Formal Verification Method for Combinatorial Circuits at High Level Design

Junji Kitamichi, Hiroyuki Kageyama and Nobuo Funabiki

Division of Informatics and Mathematical Science
Graduate School of Engineering Science, Osaka University
Osaka, JAPAN

{ kitamiti, kageyama, funabiki }@ics.es.osaka-u.ac.jp

abstract **In this paper , we propose a formal verification method for combinatorial circuits at high level design. The specification is described by both integer and Boolean variables for input and output variables, and the implementation is described by only Boolean variables. Our verification method judges the equivalence between the specification and the implementation by deciding the truth of Presburger sentence. We show experimental results on some benchmarks, such as 4bit ALU, multiplier, by our method.**

1. Introduction

With the rapid increase of size and complexity in VLSI systems, the formal verification method has become essential for their correct designs. For this goal, we have proposed a formal verification method for high level circuits design, and developed a verification support system to evaluate our method by experiments[8].

In this paper, we propose a verification method for combinatorial circuits at high level design. The specification is described by both integer and Boolean variables for input and output variables. The implementation is described by only Boolean variables for input and output variables. Our verification method judges the equivalence between the specification and the implementation by deciding the truth of Presburger sentence, which consists of integers, variables and the operators belonging to { $\wedge$, $\vee$, $\neg$, + , - , = , > , $\forall$, $\exists$ }. We describe the results of some verification experiments.

Our verification method may be similar to the method [5, 6] with BMDs(Binary Moment Diagrams). However, an integer variable is treated as one variable without decomposed into a set of Boolean variables in [5, 6]. Our verification method is more suitable for high level formal verification.

2. Proposed Verification Method for Combinatorial Circuits

In this session, we first describe the style of specification and implementation of combinatorial circuits. Then we propose the method to decide the equivalence between specification and implementation. We explain them by using an example 74382, a 4bit ALU(standard TTL).

2.1 Specification

In specification, 74382 has six inputs, $A, $B, $Cnspec, S0, S1, S2, and has three outputs, $Fspec, $Cn4spec and OVR. $A and $B are integer inputs, $Cnspec is a carry-in, and S0, S1 and S2 are function selecting inputs. $Fspec , $Cn4spec and OVR are an integer data output , a carry-out and an over-flow flag, respectively. For convenience, we use the prefix "$" as the integer variable, and use the suffix "*spec*" or "*imp*" as the variable used in specification or implementation, respectively. We show the 74382 specification in Figure 1.

```
:
^
  ((¬ S2 ^ S1 ^ ¬ S0) imply (
        if $Cnspec = 1 then (
             if $A - $B >= 0 then (
                  ( $Fspec = $A - $B) ^
                  ( $Cn4spec = 1) ^
                  ( not OVR ))
             else (
                  ( $Fspec = $A - $B + 16) ^
                  ( $Cn4spec = 0) ^
                  ( OVR )))
        else
  :
  ))
```

Figure 1 Specification of 74382

Each function, such as *and*, *or*, *addition*, is selected by function selecting inputs [S0, S1, S2]. For example, if S0 = False, S1 = Truth and S2 = False, then the subtraction from $A to $B is performed .

2.2 Implementation

In implementation, 74382 has 12 Boolean inputs A0, .., A3, B0, .., B3, Cnimp, S0, S1, S2, and has seven Boolean

outputs F0, .., F4, Cn4imp and OVR. Inputs A0, .., A3 , B0, .., B3 and Cnimp correspond to $A , $B and $Cnspec, respectively. Outputs F0, .., F4 and Cn4imp correspond to $Fspec and $Cn4spec respectively. We get the implementation by the synthesis tool with some modifications. We show the 74382 implementation in Figure 2. For example, output F0 is specified by the logic function which consists of inputs [S0, S1, S2] and Boolean variables such as _TMP009. We use the prefix "_TMP" as an internal node.

```
:
∧ (
 F0 = CINimp ∧ ¬ _TMP009 ∧ ¬ _TMP011
        ∧ S0 ∧ ¬ S2
   ∨ CINimp ∧ ¬ _TMP009 ∧ ¬ _TMP011
        ∧ S1 ∧ ¬ S2
   ∨ ¬ _TMP038 ∧ ¬ S0 ∧ ¬ S1
   ∨ ¬ _TMP038 ∧ S2
   ∨ ¬ CINimp ∧ ¬ _TMP038 )
∧ (
:
```

Figure 2 Implementation of 74382

2.3 Definition of Equivalence between Specification and Implementation

We define the equivalence between the specification and the implementation. If the specification and the implementation have the same output value for any common input value, they become equivalent. But corresponding inputs or/and outputs may have different data types such as integer and Boolean. We introduce i-to-b and b-to-i functions (or relations) , which translate integer data into Boolean data and Boolean into integer, respectively. We show these functions in Figure 3.

