Abstract

This paper presents a formal approach to verify arithmetic circuits using symbolic computer algebra. Our method describes arithmetic circuits directly with high-level mathematical objects based on weighted number systems and arithmetic formulae. Such circuit description can be effectively verified by polynomial reduction techniques using Gröbner Bases. In this paper, we describe how the symbolic computer algebra can be used to describe and verify arithmetic circuits. The advantageous effects of the proposed approach are demonstrated through experimental verification of some arithmetic circuits such as multiply-accumulator and FIR filter. The result shows that the proposed approach has a definite possibility of verifying practical arithmetic circuits where the conventional techniques failed.

1 Introduction

Arithmetic circuits are of major importance in today’s computing and signal processing systems. In addition to the standard binary arithmetic algorithms, we can introduce non-binary arithmetic algorithms for enhancing the performance of arithmetic circuits [1, 2]. These include high-radix number systems, redundant number systems, and other dedicated data structures designed for specific applications. On the other hand, most of such arithmetic algorithms are devised by researchers who had trained in a particular way to understand the basic arithmetic fundamentals. This sometimes requires us to verify structural details of arithmetic circuits at the lowest level of abstraction.

Addressing the above problem, this paper proposes a formal approach to describe and verify arithmetic algorithms with various number systems (The basic concept of the proposed description itself was first reported in [3]). Our method describes arithmetic circuits directly with high-level mathematical objects based on weighted number systems and arithmetic formulae. The arithmetic circuits are described in a hierarchical manner, where each component (i.e. sub-circuit) has a function and an internal structure defined by integer equations. The verification of an arithmetic circuit can be performed by checking for every sub-circuit whether the function is obtained from the internal structure. The equivalence checking can be performed by formula manipulations based on Gröbner Bases and polynomial reduction technique [4].

Early researches on arithmetic circuit verification are primarily based on word-level DDs or *BMDs [5, 6, 7, 8]. Reference [7], for example, presents a hardware description language, called ACV language, to verify arithmetic circuits in a hierarchical fashion using *BMDs. Our description method can also be verified by the *BMDs-based equivalence checking. However, the conventional approaches have only limited capability to verify arithmetic functions at higher-levels of hierarchy since they are inherently based on bit-level data operations. The proposed verification method, on the other hand, can be effective even for such highly-hierarchized arithmetic circuits.

This paper is organized as follows: In Section 2, we present the formal description of arithmetic circuits based on integer equations. In Section 3, we propose the equivalence checking method using Gröbner Bases and polynomial reduction technique to verify the formal description. Section 4 describes experimental results using various arithmetic circuits for both the proposed method and *BMD-based method. Finally, Section 5 concludes this paper.

2 Arithmetic circuit representation

The function of arithmetic circuits is usually represented by logic functions or lookup tables defining all the input-output combinations uniquely. Such representations, however, are not always suitable for representing large arithmetic circuits with many input variables. Assuming that arithmetic circuits implement arithmetic functions which should be dealt with in the integer arithmetic domain rather than the Boolean logic domain, we introduce a formal representation of arithmetic circuits in a hierarchical manner based on integer equations.
Figure 1. Arithmetic circuit graph.

Figure 1 shows an arithmetic circuit graph (ACG) for representing a circuit structure. An ACG $G$ is defined as

$$G = (N, E),$$

where $N$ is a set of nodes, and $E$ is a set of directed edges. The node represents an arithmetic circuit which has its functional assertion and internal structure. The directed edge, on the other hand, represents the flow of integer data between the nodes, and defines the data dependency. We assume that every node has one edge connection at least.

A node $n \in N$ is given by $n = (F, G')$, where $F$ is a set of integer equations for the functional assertion and $G'$ is an ACG for the internal structure. A node that does not have its internal structure is said to be lowest level, and is represented as $n = (F, nil)$. Let $\text{lhs}$ and $\text{rhs}$ be integer expressions given by an integer variable, an integer constant or a combination of the two or more connected by arithmetic operations $+ , -, \text{and} \times$. An integer equation is defined as a relation $\text{lhs} = \text{rhs}$, where $\text{lhs}$ and $\text{rhs}$ indicate the output and input integer expressions, respectively.

