Residue BDD and Its Application to the Verification of Arithmetic Circuits

Shinji Kimura
Graduate School of Information Science, Nara Institute of Science and Technology
8916-5 Takayama, Ikoma, Nara, 630-01, JAPAN

Abstract — The paper describes a verification method for arithmetic circuits based on residue arithmetic. In the verification, a residue module is attached to the specification and the implementation, and these outputs are compared by constructing BDD's. For the BDD construction without node explosion, we introduce a residue BDD whose width is less than or equal to a modulus. The method is useful for multipliers including C6288.

I. Introduction

With the development of VLSI technology, formal hardware verification becomes important. At present, circuits with medium size can be verified formally with methods using BDD's (Binary Decision Diagrams) [1], [2], [3].

For a class of logic functions, the size of a BDD is proportional to the polynomial of the number of primary inputs. For other logic functions (including multiplication), the size of a BDD is proportional to the exponential of the input size. Several methods have been studied for 16×16 bit multipliers (including C6288 in the ISCAS benchmark) [4], [5] and for larger multipliers [6]. These methods, however, work for limited types of multipliers.

This paper describes a novel method to verify arithmetic circuits based on residue arithmetic, where a number m is represented by a k-tuple of residues with p_1, p_2, ..., and p_k [7]. Using the residue representation, an arithmetic circuit can be verified with each modulus p_1, p_2, ..., and p_k. A residue module for each p_i is attached to the circuit and the specification, and these outputs are compared by constructing BDD's. It can be seen that *the size of the BDD's for outputs is proportional to the number of primary inputs n and the modulus p_i.*

The problem in the construction is that the intermediate BDD's may include node explosion. The paper introduces a method to convert an intermediate BDD to *a residue BDD* the size of which is less than or equal to the polynomial of n and p.

II. Preliminary

A. Logic Function and BDD

A logic function with n variables is a function from $\{0,1\}^n$ to $\{0,1\}$. Logical operations NOT ($\bar{}$), AND ($\wedge$) and OR ($\vee$) are defined in the usual manner.

A BDD is a 5-tuple $((V \cup \{\boxed{0}, \boxed{1}\}, E_0 \cup E_1), X, idx)$, where $(V \cup \{\boxed{0}, \boxed{1}\}, E_0 \cup E_1)$ is a directed acyclic graph (DAG), X is a sequence of variables and idx is a function from V to X. $idx(v)$ is called an index of v. V is a set of internal nodes. E_0 (E_1) is a set of 0-edges (1-edges) each of which is in $V \times (V \cup \{\boxed{0}, \boxed{1}\})$. A node in V has two emanating edges; one is a 0-edge and the other is a 1-edge. For an edge (v, v'), if $idx(v)$ is the i-th variable in X then $idx(v')$ should be j-th variable with $j > i$. X specifies variable ordering.

For each node v in $V \cup \{\boxed{0}, \boxed{1}\}$, a logic function f_v is defined as follows: $f_{\boxed{0}} \stackrel{def}{=} 0$, $f_{\boxed{1}} \stackrel{def}{=} 1$, $f_v \stackrel{def}{=} (\overline{idx(v)} \wedge f_{v_0}) \vee (idx(v) \wedge f_{v_1})$, where $(v, v_0) \in E_0$ and $(v, v_1) \in E_1$.

The width of a BDD is the maximum number of nodes with the same index.

Let $X = (x_{n-1}, x_{n-2}, ..., x_0)$ be a sequence of variables. A BDD is called *levelized* if $idx(v) = x_i$ then $idx(v_0) = idx(v_1) = x_{i-1}$ where v is a node in the BDD, $(v, v_0) \in E_0$ and $(v, v_1) \in E_1$. Note that v_0 may be equal to v_1.

Logic operations on BDD's can be defined as in [1], and the BDD for a logic formula is constructed using the logic operations on BDD's as in [2] and [3].

B. Residue Arithmetic

$(a)_{\mathrm{mod}p}$ is the residue of a with a modulus p, and is one of 0, 1, 2, ..., and $p-1$. A residue representation of a number a with respect to p_1, p_2, ..., and p_k is a k-tuple of residues of a with each p_i, i.e. $((a)_{\mathrm{mod}p_1}, (a)_{\mathrm{mod}p_2}, ..., (a)_{\mathrm{mod}p_k})$. For the uniqueness, p_i and p_j should not have a common divisor other than 1 if $i \neq j$.