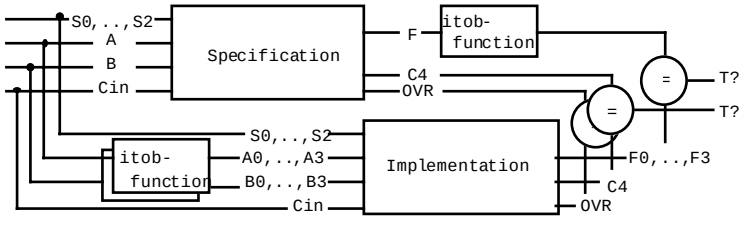
We show the outline of verification used these functions in Figure 4. For example, Figure 4 (a) verifies whether Boolean outputs in specification and implementation are equivalent, and that the results of i-to-b function for integer variables in specification and Boolean outputs in implementation are equivalent , for each common value of all inputs.

There are four deferent combinations on the data type (integer or Boolean) for inputs and outputs of specification and implementation to be verified. If one of them is verified, the implementation becomes equivalent to the specification.

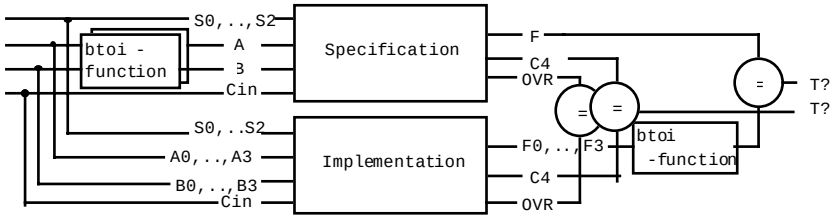
```
( A0 = ( $A = 1 ∨ $A = 3 ∨ ... ∨ $A = 15))
  ∧
( A1 = ( $A = 2 ∨ $A = 3 ∨ ... ∨ $A = 15))
  ∧
( A2 = ( 4 <= $A ∧ $A <= 7
    ∨ 12 <= $A ∧ $A = 15))
  ∧
( A3 = ( 8 <= $A ∧ $A <= 15))
  (a)Function i-to-b from $A to [A0..A3]

     if ¬ A3 ∧ ¬ A2 ∧ ¬ A1 ∧ ¬ A0
    then $A = 0
 else if ¬ A3 ∧ ¬ A2 ∧ ¬ A1 ∧   A0
    then $A = 1
      :
 else if   A3 ∧   A2 ∧   A1 ∧ ¬ A0
    then $A = 14
 else if   A3 ∧   A2 ∧   A1 ∧   A0
    then $A = 15
  (b)Function b-to-i from [A0..A3] to $A
```

Figure 3 i-to-b and b-to-i Functions



(a)Equivalence for all input S0,S1,S2, and integer *A,B,Cin*



(b)Equivalence for all Boolean input S0,S1,S2,A0,..A3,B0,..B3,Cin

Figure 4 Outline of Verification for 74382

The expression corresponding to each type of verification to be proved is given in (1):

```
∀ All variable
 ( (valid situations or not invalid situations)
   ∧  Specification
   ∧  Implementation
   ∧ {if needed}
       b-to-i functions for
          Boolean inputs or/and outputs
    ∧ {if needed}
     i-to-b functions for
          integer inputs or/and outputs)
     imply
    for each outputs
      ∧  if ¬  (don't care situations) then
      output in Specification
         = output of Implementation)          (1)
```

For example, for input $A, (0 <= $A <= 15) is a valid situation. We refer to "don't care situations" in **4.**

3. Verifier

We describe our verifier of deciding the truth of the sentences such as expression (1) in **2 .**

The Presburger sentence consists of integer variables, Boolean variables , operators " ∧ , ∨ , ¬ (not) , (,) , +, -, ∀ , ∃ , = , <", and has no free variable. Cooper's algorithm[4] can decide the truth of Presburger sentences. The notations ,such as *imply, if then, if then else,* are defined as a sequence of primitive operators.

