Abstract

In formal verification, the first step is to build a formal model. Most of the academic tools represent the designs in BLIF or BLIF-MV network, while cannot directly take Verilog source code as an input. For instance, VIS depends on the translator vl2mv to compile Verilog into the intermediate format BLIF-MV first, and then extracts the FSM (Finite State Models) from it instead of the direct Verilog source code. However, Verilog and VHDL are the most common used hardware description languages for large scale integrated circuits. It is necessary if the source code could be used directly, rather than an intermediate format. We notice that synthesizable Verilog is well-structured which can be represented in SSA (Static Single Assignment) form. In this paper, we describe a method that extracts formal model directly from Verilog source code by converting the code to SSA form. With this method, we have implemented a prototype system that extracts formal model of Verilog programs. We also prove the conversion function-preserving by translating SSA back to Verilog and checking its equivalence with the primary design. To ensure correctness and robustness, we have tested our system with ISCAS and picoJava-II, a specification of microprocessor, and the experimental results on these realistic circuits show that the conversion procedure is function-preserving. Moreover, we take it as the front-end of CEC (Combinational Equivalence Checking) and demonstrate the formal model extracted with our method can be exploited efficiently in CEC.

Keywords: model extraction, SSA, formal extraction, combinational equivalence checking

1. Introduction

One of the crucial steps in formal verification [1] is constructing a formal model of the system being verified. However, creating an accurate model of the design is a difficult problem. Usually formal models are constructed from informal design documents. But such documents are often ambiguous and do not reflect the actual system completely: many decisions taken at the time of implementations are not documented. In such cases, the verification results are questionable. While, building verification models directly and automatically from the code can avoid those problems above.

Most of academic tools represent designs in BLIF or BLIF-MV [2] networks, and cannot take an input directly from Verilog source code. For example, VIS [3] depends on the translator vl2mv [4] to compile Verilog into intermediate format BLIF-MV first, and then extract formal model from these files instead of the direct Verilog source code. There are only few tools that can directly take Verilog source code as an input, but extracting formal model like this brings convenience indeed. So we come up with the idea of extracting formal model directly from Verilog source code.

The Static Single Assignment (SSA) [5, 6] form is an Inter-mediate Representation (IR) that adopted by a wide range of compilers like GCC (since GCC 3.0 Release), Intel CC, and LLVM. The SSA representation proves very efficient as it allows efficient implementations of data flow analysis and optimizing transformations, such as constant propagation, induction variable analysis, dead code elimination, global value numbering [7, 21] and partial redundancy elimination [8, 22]. The SSA transformation, originally crafted for sequential programs manipulating scalars, has been extended to parallel programs as well as the sequential programs with arrays. [9, 10]
The Combinational Equivalence Checking (CEC) [11] problem is a formal verification method that targets to check whether two given combinational circuits—in case the circuit contains no storage elements (flip-flops or latch)—are functional equivalence. We notice that the synthesizable [12] Verilog is well-structured and can be represented in SSA form, and the SSA form of combinational Verilog just defines the CEC constraints. In this paper, we present a straightforward method using the SSA form to mechanically extract models directly from the source code of Verilog design, which can be served as the front-end of formal verification tool. Though the extracted model is used as the front-end of CEC, our method for extracting formal model is not limited to combinational circuits. It is also suitable for sequential circuits.

The rest of this paper is organized as follows. In section 2 we briefly review the backgrounds of SSA conversion. Section 3 presents the implementation details of conversion to and out of SSA for a subset of synthesizable Verilog. Section 4 reveals the solution how we can guarantee the formal model extracted from Verilog source code function-preserving. Section 5 presents the experimental results of the SSA conversion and Section 6 introduces some related works we have done for building our prototype system. Section 7 is a conclusion of this paper.

2. Background

The CEC process consists of several steps. First, the reference and implementation circuits are parsed by the front-end lexical and syntax analyzer. Then after elaboration process, the formal model defining the system constraints is extracted separately, and finally a Miter [13] circuit is constructed and then handed over to the back-end constraint solver.

In this chapter, we will give a brief introduction about the first few steps from lexical analyzing to SSA conversion.

2.1. Lexical/Syntax Analyzing

Lexical and Syntax analyzing are the first two phases in compiling. The lexical analyzer is a pattern matcher that groups the input streams into tokens according to the regular expression patterns. Afterwards, the syntax analyzer reads the token streams and builds an internal syntactic representation of the program. The most common IR is the Abstract Syntax Tree (AST).