This representation can uniquely express $p_1 \times p_2 \times ... \times p_k$ numbers, and the addition, subtraction and multiplication operations can be processed on each residue independently because of the following property :

$$(x \circ y)_{\mathrm{mod}p} = ((x)_{\mathrm{mod}p} \circ (y)_{\mathrm{mod}p})_{\mathrm{mod}p}$$

where x and y are numbers and $\circ$ is an operation [7].

With residue representation, a large number can be manipulated as a k-tuple of small numbers. For example, the product of 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47 and 53 becomes 32589158477190044730 ($> 2^{64}$).

III. Logic Functions Depending on Only the Residue of an Input

We show properties of BDD's representing logic functions for outputs of an arithmetic circuit with a residue module, where these functions depend on only the residue of an input.

A. Logic Functions with One Operand

Definition 1: Let f be a logic function with n variables, p be an integer. Let x_{n-1}, x_{n-2}, ..., and x_0 be input variables

32nd ACM/IEEE Design Automation Conference ®

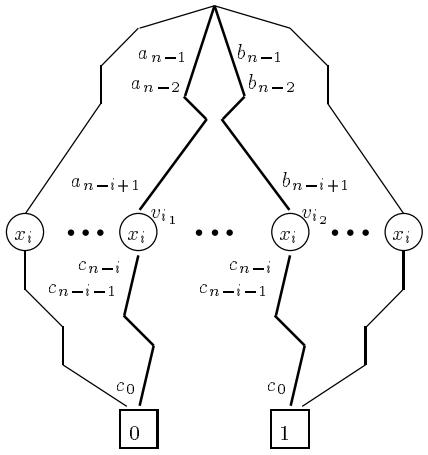

Fig. 1. Property of a residue BDD.

of f, and x_i correspond to 2^i. f depends on only the residue of an input if and only if

$$f(x) = f((x)_{\mathrm{mod}p})$$

Since each variable of f corresponds to an integer, a number is associated to a path on the BDD for f in a natural manner, i.e. the summation of numbers each of which corresponds to an indexed variable of the node whose 1-edge is selected in the path. The number associated to a path is also associated to the end node of the path.

From the definition, the following properties can be proved.

Lemma 1: Let f and g be logic functions with n variables depending on only the residue of an input with a modulus p. Then $\overline{f}$, $f \wedge g$ and $f \vee g$ are also logic functions with n variables depending on only the residue of an input with p.

Theorem 1: Let f be a logic function with n variables depending on only the residue of an input with p. Then the width of the BDD for f is less than or equal to p.

Proof: We can use a levelized-BDD in this proof without loss of generality.

The proof follows the refutation. Let x_{n-1}, x_{n-2}, ..., and x_0 be input variables. We assume that the width of a BDD is greater than p with an index x_i.

From this assumption, there are two different nodes v_{i_1} and v_{i_2} indexed as x_i with the same residue (Fig. 1).

Since v_{i_1} and v_{i_2} are not equivalent logically, there exist paths p_{i_1} and p_{i_2} from v_{i_1} and v_{i_2} respectively associated with the same index on the edges to the different leaves as shown in Fig. 1.

Paths from the root to the leaves including these paths, however, show the contradiction. Since the residues of them are the same, the leaves for them should be should be the same from the definition. Thus the width of the BDD is less than or equal to p. □

We do not use any assumption on the order of variables, and the theorem can be shown for any variable ordering.

B. Logic Functions with Two Operands

Here we consider logic functions with two operands, where each operand denotes an integer.

Definition 2: Let f be a logic function with $2n$ variables. f depends on only the residue of inputs if and only if

$$f(x,y) = f((x)_{\mathrm{mod}p}, (y)_{\mathrm{mod}p})$$

The following corollary holds for any variable ordering.

Corollary 1: Let p be an integer. If f is a logic function with $2n$ variables depending on only the residue of inputs with p, then the width of the BDD for f is smaller than or equal to p^2.

IV. Residue BDD

As shown in the former section, the width of the BDD for a logic function depending on only the residue of an input with p is less than or equal to p (or p^2 for two operand function). The construction of such BDD's, however, may include node explosion, since the width of the BDD's for intermediate functions may be greater than p. Here we show a conversion method from an intermediate BDD to a BDD whose width is less than or equal to p. We also show that the converted BDD's can be used in the construction of the final BDD's.

