IEICE TRANS. INF. & SYST., VOL.E97–D, NO.4 APRIL 2014

PAPER

Automatic Rectification of Processor Design Bugs Using a Scalable and General Correction Model

Amir Masoud GHAREHBAGHI[†*a)], ***Member and*** **Masahiro FUJITA**[†], ***Nonmember***

SUMMARY This paper presents a method for automatic rectification of design bugs in processors. Given a golden sequential instruction-set architecture model of a processor and its erroneous detailed cycle-accurate model at the micro-architecture level, we perform symbolic simulation and property checking combined with concrete simulation iteratively to detect the buggy location and its corresponding fix. We have used the truth-table model of the function that is required for correction, which is a very general model. Moreover, we do not represent the truth-table explicitly in the design. We use, instead, only the required minterms, which are obtained from the output of our backend formal engine. This way, we avoid adding any new variable for representing the truth-table. Therefore, our correction model is scalable to the number of inputs of the truth-table that could grow exponentially. We have shown the effectiveness of our method on a complex out-of-order superscalar processor supporting atomic execution of instructions. Our method reduces the model size for correction by 6.0x and total correction time by 12.6x, on average, compared to our previous work.

key words: *design error diagnosis, design error correction, micro architecture debugging, formal verification, processors*

1. Introduction

Design error diagnosis and correction (DEDC) is the process of debugging and rectifying errors in a design. Given an erroneous design and its reference design or golden model, DEDC does the following three operations: error diagnosis, error correction, and correction certification.

Error diagnosis is the process of analyzing the erroneous design and suggesting a number of candidate locations in the design whose appropriate modifications may correct the error. The error correction process selects the "good" candidates from the list of suggested candidates and tries to rectify the error by changing those portions of the design. After the design is changed, it should be verified/validated to certify that the bug is actually fixed and no new bug is introduced, as error corrections are mostly based on counter examples given and do not refer to the entire specification.

In this paper, we address the problem of automatic rectification of design bugs in processors at the micro-architecture level employing formal methods. At this level of abstraction, the elements of the datapath and their high-level functionality as well as the flow of data and sequence of micro operations for each instruction are modeled. Therefore, the bugs that are found in this level may reveal serious control problems. In this paper, we are targeting bugs in the controlling parts of processors. Controlling parts include the controller itself as well as the select line of the multiplexers in datapath. Therefore, we can find bugs related to incorrect routing of data as well as bugs in the controller. Note that the datapath elements, such as functional units, are assumed to be verified separately, so they are abstracted in our method.

As shown in Fig. 1, inputs of our method are two formal models of a processor. The first model (golden model) represents the sequential behavior of the processor's instruction-set architecture (ISA). The second model (erroneous implementation model) represents a detailed cycle-accurate behavior of the processor at the micro-architecture level.

In the diagnosis phase, we insert multiplexers in different locations of the erroneous model, similar to the previous methods for RTL/gate-level designs [1], [2], to obtain a new enriched model. We do equivalency checking between the erroneous model and the golden model to find the sequence of instructions that are executed incorrectly in form of counter examples. Then, we use the counter examples in the multiplexer-enriched model to determine the candidate location for correction. In this work, we try to correct the design by changing only one candidate location at a time. However, it can be extended to consider more than one location for correction at a time, as discussed later in Sect. 4.5.

In the correction phase, we find a function, which we name it the correcting function, for the candidate location that is obtained from the diagnosis phase, such that its exclusive-or with the original erroneous function makes it correct, as shown in Fig. 2. We model the correcting function in a truth-table. We find the rows of the truth-table that should be 1 (or minterms) to correct the design in an iterative process employing the counter examples from the diagnosis phase. In other words, we determine under which input values the erroneous function must be complemented, using the exclusive-or operation. In this phase, first we employ a concrete simulator and the generated counter example to quickly find and filter out the correcting functions that cannot actually correct the design. After that, we go through the process of formally certifying the correction, which is more time consuming due to using formal methods.

In the correction phase, we do not explicitly represent the truth-table in the erroneous model. Therefore, we do not add any new variable to the model. Instead, we extract

Manuscript received June 25, 2013.
Manuscript revised November 20, 2013.

[†]The authors are with the VLSI Design and Education Center (VDEC), The University of Tokyo, Tokyo, 113–0032 Japan.
[*]Presently, with the Deprtment of Electrical Engineering and Information Systems, The University of Tokyo, Tokyo, Japan.
a) E-mail: amir@cad.t.u-tokyo.ac.jp

DOI: 10.1587/transinf.E97.D.852

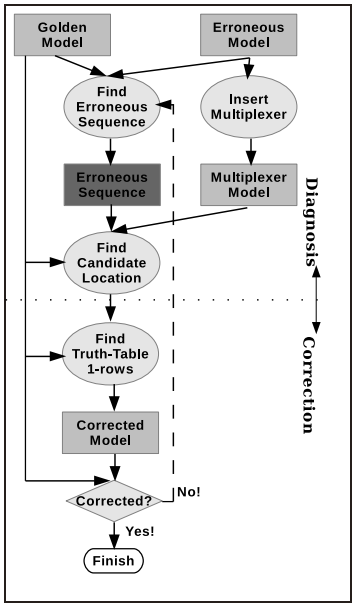

Fig. 1 Overview of our method.