A directed edge $e \in E$ is defined as $e = (\text{src}, \text{dest}, x)$, where $\text{src}$ indicates the source node, $\text{dest}$ indicates the destination node, and $x$ indicates the integer variable. A directed edge is said to be a half edge if one endpoint of the directed edge is not connected to any node. The half edge represents an external input or output for the given ACG. Each integer variable is associated with a weighted number system. A weighted number system $NS = (W, D)$, where $W$ is the weight vector, and $D$ is the digit set vector, respectively. More precisely, $W$ and $D$ are given as

$$W = (\cdots, w_h, \cdots, w_{i+1}, w_i, w_{i-1}, \cdots, w_1, \cdots),$$
$$D = (\cdots, D_h, \cdots, D_{i+1}, D_i, D_{i-1}, \cdots, D_1, \cdots),$$

where $w_i$ and $D_i$ are the weight and digit set in the $i$th digit. Let $h$ and $l$ $(l \leq h)$ be the most and least significant digits, respectively. An integer variable is given as $x = (NS, (h, l))$, where the tuple $(h, l)$ is called range constraint. Using the above notation, we easily handle a specific integer variable $x_i$ in the $i$th digit, which is called digit variable. A combination of digit variables represents an integer variable at a higher level of abstraction.

Let $x$ be an integer variable associated with a weighted number system. Let $x_i$ $(l \leq i \leq h)$ be the $i$th digit variable. The function of a decomposition node is given as

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

On the other hand, The function of a composition node is given as an inverse relation between the above input and output. The ranges of $x_i$ and $x$ are given as

$$R_{x_i} = \{w_i \cdot d \mid d \in D_i\},$$
$$R_x = \{r_h + \cdots + r_i + \cdots + r_l \mid r_h \in R_{x_h}, \ldots, r_i \in R_{x_i}, \cdots, r_l \in R_{x_l}\}.$$

By using the decomposition/composition nodes, we can change the level of abstraction in edge representation. Note that these nodes are implemented by wiring and have no internal structures.

For example, an integer variable $x \in \{0, 1, 14, 15\}$ is represented as

$$W_{UB} = (\cdots, 2^{i+1}, 2^i, 2^j, \ldots),$$
$$D_{UB} = (\cdots, \{0, 1\}, \{0, 1\}, \{0, 1\}, \cdots),$$
$$x = (UB, (3, 0)).$$

where the $UB = (W_{UB}, D_{UB})$ indicates the unsigned binary number system. Thus, the integer variable $x$ consists of four digit variables $x_i$ $(0 \leq i \leq 3)$. The function of the decomposition node is given as $x = x_3 + x_2 + x_1 + x_0$.

The above ACG can be used also for representing any logic circuits. For example, the functions of NOT, OR, AND, and XOR circuits are given as

$$\text{NOT}(v) = 1 - v,$$
$$\text{OR}(v, u) = v + u - vu,$$
$$\text{AND}(v, u) = vu,$$
$$\text{XOR}(v, u) = v + u - 2vu,$$

respectively, where the variables $u$ and $v$ satisfy the conditions $u^2 = u$ and $v^2 = v$, respectively. Thus, we can represent any arithmetic/logic circuits using ACGs in a hierarchical manner.

As an example, Figure 2 shows the ACGs for 4-bit 4-input multiply-accumulators at various levels of abstraction, where the square blocks indicate the nodes. The “Multiply-accumulator” block in (a) is in the highest level of hierarchy. The blocks in Figs. 2 (a), (b), and (c) correspond to the shaded parts in Figs. 2 (b), (c), and (d), respectively. Each block has its internal structure given by a combination of smaller blocks in the corresponding shaded part. For example, “Accumulator” block in (b) consists of four smaller blocks “CSA0”, “CSA1”, “CSA2”, and “CSA3.” Tables 1 and 2 shows the functional assertions and integer variables,
Figure 2. Arithmetic circuit graphs for 4-input multiply-accumulator.

respectively. Note that decomposition/composition nodes are not shown in Fig. 2 and Table 1. For example, an ACG $G_3$ is represented as

$$G_3 = \left\{ n_9, n_{10}, n_{11}, n_{12}, \{ x_2, x_3, p_0, p_1, p_2, p_3, t_0, t_1, w_{22}, w_{23}, w_{24}, w_{25}, w_{26}, w_{27} \} \right\}.$$  

where

$$n_9 = \{ w_{22} + w_{23} = p_0 + p_1 + x_2 \}, \quad G_8),$$  

$$n_{10} = \{ w_{24} + w_{25} = w_{23} + w_{22} + x_3 \}, \quad G_9),$$  

$$n_{11} = \{ w_{26} + w_{27} = w_{25} + w_{24} + p_2 \}, \quad G_{10}),$$  

$$n_{12} = \{ t_0 + t_1 = w_{27} + w_{26} + p_3 \}, \quad G_{11}).$$  

(14)
Table 1. Functional assertions in Fig. 2

Table 2. Number systems and integer variables in Fig. 2

3 Arithmetic circuit verification using symbolic computer algebra

3.1 Basic concept

We now propose a formal method for verifying arithmetic circuits based on the above representation. The basic idea is to check for every node in the graph (i) whether its function is derived from its structure or (ii) whether its function is implemented on hardware under the range constraint of input/output variables. The proposed method includes formula evaluation and range evaluation according to the above two tasks.