A. Definition of Residue BDD

Let f be a logic function with n variables, i.e. $f(x_{n-1}, x_{n-2}, ..., x_0)$, and x_i correspond to 2^i. Variables are ordered as x_{n-1}, x_{n-2}, ..., x_0 in the construction of BDD's.

We associate a number and a residue for each path on a BDD as shown before. The number is the summation of numbers each of which corresponds to an index x_i of a node whose 1-edge is selected. Note that variables not appearing in the path is assumed to be 0. From the semantics of BDD's, there exists a choice of 1 for variables not appearing in the path.

Definition 3: Let j_1 and j_2 be numbers associated to nodes v_{i_1} and v_{i_2} respectively. A BDD is a *residue BDD with p* if it satisfies the following condition: for any v_1 and v_2, if $idx(v_1) = idx(v_2)$ and $j_1 \equiv j_2 (\mathrm{mod}p)$ then $v_1 = v_2$.

From the above definition, the following property can be shown.

Lemma 2: The width of a residue BDD with p is less than or equal to p.

Note that the BDD for a logic function depending on only the residue of an input is a residue BDD, but usual BDD's are not residue BDD's.

We show a conversion method from a BDD to a residue BDD. In the conversion, we have only to traverse each BDD node, to keep the firstly traversed node for each variable and each residue, and nodes with the same index and the same residue are merged to the firstly traversed one.

A recursive procedure for the conversion is shown in Fig. 2. Arguments of the procedure are a pointer $ptr1$ to a node of the BDD, a residue k of the path to the node and a modulus p. If $ptr1$ points a leaf, then the procedure returns $ptr1$. Otherwise, the procedure traverses the 0-edge of the node pointed by $ptr1$ with the same residue and the 1-edge of that with $(k + 2^i)_{\mathrm{mod}p}$, where 2^i corresponds to the index of the node pointed by $ptr1$.

Procedures *check* and *register* are used for maintenance of the firstly traversed node with the same index and the

```
BDD_pointer rsd_bdd(ptr1, k, p)
  BDD_pointer ptr1;
  int k, p;
{
  int index, new_k;
  BDD_pointer ptr2, low, high;

  if (ptr1 == FALSE || ptr1 == TRUE)
          return(ptr1);

  index = ptr1->index;
  ptr2 = check(index, k);
  if (ptr2 != NULL) { return(ptr2); }

  new_k = (k + get_num(index)) % p;

  low  = rsd_bdd(ptr1->low, k, p);
  high = rsd_bdd(ptr1->high, new_k, p);
  ptr2 = gen_node(index, low, high);

  register(index, k, ptr2);
  return(ptr2);
}
```

Fig. 2. A converter to a (default 0) residue BDD.

same residue. *Get_num* denotes a procedure to obtain a number corresponding to an index. *Gen_node* denote a procedure to generate a new BDD node.

The converted BDD is unique and the function represented by the converted BDD is also unique. We denote $[f]_{\mathrm{mod}p}$ for the converted function.

This conversion procedure can easily be extended to manipulate BDD's for logic functions with two operands. The maximum width of a converted BDD is less than or equal to p^2 for a modulus p.

As mentioned above, the conversion procedure assumes *0* for variables not appearing in the path, i.e. 0-default traversal. It is easy to make a procedure assuming *1* for variables not appearing in the path (1-default traversal). There may exist other traversal methods. A residue BDD generated with 0-default traversal is not the same as the one with 1-default traversal. This corresponds to the loss of information by restricting the width of BDD's.

If we use a levelized-BDD in the traversal, we need no assumption. The conversion, however, is too restrictive, since any BDD becomes the BDD representing a logic function depending on only the residue of an input.

In the following, we consider 0-default traversal, but the properties can be shown for 1-default traversal.

B. Properties of Residue BDD

Here we show properties of residue BDD's generated by the above conversion method.

Lemma 3: Let x_i and x_j be input variables.