There are tools developed for generating analyzers purpose. For example, the Lex and YACC are free and open source tools for generating lexical analyzer and syntax analyzer [14].

2.2. Elaboration

In the Verilog and VHDL communities, elaboration [12] is a process of finding identifier definitions, resolving parameters overridden by instantiation and defparam statements, and generating fully expanded code.

After a Verilog design has been parsed, and before simulation or synthesis begins, the design needs the modules being instantiated linked to the modules being defined, the parameters propagated among the various modules, and hierarchical references resolved. These are all solved during the elaboration phase.

In our implementation, the elaboration phase revises the AST to meet the needs. The elaboration phase is a necessary pre-process for the SSA conversion.

2.3. SSA Form

In this section we give a brief review of the SSA form representation. SSA form is an IR that allows simple yet efficient optimizations and analyses.

Definition 1 (SSA Form [5, 6]) A program is in SSA form if every variable is assigned exactly once, while each use of a variable is renamed to match the nearest definition.

Programs are seldom in SSA form initially, because variables tend to be assigned multiple times, not just once. SSA-based representation modifies the program so that every variable is assigned only once and when the same variable is assigned again, a new version of this variable is created. Different
versions of the same variable can be distinguished by subscripting the variable name with a version number.

A simple example of SSA transformation of sequential program is depicted in table I.

<table>
<thead>
<tr>
<th>Code</th>
<th>SSA Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p = a + b )</td>
<td>( p_1 = a + b )</td>
</tr>
<tr>
<td>( q = p - c )</td>
<td>( q_1 = p_1 - c )</td>
</tr>
<tr>
<td>( p = q \times d )</td>
<td>( p_2 = q_1 \times d )</td>
</tr>
<tr>
<td>( p = e - p )</td>
<td>( p_2 = e - p_2 )</td>
</tr>
<tr>
<td>( q = p + q )</td>
<td>( q_2 = p_2 + q )</td>
</tr>
</tbody>
</table>

Table I. SSA Form of Sequential Program

In case of branches and loops, a variable may have multiple versions at the joint, and the joint has two or more edges. To solve them, the SSA representation introduces new assignment statements at the joint of the program, and uses the \( \varphi \) (phi) function to distinguish values of variables transmitted on distinct incoming control flow edges.

**Definition 2\( (\varphi \) function\[5, 6\])**

A \( \varphi \) (phi) function has the form \( U = \varphi (X_1, X_2, X_3, \ldots) \) to distinguish values of variables transmitted on distinct incoming control flow edges, where \( X_1, X_2, X_3, \ldots \) are variables and the number of operands \( X_1, X_2, X_3, \ldots \) is the number of control flow predecessors of the point where the \( \varphi \) function occurs.

Take the branch code in table II for example. The variable \( x \) has two versions, \( x_1 \) and \( x_2 \), after the if-then-else statement. Therefore, an \( \varphi \) function is needed to merge these two versions. As a result, a new version of \( x \) is introduced, and the assignment is placed at the joint of the if-then-else statement. In this simple test case, the \( \varphi \) function is represented by the conditional operator \( (\text{condition} ? \text{result1} : \text{result2}) \).

<table>
<thead>
<tr>
<th>Code</th>
<th>SSA Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (flag)</td>
<td>if(flag)</td>
</tr>
<tr>
<td>( x = -1; )</td>
<td>( x_1 = -1; )</td>
</tr>
<tr>
<td>else</td>
<td>else</td>
</tr>
<tr>
<td>( x = 1; )</td>
<td>( x_2 = 1; )</td>
</tr>
<tr>
<td>( y = x * a; )</td>
<td>( y = x_1 * a; )</td>
</tr>
</tbody>
</table>

Table II. SSA Form of Branching program

There are mature algorithms for computing SSA form, while this is beyond the scope of this paper.

3. **Formal Model Extraction Implementation**

As depicted in Figure 1, our front-end Verilog parser contains two processes: lexical/syntax analyzing and elaboration. Through the first process, we can get the IR of Verilog source code, which we call an Abstract Syntax Tree. Elaboration is executed on the basis of AST and usually needs several iterations to complete the whole process. This phase is a necessary pre-process for the SSA conversion. Extraction of formal model is crucial in combinational equivalence checking and it is in this phase that our Verilog model is transformed into SSA form which can be exploited by back-end solver conveniently. After extraction of formal models for the two Verilog models separately, a Miter circuit is constructed and then handed over to the back-end constraint solver. A back-end constraint solver is usually a SAT solver or SMT solver that decides whether the clauses of a given Boolean formula can be assigned in such way to make the formula true. In our prototype system we adopt a solver based on SMT called STP (Simple Theorem Prover) as our back-end constraint solver.