Figure 3 shows an overview of the verification procedure. This algorithm separates lowest-level nodes other higher-level nodes for complete and efficient verification. Each lowest-level node shall have a correct functional assertion, and thus be checked only by the range evaluation (RangeEvaluation). If the left-hand (output) range subsumes the right-hand (input) range in the functional assertion, the range evaluation returns true. We assume here that a lowest-level node represents a bitwise operation or a small word-level operation which can be simply verified by a greedy method using the functional assertion and in-
The formula evaluation can be preformed by formula manipulation, which utilizes Gröbner Basis [4] and polynomial reduction techniques. In the following, we describe the fundamentals of Gröbner Basis, and then explain how the formula manipulation is applied to the formula evaluation.

### 3.2 Gröbner Basis

Let \( m = x_{e1}^1 x_{e2}^2 \cdots x_{en}^e \) be the monomial where \( x_1, x_2, \cdots, x_n \) are the variables and \( e_1, e_2, \cdots, e_n \) are the nonnegative integers. The polynomial is the finite sum of terms in the form \( c \cdot m \) where the coefficient \( c \) is the rational number.

We denote \( R[x] = R[x_1, x_2, \cdots, x_n] \) as the ring of all polynomials obtained from variables \( x = (x_1, x_2, \cdots, x_n) \). Every finite set of polynomials \( P = \{p_1, p_2, \cdots, p_k\} \subseteq R[x] \) generates an polynomial ideal (or simply, ideal) \( I \) as follows:

\[
I = \{ a_1 p_1 + a_2 p_2 + \cdots + a_k p_k \mid a_1, a_2, \cdots, a_k \in R[x] \}.
\]

The set \( P \) is called generator or basis of \( I \).

Buchberger [9] has shown that an arbitrary ideal basis can be transformed into a basis with specific properties, which is called Gröbner Basis. A reduced Gröbner Basis forms a canonical representation for a polynomial ideal. The canonical representation enables us to check whether the given polynomial is in the ideal.

To convert a finite set of polynomials into a Gröbner Basis, Buchberger’s algorithm employs the following polynomial reduction, normal form and S-polynomials.

**Definition 3.1** Given a polynomial \( p \), let \( HT(p) \) be the monomial in the maximal term (or head term) among those in \( p \) with respect to a total ordering of the variables. Let \( HC(p) \) be the coefficient of the maximal term. Given polynomials \( p, q \neq 0 \), suppose that a term \( M \), which can be divided by \( HT(q) \), exists in \( p \). The polynomial reduction \( h \) of \( p \) and \( q \) is defined as

\[
h = p - \frac{C_M M}{HC(q)HT(q)} q,
\]

where \( C_M \) is the coefficient of \( M \).

**Example 3.1** Two polynomials \( p \) and \( q \) are given as

\[
p = x^5 - 2yx^2 + y^5,
\]

\[
q = x^3 + y^3 - 1.
\]

Using a lexicographic order \( y > x \), we have

\[
h = (x^5 - 2yx^2 + y^5) - \frac{1}{1 \cdot 1 \cdot 1 \cdot 1 \cdot 1} \cdot y_5 \cdot (x^2 + y^2 - 1)
\]

\[
= x^5 - y_3 x^2 - 2yx^2 + y_3.
\]

**Definition 3.2** For a polynomial \( p \), a unique element \( k \) is given by repeated polynomial reductions with respect to a set of polynomials \( Q = \{q_1, \cdots, q_m\} \). The element is called a normal form, and is denoted by \( k = NF_Q(p) \).
**Definition 3.3** Given two polynomials \( p \) and \( q \), the S-polynomial of \( p \) and \( q \) is defined as

\[
S_{\text{poly}}(p, q) = \frac{\text{lcm}(HT(p), HT(q))}{\text{lcm}(HT(p), HT(q))} p - \frac{\text{lcm}(HT(p), HT(q))}{\text{lcm}(HT(p), HT(q))} q, \tag{24}
\]

where \( \text{lcm} \) is the least common multiple of \( p \) and \( q \).

