical condition) \([13],[14]\). The delay/area information is 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 can be summarized as shown in Figs. 7-10. Figures 7-8 illustrate the performance of 2-operand adders for various operand lengths. Figure 9 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 10 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 suggest that AMG can generate the arithmetic modules faithfully according to the design specifications.

Figures 11 shows all the types of 32×32 unsigned binary multipliers for HITACHI 0.18\( \mu \)m process: (a) PPG grouping, (b) PPA grouping, (c) FSA grouping.
TABLE III
Performance comparisons

<table>
<thead>
<tr>
<th>Arithmetic module</th>
<th>Type</th>
<th>Conventional synthesis tool</th>
<th>AMG</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-operand adder (64bit)</td>
<td>Small</td>
<td>8455.68</td>
<td>16.33</td>
</tr>
<tr>
<td></td>
<td>Fast</td>
<td>90984.96</td>
<td>1.22</td>
</tr>
<tr>
<td>Multiplier (32 bit)</td>
<td>Small</td>
<td>169351.67</td>
<td>19.28</td>
</tr>
<tr>
<td></td>
<td>Fast</td>
<td>212398.08</td>
<td>11.22</td>
</tr>
<tr>
<td>Constant-coefficient multiplier (32 bit)</td>
<td>Small</td>
<td>102558.72</td>
<td>17.91</td>
</tr>
<tr>
<td></td>
<td>Fast</td>
<td>220577.28</td>
<td>8.60</td>
</tr>
<tr>
<td>Multiply accumulator (32 bit)</td>
<td>Small</td>
<td>185633.28</td>
<td>21.48</td>
</tr>
<tr>
<td></td>
<td>Fast</td>
<td>237304.31</td>
<td>12.28</td>
</tr>
</tbody>
</table>

synthesis tools. Note here that we have different distributions in the case of other target technologies.

The other evaluation results can be available on our website [8] which would be helpful for prospective designers as a reference.

IV. Conclusion

In this paper, we have 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.

References

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