In this paper, we will not explain all main elements of our model, but only focus on formalization for the extraction of formal models from Verilog. To illustrate our algorithm, we give a simple version of formal syntax, which defines a subset of synthesizable Verilog [12], and it is enough to reveal our approach.
### 3.1. Formal Syntax

The formal syntax given above is not strictly following IEEE standard, and only contains representative structures of synthesizable Verilog.

In Verilog designs, one module can instantiate another one, through the module instantiation structure (4) defined above. The module without instantiated by any other modules is termed as top module, which could be several in a common design.

One special structure is the procedural assignment, or precisely the event control statement (12) in our paper. Other structures of procedural assignment, like delay control statement, is not defined here. The procedural statement only appears in `always` block (3). The `stmt-item` is executed when a specified transition (rising/falling edge) on a signal happens if the event-expr is Edge-sensitive, or the event-expr evaluates true if Level-sensitive. The assignment in procedural assignment (12) could be blocking or non-blocking assignment. In addition, the Left Hand Side (LHS) variables of assignment in procedural assignment should be register-type.

In our paper, we assume the design is clear. That is, blocking assignment only appears in combinational `always` (level triggered) while non-blocking only appears in clocked `always` (edge triggered). We also assume that the loop construct (10) is constant bounded, and there is no expression construct with side effect.

Various kinds of module-item it is, however, the same functionality can be implemented with only continuous assign (2) and clocked always. The module instantiated could be flattened, and the gate instantiation can be represented by continuous assign, while the combinational always can be transformed to equivalent continuous assign statements. In this way, there are only two kinds of assignments left in Verilog after conversion: the blocking assignment in continuous assignment and the non-blocking assignment in clocked always. The equivalent structure is listed below.
Table III. Equivalent structure

<table>
<thead>
<tr>
<th>Original structure</th>
<th>Equivalent structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>and (a, b, c);</td>
<td>assign c = a &amp; b;</td>
</tr>
<tr>
<td>always @(a or b)</td>
<td>assign c = a + b;</td>
</tr>
<tr>
<td>c = a + b;</td>
<td></td>
</tr>
<tr>
<td>if (e)</td>
<td>assign c = e? a: b;</td>
</tr>
<tr>
<td>c = a;</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td>c = b;</td>
<td></td>
</tr>
</tbody>
</table>

3.2. Convert to SSA

Branching structure here is the conditional statement (13) (case statement (14) will elaborate to nested conditional statement first), which is the only one needs to place \( \phi \) function in. As there are only two branches, it is straightforward to use the question-mark expression \(? :?\) to represent the \( \phi \) function.

To meet the needs of uniqueness of variables assigned in SSA form, a global counter is used to generate unique number (UID) for each assignment (instead of for each id).

The main ideal of SSA transformation is to extract assignments info (we refer it to dataflow) from various syntax structures. The assignment info includes assignment kind (blocking or non-blocking), Left Hand Side (LHS) expression, Right Hand Side (RHS) expression and the Sensitive List (SL). The SL is part of event-expr (12).

To memorize the dataflow, we need three key maps: the Dataflow map, Bits-core map and SSA-ID map. The Dataflow map stores all the assignments info. The Bits-core map stores a vector of number for each id, one number for one bit. The relation of the newly introduced variable and the original variable is stored in SSA-ID map in a way that new variable represents certain bits of the original variable. The formal definitions are listed below.

- **Bit-score**: ID -> UID*
- **Dataflow**: UID -> LHS × KIND × RHS × SL
- **SSA-ID**: ID -> ID×BIT_RANGE

Once an assignment is encountered, a unique number getting from the global counter with the assignment info will be added into Dataflow. Also, the corresponding bits of the LHS variable will be updated to the unique number. Therefore, the bit-vector number for each ID in Bit-score is a key for retrieving the assigned assignments in Dataflow.

Note that the LHS (lvalue) could be single variable, part-select, or a concatenation of them. In case of concatenation, a temporary variable will be introduced to get the value, and each piece of the LHS will get the value by part selecting this temporary variable.

The SSA conversion starts from the top modules, and iteratively calls conversion function of each syntax structure.