1. $[x_i]_{\mathrm{mod}p} = x_i$
2. $[x_i \wedge x_j]_{\mathrm{mod}p} = x_i \wedge x_j$
3. $[x_i \vee x_j]_{\mathrm{mod}p} = x_i \vee x_j$

Proof: The above properties can be shown from the fact that the width of each BDD is 1. □

Lemma 4: Let f, g and h be logic functions with n variables, and $\circ$ be a logic operation. If $f = g \circ h$ then

$$[f]_{\mathrm{mod}p} = [[g]_{\mathrm{mod}p} \circ [h]_{\mathrm{mod}p}]_{\mathrm{mod}p}$$

Proof: Let v be a node in the BDD for $[f]_{\mathrm{mod}p}$ with an index x_i, and j be a residue of a path to v. In this proof, a node v is identified as the function represented by a sub-DAG starting from v. Since v is a result of $g \circ h$, there exist v_1 in the BDD for g and v_2 in the BDD for h, and v is the result of $\circ$ on BDD's starting from v_1 and v_2. Note that there should be paths to v_1 and v_2, where the choices of 0-edge or 1-edge for each index on paths are the same and the residues for these paths are the same.

Since v is firstly traversed in the BDD for f, v_1 and v_2 are also firstly traversed in the BDD's for g and h respectively. Thus v_1 and v_2 are included in the BDD's for $[g]_{\mathrm{mod}p}$ and $[h]_{\mathrm{mod}p}$, and v is in the BDD for $[g]_{\mathrm{mod}p} \circ [h]_{\mathrm{mod}p}$. Moreover, since v is firstly traversed, v is in the BDD for $[[g]_{\mathrm{mod}p} \circ [h]_{\mathrm{mod}p}]_{\mathrm{mod}p}$.

On the other hand, it is easy to see that a node v' in the BDD for $[[g]_{\mathrm{mod}p} \circ [h]_{\mathrm{mod}p}]_{\mathrm{mod}p}$ is also in the BDD for $[f]_{\mathrm{mod}p}$. □

Lemma 5: Let f be a logic function depending on only the residue of an input with p. Then $[f]_{\mathrm{mod}p} = f$.

Proof: From the assumption on f, $f(x) = f((x)_{\mathrm{mod}p})$, where x is a sequence of x_{n-1}, x_{n-2}, ..., x_0. When we fix up the upper k bits of x to a constant, then

$f(a_{n-1}, a_{n-2}, ..., a_{n-k}, x_{n-k-1}, ..., x_0) =$

$f((a_{n-1}a_{n-2}...a_{n-k})_{\mathrm{mod}p}, x_{n-k-1}, ..., x_0)$

Thus the paths with the same residue reach to the same node. □

From Lemma 5, the following theorem can be proved.

Theorem 2: Let f be a logic function depending on only the residue of an input with p. Then we can use residue the BDD's for intermediate results in the construction of the BDD for f.

Proof: The theorem is shown by applying Lemmas 4 and 5 recursively in the construction of the BDD for f. □

From the theorem, the BDD for f can be constructed using BDD's each of whose width is less than or equal to a modulus p. It is easy to extend the theorem for logic functions with two operands, where the width is less than or equal to p^2.

V. Verification of Arithmetic Circuits Using Residue BDD

Here we show a verification method for arithmetic circuits using residue BDD's.

A. BDD Construction Using Residue BDD

The following is a procedure to construct a BDD from a logic formula using residue BDD's.

1. Construct the constant 0-leaf and the BDD's for primary input variables.
2. Parse a logic formula and sort logical operations in the formula.
3. Process the logic operations on BDD's and convert the result BDD to a residue BDD with the obtained order.
4. Convert the final BDD's to levelized-BDD's, and then convert the levelized-BDD's to residue BDD's.

B. Verification of Arithmetic Circuits

Here we consider equivalence check of a specified logic function and the logic function implemented by a logic circuit. As mentioned above, we verify the circuit with a modulus p_1, a modulus p_2, ..., and a modulus p_k, where p_i and p_j have no common divisor other than 1. If the circuit is equivalent to the specification for any p_i, then the circuit is equivalent to the specification.

We attach a residue module to a specification and an implementation, then construct the BDD's for outputs of the modified specification and the modified implementation using residue BDD's.

Note that there are several methods converting to residue BDD's depending on traversal methods. If we try to verify with all traversal methods, then the verification is perfect. Otherwise the difference in two functions may not be detected, since residue BDD's lose the original information. We now try to clarify the limitation of the residue BDD based verification. At present, 0-default traversal and 1-default traversal seem to cover almost all errors.