We have used two verifiers. One verifier has BDD-like data structure, by extended to hold the integer expressions[9]. We denote this verifier as *Sys1.* In the data structure in Sys1,

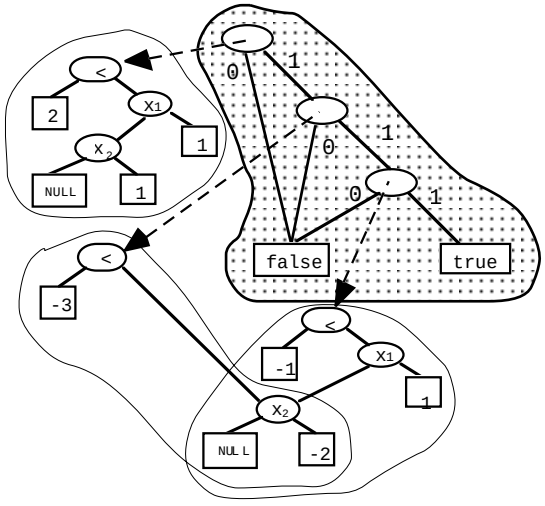


Figure 5 An example of our proposed data structure

terms (or numerical parts) are expressed in list and logical parts are expressed in BDDs[1]. The expression with integer variables is treated as one Boolean variable node. The value of this Boolean variable would be evaluated as truth or false when the numerical parts has no free integer variable (that is, the last integer variable is eliminated). We explain our proposed data structure with an example sentence and its corresponding data structure.

Figure 5 denotes a diagram which corresponds to a sentence $(2 < x_1 + x_2) \wedge (2\ x_2 < 3) \wedge (-1 < x_1 - 2\ x_2)$. The meshed area expresses the sentence (or logical part) by BDD. The BDD in this area is the same as BDD corresponding to $a_1 \wedge a_2 \wedge a_3$, which is obtained such that each numerical part of the sentence is transformed into a_1, a_2 and a_3. The internal nodes of this data structure have three pointers to *then node, else node* and *numerical node* which points the numerical part. The numerical part holds the constant integer on the left side and the list of integer variables and integer constants on the right side. The term $(-1 < x_1 - 2\ x_2)$ is expressed as a node holding -1 on the left side and $x_1 - 2\ x_2$ on the right side.

The storage which should hold the constant value in BMD[6] is replaced with NULL in this data structure. The reason is that all lists of integer variables which are generated in the operation ∀ or ∃ , (except for constant integers) are shared.

We implemented the library of each operation for the proposed data structure by existing the BXD library[7]. The BXD library includes the basic operations for BMD and BDD , but does not include the operations ∀ and ∃ for integers. We added the operations for ∀ , ∃ and the other operations which aren't included in the BXD library.

The order of variables in the diagram decides the size of generated data structure. In our library, the variables order is fixed by the order in which the variables appear in the syntax analysis of input sentence.

Another verifier is able to decide the truth of subclass Presburger sentences given by the prenex normal form of only ∀ , without ∃ . They may include the expression(1) in **2 .** or a variety of sentences for circuit verifications[8][10]. We denote this verifier as *Sys2.* Sys2 decides the truth of the sentence ∀ x1 ∀ x2... ∀ xn F(x1,x2,..,xn). It adapts some techniques for fast operation and of memory saving, such as determination the order of ∀ operations. But Sys2 does not share the common internal data unlike Sys1 .

4. Experiments

4.1 Verification of 74382

We describe the verification result of 74382 in **2.** We show the result in Table 1.

Sys2 needs the same 0.3 seconds for four types of

verification. Sys1 takes longer time for the verification of Boolean inputs and Boolean outputs. The reason is that the verification by Sys1 requires three b-to-i functions which result in three large complete binary sub-trees and that Sys1 takes much time and space to treat such data structure.

Then, we show the verification result of the equivalence between two different implementations *imp1* and *imp2* of 74382. The BDD data structures corresponding to imp1 and imp2 are constructed in Sys1, where only pointers to the top node corresponding BDD data structure are compared.