1) **Statement**
   a) Blocking assignment (7)
      1) Rename RHS based on current data flow. That is replace every ID in RHS expression with the newest version, the newest version can be checked from the Bit-score map.
      2) Update data flow with LHS (Get a UID from the global counter, introduce a new version variable for this assignment, and then add this variable into SSA-ID map. Update the corresponding bits of LHS in Bit-score to UID, and add the assignment into Dataflow).
   b) Non-blocking assignment (8)
      Similar to blocking-assignment, except the assignment type is NON-BLOCKING.
   c) Sequential block (9)
      Apply SSA conversion to each statement in the list.
   d) Loop statement (10)
      Be unfolded in elaboration phase.
   e) Conditional statement (13)
      1) Rename conditional expression first.
      2) Make a copy of data flow for each branch, and transform each branch using its copy.
      3) Merge data flow of then and else branch. That is, for each variable assigned in any of the branch, insert a \( \phi \) function to merge these two assignments.
j) Case statement (14)
  Be converted to conditional-statement in elaboration phase.

k) Function call statement (11)
  1) Get function definition (6) body being called.
  2) Do SSA transformation on this function definition, and get the return value assigned to this function id.
  3) Replace each formal id in return value with actual id.
  4) Update data flow with LHS.

l) Procedural assignment (12)
  1) Do SSA transformation on its body stmt-item.
  2) In case the statement is edge-sensitive, that is the assignments are non-blocking assignments, as we assumed before, then set all the newly introduced variables to net-type, and choose one or introduce a new version for each register variable assigned. Keeping only one version for each register is required by the Verilog semantic, or the cycle time will be changed.

2) Module item

a) Continuous-assign (2)
  Be treated as blocking assignment.

b) Variable declaration (1)
  If this declaration has assigned an initial value, then treat this assignment as continuous assign.

c) Gate instantiation (5)
  Take it as continuous assignments, as illustrate in table III.

d) Always block (3)
  Do SSA transformation on its body stmt-item.

e) Module instantiation (4)
  1) Put input formal id = actual param in Dataflow.
  2) Do transformation on the module being instanced.
  3) Put actual output param = output formal id in Dataflow.
  4) For every formal id, replace it with a new id.

f) Function definition (6)
  1) Do SSA transformation on the stmt-item list.

The SSA transformation call of function definition will be called when a function call statement is encountered.

g) Module declaration
  1) For each module item, apply SSA conversion.
  2) Keep a latest version of output variable the same as in original Verilog.
  3) As Verilog is not sequential executed, the variable used above may be assigned below. So another round should be taken to update each variable to the latest. In this step, no more variables would be introduced, just replace the variables of RHS expression in Dataflow with newest version.

3.3. Out of SSA

The SSA form can be easily transformed back to Verilog, for performing optimization and debugging. As we have all the statement information needed in Dataflow, the conversion out of SSA is straightforward. We only need to convert blocking assignment to continuous assign, and place non-blocking assignment in clocked always. As the top module is flattened during SSA conversion, there will be only one module for each top module.

The table IV below shows a Verilog program, and the Verilog converted out of SSA.
### Table IV. Out of SSA Form

<table>
<thead>
<tr>
<th>Original Verilog</th>
<th>Out of SSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>module simple_if(a, b);</td>
<td>module simple_if(a, b);</td>
</tr>
<tr>
<td>input [2:0] a;</td>
<td>input [2:0] a;</td>
</tr>
<tr>
<td>output [2:0] b;</td>
<td>output [2:0] b;</td>
</tr>
<tr>
<td>reg [2:0] b;</td>
<td>wire [2:0] b_1;</td>
</tr>
<tr>
<td>always @ (a) begin</td>
<td>wire [2:0] b;</td>
</tr>
<tr>
<td>b = 0;</td>
<td>assign b_1 = 3'b000;</td>
</tr>
<tr>
<td>if (a) b[1:0] = a[1:0];</td>
<td>assign b[0] = (a)?a[0] : b_1[0];</td>
</tr>
<tr>
<td>end</td>
<td>assign b[2] = (a)?b[0] : b_1[2];</td>
</tr>
</tbody>
</table>

The internal maps during the SSA transformation process are listed in table V. In the fourth row, when the two branch s3, s4 of the conditional statement s2 are finished, the Dataflow are merged. Besides, we use the copy propagation to eliminate the internal variable b_2 and b_3.