In the following, we show verification of multipliers. We compare an adder array type multiplier, an $O(\log n)$ multiplier (a Wallace-tree type multiplier) and C6288. We have implemented and evaluated our programs in C on SUN SS10/51 (50 MHz SuperSPARC, 1MB SuperCache, 64 MB Main Memory).

In our experiment, we use generators for adder-array and Wallace-type multipliers and a generator for a residue function with p. In the construction of a BDD, primary inputs are ordered as a_{n-1}, a_{n-2}, ..., a_0, b_{n-1}, b_{n-2}, ..., b_0, and the maximum size of BDD nodes is restricted to 1,500,000.

Table I shows CPU time (in seconds) to construct output BDD's and the number of nodes for primary outputs for each modulus. The execution time is evaluated with a procedure using 0-default traversal in the conversion to residue BDD. The execution time with a procedure using 1-default traversal is almost the same.

We checked these functions with 10 modulus: 2, 3, 5, 7, 11, 13, 17, 19, 23 and 29. The product of these numbers is 6469693230, which is greater than 2^{32} (= 4294967296). These functions are shown to be the same.

The execution time and the number of nodes become large when the modulus becomes large. This is because the node size is proportional to $n \times p^2$ for the number of primary inputs n and a modulus p.

We made several erroneous C6288 circuits inserting a gate error, where some logic gate is changed to another logic gate, and compared the correct and erroneous circuits. In the experiment, all errors are detected and the difference is usually detected with rather small residue such as 7 or 11. There are errors which cannot be detected with 0-default traversal.

TABLE I

The comparison of 16-bit multiplier.

mod.	time (sec.)			Output
	AA	WA	C6288	nodes
29	5517	3985	4266	25713
23	2198	1754	1700	15931
19	1197	917	887	12323
17	849	634	636	5278
13	351	241	216	4479
11	225	158	137	3840
7	83	54	37	800
5	48	35	22	596
3	30	23	14	207
2	12	11	6	2

AA: Adder Array Type; WA: $\log n$ Type
Maximum node is restricted to 1,500,000

VI. Concluding Remarks

We propose a residue BDD and its theoretical framework. The width of a residue BDD is less than or equal to a modulus p; thus, a residue BDD has no node explosion.

We also show a verification method based on residue arithmetic and apply the method to verify multipliers including C6288. Verification requires several thousand seconds of CPU time, but we believe the execution time problem can overcome with coding effort and by using parallel computing environment.

From our experiment, we have found that the difference between a correct and erroneous circuits can be detected with small modulus.

Residue BDD's are interesting from the theoretical point of view. The class of residue BDD's are sub-classes of BDD's whose nodes are proportional to the polynomial of the size of variables.

Further studies are needed on the limitation of residue BDD's and an application of residue BDD's to sequential circuit verification.

Acknowledgment I would like to thank Professor Naofumi Takagi of Nagoya University for comments on residue arithmetic. I also would like to thank Mr. Takeo Kunishima and Mr. Kouiti Nagami for reviewing this paper. I wish to thank Professor Katsumasa Watanabe and the members of Watanabe Lab. of Nara Institute of Science and Technology for their discussions on this research. Thanks are due to anonymous reviewers for their comments that helped to refine this paper.

References

[1] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation", *IEEE Trans. on Comput.*, vol. C-35, no. 8, pp. 677–691, Aug. 1986.

[2] S. Minato, N. Ishiura, and S. Yajima, "Fast Tautology Checking Using Shared Binary Decision Diagram –Benchmark Results–", in *Proc. IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, Nov. 1989, pp. 580–584.

[3] R. E. Bryant, "Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams", *ACM Computing Surveys*, vol. 24, no. 3, pp. 293–318, Sep. 1992.

[4] Jerry R. Burch, "Using BDD to Verify Multipliers", in *Proc. of 28th DA Conference*, 1991, pp. 408–412.

[5] J. Jain, J. Bitner, J. A. Abraham, and D. S. Fussell, "Functional Partitioning for Verification and Related Problems", in *Proc. of the 1992 Brown/MIT Conference*, 1992, pp. 210–226.

[6] Yirng-An Chen Randal E. Bryant, "Verification of Arithmetic Functions with Binary Moment Diagrams", Tech. Rep. CMU-CS-94-160, Carnegie Mellon University, 1994.

[7] Norman R. Scott, *Computer Number System & Arithmetic, Chapter 7.6*, Prentice Hall, 1985.