**Example 3.2** The S-polynomial of Eqs. (21) and (22) is

\[
S_{\text{poly}}(p, q) = \frac{y^5}{(1) - y^2} (x^5 - 2yx^2 + 2y^2 - 1) = x^5 - y^3 x^2 - 2yx^2 + y^3. \tag{25}
\]

**Definition 3.4** The basis \( B \) of the ideal \( I \) is a Gröbner Basis when the arbitrary polynomial \( p \in I \) can be reduced to 0, that is \( NF_B(p) = 0 \).

Figure 4 shows Buchberger’s algorithm using the above notation. This algorithm can generate a Gröbner Bases from any polynomial set with a finite steps. The calculation time heavily depends on the complexity of \( P \) associated with the number and degree of polynomials.

**3.3 Formula evaluation using computer algebra**

Figure 5 illustrates the formula evaluation procedure using Gröbner Basis. Given a functional assertion \( f \) and internal structure \( G = (N, E) \), a set of polynomials \( P \) is generated from functional assertions included in the internal structure. A Gröbner Basis \( B \) is then obtained from \( \text{GröbnerBasis}(P) \) as shown in Fig. 4. If the normal form of \( f \) with respect to \( B \) is equal to zero, \( f \) is a member of the ideal from \( P \). This means that the functional assertion can be realized with the internal structure. Therefore, \( \text{FormulaEvaluation}(f, G) \) returns true.

**Example 3.3** Consider a formula evaluation for \( n_0 = (s = x_0 x_1 + x_2 + x_3, G_1) \) in Fig. 2. We first derive a set of polynomials \( P \) from the internal structure including four nodes \( n_1, n_2, n_3 \) and \( n_4 \).

\[
P = \{p_0 + p_1 + p_2 + p_3 - (x_0 x_1 + 2^{10}), \]
\[
t_0 + t_1 - (p_0 + p_1 + x_2 + x_3 + p_2 + p_3), \]
\[
us - (t_0 + t_1), s - (us - 2^{10}) \}. \tag{26}
\]

Then, we have the Gröbner Basis \( B \) from \( P \) as follows:

\[
B = \{-2^{10} - s + us, -2^{10} - s + t_0 + t_1, \]
\[
s - x_0 x_1 - x_2 - x_3, \]
\[
-2^{10} + p_0 + p_1 + p_2 + p_3 - s + x_2 + x_3 \}. \tag{27}
\]

The normal form of the function with respect to \( B \) is given as \( NF_B(s - (x_0 x_1 + x_2 + x_3)) = 0 \). Therefore, the formula evaluation returns true.

**Example 3.4** Consider a formula evaluation for a full adder whose structure includes logic expressions. The function of a full adder can be expressed as \( c + s = x + y + z \) and the structure is given by the following logic equations.

\[
c_l = x_l \& y_l | z_l \& (x_l | y_l), \tag{28}
\]
\[
s_l = x_l \oplus y_l \oplus z_l, \tag{29}
\]

where the symbols \( x_l, y_l, z_l, c_l, \) and \( s_l \) are Boolean variables. We convert the logic equations into integer equations, and thus have

\[
c_l = x_l y_l - (x_l + y_l - x_l y_l) z_l + x_l y_l (x_l + y_l - x_l y_l) z_l, \tag{30}
\]
\[
s_l = x_l y_l - (x_l + y_l) z_l + 2x_l (y_l + z_l - 2y_l z_l), \tag{31}
\]

where

\[
c_l^2 = c_l, s_l^2 = s_l, x_l^2 = x_l, y_l^2 = y_l, z_l^2 = z_l. \tag{32}
\]

The mappings from Boolean variables to integer variables are represented as

\[
c = 2 c_l, s = s_l, x = x_l, y = y_l, z = z_l. \tag{33}
\]

Then, the Gröbner Basis is given as

\[
B = \{-z_l + z_l^2, -y_l + y_l^2, -x_l + x_l^2, \]
\[
s_l - x_l - y_l + 2x_l y_l - z_l + 2x_l z_l + 2y_l z_l - 4x_l y_l z_l, \]
\[
c_l - x_l y_l - x_l z_l - y_l z_l + 2x_l y_l z_l, \]
\[
z - z_l, y - y_l, x - x_l, \]
\[
s - x_l - y_l + 2x_l y_l - z_l + 2x_l z_l + 2y_l z_l - 4x_l y_l z_l, \]
\[
c - 2x_l y_l - 2x_l z_l - 2y_l z_l + 4x_l y_l z_l \}. \tag{34}
\]
The normal form of the function \((c + s = x + y + z)\) with respect to \(B\) becomes: \(NF_B(c + s - (x + y + z)) = 0\). Therefore, the formula evaluation returns true.