### Table V. SSA Form internal maps

<table>
<thead>
<tr>
<th>ID</th>
<th>Dataflow</th>
<th>Bit-score</th>
<th>Ssa-id</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>1, (b, 1 [2:0], BLK, 3'b000, NULL)</td>
<td>b_1[1 1]</td>
<td>b_1(b, 2,0)</td>
</tr>
<tr>
<td>s3</td>
<td>1, (b, 1 [2:0], BLK, 3'b000, NULL)</td>
<td>b_1[1 2 2]</td>
<td>b_1(b, 2,0)</td>
</tr>
<tr>
<td></td>
<td>2, (b, 2 [1:0], BLK, a [1:0], NULL)</td>
<td>b_2[2 0]</td>
<td>b_2(b, 1,0)</td>
</tr>
<tr>
<td>s4</td>
<td>1, (b, 1 [2:0], BLK, 3'b000, NULL)</td>
<td>b_3[3 3 1]</td>
<td>b_3(b, 2,0)</td>
</tr>
<tr>
<td></td>
<td>3, (b, 3 [2:1], BLK, a [2:1], NULL)</td>
<td>b_4[4 5 6]</td>
<td>b_4(b, 2,0)</td>
</tr>
<tr>
<td></td>
<td>4, (b, 4 [2:1], BLK, a[0:0], NULL)</td>
<td>b_5[4 5 6]</td>
<td>b_5(b, 1,0)</td>
</tr>
<tr>
<td></td>
<td>5, (b, 5 [2:1], BLK, a[0:0], NULL)</td>
<td>b_6[4 5 6]</td>
<td>b_6(b, 2,2)</td>
</tr>
</tbody>
</table>

3.4. Formal Model

The assignments in the Dataflow represented in SSA form define the formal model. For combinational circuit, the internal variables in the programs are net-type variables. The constraints for CEC can be modeled as letting the LHS equals to the RHS satisfied for each entry in Dataflow.

While for sequential circuit, we have to model the behavior of register variables. The register variable only appears in non-blocking assignment. And the SL is used for modeling the condition that triggers this non-blocking assignment.

4. Function Preserving

As the front-end of the combinational equivalence checking, the formal model extracted by our method must be function-preserving to ensure the verification result valid. In our approach, we make sure every transition in the prototype system would be function preserving by checking the equivalency of the model after transition and source model. Formality is an equivalence-checking solution that uses formal, static techniques to determine whether two versions of a design are functionally equivalent. We use this famous verification tool in the industrial world to check the equivalency.

From Verilog source model to our formal model, three transitions are needed: Verilog source model to AST model, AST model to elaboration model and elaboration model to SSA form model. SSA form model is the final extraction model to hand over to back-end solver. After every transition, we take the source code model as a reference model of Formality and the model through the transition as an implementation model to check whether the conversion is function-preserving. We also use ISCAS and PicoJava-II to test the robustness for each transition and make sure the transition is appropriate for most Verilog source models.
5. Experiment Result

5.1. Experiment Result

Our prototype system for extracting the formal model is implemented in C++ and we run the experiments on a 32-bit 1.9-GHz AMD Athlon(tm) 64 X2 Dual Core Processor with 2 GB of RAM under Linux. Currently, our system only support scalar and bit-vectors variable, and does not support memory type.

In order to prove that our converting procedure does not lose any information, we translate the Data-flow out of SSA to Verilog, and checking its equivalence with the primary design by Formality. We have tested it on a large number of synthesizable circuits. The circuits listed in table VI are some representative test cases, including both combinational and sequential circuits.

Among the test cases, the largest one is picoJava-II [15]. The picoJava-II has 15 blocks in total, while the topmost block CPU has memory variables which we do not support (Not Support, NS). Besides, the ICU block, containing combinational always that synthesis to latch, is not supported as we discussed in section III. We have run SSA conversion on the left 13 test cases, and 11 of them have proved equivalent. The other two blocks, IU and IFU are too large for the equivalence checking tool to load in (Over Memory, OM), so we are not sure about the equivalency.