4.2 Verification of Multiplier

We describe the verification results of 4bit and 5bit multipliers. These circuits use a sign-magnitude format as input and output data. Integer "0" is expressed in two ways such as "FFFF" or "TFFF" in the case of 4bit. Thus when data output is "0", sign output does not need to be considered. We use *don't care situations* expression (2):

```
      :
if ¬  ( $DataOutput in Specification = 0 ) then
   /* or  ¬ ( $inputA = 0 or $inputB = 0 ) */
            SignFlag in Specification
                = SignFlag of Implementation)
      :
```

(2)

4.3 Verification of Output nan-res in fp-add

We are challenging the verification of **fp-add** in HLSynth95, floating point adder. We regard as specification the relation from inputs to outputs, and as implementation the relation among inputs, outputs, internal terminals from VHDL description. All numerical terminals in specification and implementation such as exponent and mantissa are treated as integer.

Using our method in **2 .**, the verification for only output **nan-res** can be performed. Sys1 can verify more quickly than Sys2.

	Input	Output	# of Bool Variables	# of Int Variables	Sys1	Sys2
74382						
	Int	Bool	56	4	1.6	0.3
	Bool	Int	56	4	3.1	0.4
	Int	Int	56	4	1.6	0.3
	Bool	Bool	56	4	6.7	0.3
	(Imp1 = Imp2)		81	0	1.7	0.3
4mult	Int	Bool	40	4	N/A	181.3
	Int	Int	40	4	N/A	724.5
	(Imp = Imp)		71	0	0.7	7.6
5mult	Int	Bool	60	4	N/A	5816.1
	(Imp = Imp)		109	0	12.8	6823.8
HLSynth95 FP_ADD						
	Output	nan-res	45	39	207.4 (sec.)	256.1 (sec.)
					PentiumII 300MHz	128MBmem

Table 1 Results of verifications

5. Conclusions

We propose the formal verification method for high-level combinatorial circuits and show the result of verification experiments using our verifier.

Sys1 is superior to Sys2 in the verification of **nan-res**, and there is more prospect in Sys1 for the high-level verification of combinatorial and sequential circuits. Now we have been improving Sys1 and will apply it to more large and high levels, including the sequential circuits verification.

References

[1] S. Minato: “Binary Decision Diagrams and Applications for VLSI CAD”, Kluwer Academic Publishers, (1996).

[2] G.D.Hachtel and F.Somenzi:“Logic Synthesis and Verification Algorithms”, KAP(1996).

[3] T. Higashino, J. Kitamichi, T. Kenichi, “Presburger Arithmetic and its Application to Program Developments”, Computer Software, Vol.9, No.6(1992) (In Japanese)

[4] D.C.Cooper: “Theorem Proving in Arithmetic without Multiplication”, Machine Intelligence, No.7(1972).

[5] R.E. Bryant and Y.-A. Chen:“Verification of Arithmetic Functions with Binary Moment Diagrams”, Technical Report CMU-CS-94-160, School of Computer Science, Carnegie Mellon University, (1994).

[6] R.E. Bryant and Y.-A Chen: “Verification of Arithmetic Circuits with Binary Moment Diagrams”, 32nd DAC, pp.535-541 (1995).

[7] BXD Package Home Page, http://www.cs.cmu.edu/afs/cs.cmu.edu/usr/yachen/www/bxd.html.

[8] J.Kitamichi, S.Morioka, T.Higasino and K.Taniguchi:“Automatic Correctness Proof of Implementation of Synchronous Sequential Circuits Using Algebraic Approach”, Proc. of the 1994 Conference on Theorem Provers in Circuit Design (TPCD94).Vol.901 of LNCS, pp.165-184, Springer Verlag (1995).

[9] J. Kitamichi, N. Funabiki and S. Nishikawa, “Proposal of Data Structure for Presburger Arithmetic and its Application to Circuits Verification”, 1997 Int. Symp. on Nonlinear Theory and its Applications, Vol.2, pp.1233-1236(1997).

[10] T.Amon, G.Borriello, T. Hu and J.Liu, “Symbolic Timing Verification of Timing Diagrams using Presburger Formulas”, Int. Conf. 34nd DAC(1997).