As shown in Fig. 4, Buchberger’s algorithm sometimes takes long time and requires large memory space. If the set of polynomials consists of linear polynomials, however, the Gröbner Basis calculation is equivalent to Gaussian Elimination [4]. In this case, the computation cost of the proposed method becomes \(O(n^3)\), where \(n\) is the number of integer equations. For many arithmetic circuits, word-level structures are commonly represented by linear integer equations, and thus the proposed verification method can be effective for verifying such word-level functions.

4 Experimental verification

To evaluate the verification times of the proposed method, we designed some kinds of arithmetic circuits: 2-operand adders, multi-operand adders, parallel multipliers, multiply-accumulators, and FIR filters. (Please see [10] for more details on the arithmetic algorithms). In this experiment, we performed the proposed verification using Mathematica (version 5.2) on Intel Core 2 Extreme X6800 2.93 GHz and 2GB memory. For comparison, we also performed the conventional verification technique based on *BMD equivalence checking [7].

Fig. 6 shows the verification time of 2-operand Kogge-Stone adders whose operand lengths are ranging from 4 bits to 72 bits at every 2 bit. The structures of Kogge-Stone adders are fully based on logical expressions. The calculation cost for deriving the Gröbner Basis increases rapidly when the set of polynomials includes such non-linear polynomials. Therefore, we can verify only small adders with formula manipulations. On the other hand, we have a compact *BMD for the above logical expressions, and thus verify more-than 70-bit adders effectively using the *BMD-based equivalence checking.

Fig. 7 shows the verification time of \(n\)-bit \(n\)-operand Wallace trees which are frequently used in actual applications. The number of operands and operand lengths are ranging from 4 bits to 72 bits at every 2 bit. We can easily verify them using formula manipulations since the multi-operand adders consists of many word-level adders. In this experiment, for example, the 72-operand Wallace tree was verified within a few second. On the other hand, the *BMD-based method is not effective for such multi-operand adders since *BMD representation requires to expand the word-level operations into many bit-level operations.

The above observation suggests that the effective verification methods change with the representation types of internal structures. Please remember that the arithmetic algorithms are represented in a hierarchical manner as shown in Fig. 2, and thus each block can be verified independently. Then, we evaluated a hybrid approach combining the *BMD equivalence checking and the formula manipulation. In this experiment, we applied the *BMD-based
method to only lowest-level blocks containing logical expressions and then applied the proposed method to the other higher-level blocks. Figure 8 shows the verification time of unsigned binary parallel multipliers, whose operand lengths are ranging from 4 bits to 72 bits at every 2 bit. This result clearly showed that the hybrid approach improved the verification efficiency for arithmetic circuits including both word-level and bit-level functions.

Table 3 shows the verification times for some arithmetic circuits, where the word-level blocks indicate higher-level blocks whose functional assertion and internal structure are represented by linear integer equations. We evaluated the partial verification times for such word-level blocks in addition to the total verification times. Karatsuba multiplier [11] consists of some smaller-wordlength multipliers, and thus includes more word-level blocks compared with other multipliers. FIR filter considered here is a transposed direct-form type consisting of constant-coefficient multipliers and adders. In this experiment, FIR filter was represented as a combinational circuit equivalent to a 10-tap 32-bit multiplier blocks whose coefficients are selected at random.

We observed here that the proposed formula manipulation is about 2-15 times faster than the conventional *BMD-based method for word-level blocks though it is not adequate to verify digit-level blocks including logical expressions. As a result, the hybrid approach reduced the total verification times significantly compared with the two methods. We successfully verified practical arithmetic circuits, such as Karatsuba multiplier and FIR filter circuit, within 400 seconds under the experimental condition.

5 Conclusion

This paper proposed a formal method to describe and verify arithmetic circuits using symbolic computer algebra. The key idea is to describe arithmetic circuits with integer equations in a hierarchical manner. The proposed representation can be formally verified by formula manipulations based on Gröbner Bases. The experimental result showed that a hybrid approach combining the proposed method and the conventional *BMD-based method can reduce the verification time even for highly-hierarchized arithmetic circuits such as multipliers and FIR filter circuits.

The proposed verification is available for an arithmetic module generator on our website [10]. The generator supports more than one thousand hardware algorithms for 2-operand adders, multi-operand adders, multipliers, constant-coefficient multipliers and multiply accumulators. The generated modules in HDLs are completely verified by the proposed method. For example, a 64x64-bit 3-term multiply-accumulator can be verified within 60 seconds.

References