Table VI. Verified test cases

<table>
<thead>
<tr>
<th>Test case</th>
<th>Circuits Description</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>PicoJava-II</td>
<td>11 of 15 blocks</td>
<td>PASS</td>
</tr>
<tr>
<td>ISCAS85[16]</td>
<td>10 combinational</td>
<td>PASS</td>
</tr>
<tr>
<td>ISCAS89[16]</td>
<td>31 sequential</td>
<td>PASS</td>
</tr>
<tr>
<td>ARITH[17]</td>
<td>A collection of</td>
<td>PASS</td>
</tr>
</tbody>
</table>

5.2. Efficient CEC Based on Our Front-end

As mentioned above, the extracted model can be used as the front-end of CEC. In [18], the CEC system based on our front-end makes a better performance in time and space complexity than Formality. In [18], the author performs an experiment with ISCAS ’85 benchmarks. The benchmarks contain several groups of test cases, which are all functional equivalents but implemented in different Verilog codes. ISCAS’85 circuits are based on gate level design. The experiment result shows the memory usage of CEC system is almost only half of Formality’s. Also, there is another experiment conducted in [18] with ARITH. ARITH benchmarks are used as arithmetic circuits to verify CEC system. In this test, it also demonstrates good performance in both time and space complexity compared with Formality.

The test result not only reveals our front-end is of function equivalence and robust, but also proves that our extracted model can be exploited efficiently.

6. Related Work

VI2mv is a tool that compiles a given Verilog description into FSMs. VI2mv extracts a set of finite state machines which preserves the behavior of the source Verilog program defined in terms of simulated results [4]. The vl2mv project is now part of a larger project VIS [3] that attempts to implement formal verification, synthesis, and simulation of finite state systems. Compared with our method, vl2mv supports a wider range of Verilog subset, and supports time constraints. Our method is more targeted, and it is sufficient for CEC. Besides, various optimizations on the SSA form are available, which can narrow the solution space of further verification.
SSA form has been recommended to represent data flow and it is an efficient data structure applied in program analysis and compiler optimization. Though SSA form is an attractive data structure, the procedure of its construction is not so easy and the problem of space explode discourages its use [19]. Minimal SSA form is a refinement of pseudo assignment put forward by Shapiro and Saint [20], but if there does not exist obvious $\phi$ function, it is difficult to manage the relationship between new names and old names. In [5], R. Cytron et al. came up with an idea to construct SSA from. The algorithms use dominance frontiers to place $\phi$ function, and it is a two-step process. First, some $\phi$ functions are inserted to the join nodes of data flow. Second, new variables $V(V = \phi(V_1, V_2))$ are generated. The transition can be performed efficiently and also leads a moderate increase in space. In our system, we adopt the concept $\phi$ function mentioned by R. Cytron et al. to convert to SSA form. According to the concrete Verilog source model, we analysis and research Verilog language feature to find how the source model can be converted to SSA form efficiently with function preservation and what structure we should use to store our SSA form data flow. Before converting it to SSA form, our system goes through AST and elaboration period. After the two processes, we can insert $\phi$ function to join nodes and rename new variables conveniently.

7. Conclusion

In this paper, we present a straight-forward method to mechanically extract formal models directly from the source code of Verilog design. Many formal verification tools cannot take input directly from Verilog source code, so the high-level designs have to convert to BLIF or BLIF-MV format by third party tools. In addition, the SSA form can be easily translated back to Verilog in this paper, and the correctness of the SSA conversion can be checked by the equivalency with our origin Verilog design. Experimental result shows that the SSA convention is function-preserving and robust. The method proposed here can be served in front-end design in formal verification and extracted model can be exploited efficiently in CEC.

Though we have converted the Verilog model to SSA form successfully, the SSA representation is not guaranteed to be optimal. In the future, we plan to improve and extend the SSA conversion and try to represent the formal model with an optimal SSA form.

Acknowledgment

This work was supported in part by NSFC under Grant Nos. 60973089, 60873148, 60773097, 61003101; Jilin Province Science and Technology Development Plan under Grant Nos. 20101501, 20100185, 20090108, 20080107, 201101039; Doctoral Fund of Ministry of Education of China (20100061110031); Opening Fund of Top Key Discipline of Computer Software and Theory in Zhejiang Provincial Colleges at Zhejiang Normal University; Zhejiang Provincial Natural Science Foundation under Grant Nos. Y1100191; Erasmus Mundus External Cooperation Window's Project (EMECW): Bridging the Gap, 155776-EM-1-2009-1-IT-ERAMUNDUS-ECW-L12. The Opening Foundation of Key Laboratory of Symbolic Computation and Knowledge Engineering of Ministry of Education under Grant No. 93K-17-2009-K05.

8. References


Arithmetic module generator based on ARITH. Available: www.aoki.ecei.tohoku.ac.jp/arith/mg/index.html