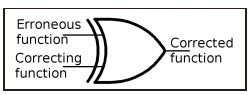

Fig. 2 General view of correcting function.

the rows of the truth-table (or minterms), which are used for correction, from our backend formal engine in terms of counter examples. This way, we implicitly model truth-tables of any size without increasing the complexity of the model. Consequently, our correction model becomes scalable to any correcting function with any arbitrary number of input variables. Note that explicitly representing a truth-table with N inputs requires 2^N new variables to show the results of each row, which is not practical for large Ns.

After the correction phase, we formally check equivalency between the corrected model and the golden model to make sure that the error is corrected and no new error is introduced. If correction is not successful, we continue the diagnosis and rectification processes with a new counter example. Note that if the correction is not possible with the current candidate location, we select another candidate location in the diagnosis phase and continue the process.

We have presented the idea of automatic correction of design errors of complex processors employing formal methods with bug models in [3] and without bug models in [4]. In this work, we have improved our previous works through the following main contributions:

1. Introducing a new correction method based on implicit representation of truth-tables for correction, which makes the correction model scalable for large functions as well as significantly reducing the model size for our backend formal engines; hence, improving runtime and memory usage of the method.
2. Introducing a method based on concrete simulation to accelerate the correction phase by finding and filtering out the correcting functions that are not actually correcting; consequently, accelerating the overall correction process.
3. Introducing a new approach for diagnosis and correction by splitting our previous diagnosis formula, which substantially reduces the model size for our backend formal engine; hence, improving runtime and memory usage of the method.
4. Introducing new sophisticated constraints in the diagnosis and rectification phases, which significantly improves the correction resolution of the method.

Finally, we have shown new experiments on a 4-way out-of-order superscalar processor, which we could not handle in our previous methods due to the size of the model for our backend formal engine.

The rest of the paper is organized as follows. Section 2 overviews the related work. Section 3 briefly presents background on processor verification using formal methods. In Sect. 4 we present our diagnosis method followed by Sect. 5 that presents our correction method. Section 6 shows the experimental results. Section 7 concludes the paper.

2. Related Work

Most of the previous works are at transistor-level [5] or gate-level [6], [7] and they only address automatic error diagnosis for combinational circuits or sequential circuits assuming existence of full scan chain. Although there are some works such as [8], [9] which also propose automated correction, the correction process remains a manual task in most of those works.

There are other works targeting RTL designs that try to find and correct errors in RTL descriptions. One category of works use test vectors and simulation techniques to locate the erroneous parts of the design [10], [11]. Some of the works also try to correct the error [12] using bug models. Other category of works employs formal methods. They need a golden model (description) to find the errors. Compared to the simulation-based methods, they do not rely on sophisticated test vectors, although formal methods cannot deal with large designs due to the state space explosion problem. Those methods insert some form of multiplexers into the erroneous design [1], [2] and check a property for error diagnosis.

[13] introduces a method for efficient error diagnosis in processors. The method is based on correspondence checking on a specification and an (erroneous) implementation model of a processor to find the differences in form of a counter example. Then, the counter example is used to simplify the CNF formulas that are generated from the multiplexer-enriched implementation model for SAT solver. Finally, the minimum numbers of multiplexers that can fix the bug are determined in an iterative manner employing a heuristic for variable ordering of CNF formulas that dramat-

ically reduces the debug time. However, it does not address the correction of the errors and it remain a manual task.

Our work is different from the other previous works in several aspects. First, we are targeting processors at micro architecture level that is a higher and more abstracted level than RTL/gate-level/transistor-level. Second, although our diagnosis method is multiplexer based and uses a formal approach, like some of the RTL methods, we have proposed a new approach to ease both the diagnosis and the correction processes. Third, the input sequences that lead to error in the model are generated using formal methods in the form of counter examples. Those sequences are usually obtained by simulation/emulation or actual hardware run in the existing RTL/gate-level methods. Fourth, for the correction phase, we do not use bug models and we have proposed using truth-tables that are general format for correcting functions. In addition, we do not resynthesize the erroneous function for correction. Instead, we find conditions that the function is incorrect and its output should be corrected (realized as complementing the Boolean functions). Finally, for correction certification, we do not rely on test vectors and instead use formal methods.

3. Background

3.1 The UCLID System

The UCLID system utilizes the logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions (CLUF) to specify and verify systems which may have infinite or unbounded states. The UCLID system consists of an input language, a symbolic simulator, and a decision procedure. The UCLID input format supports CLUF related constructs including uninterpreted functions and predicate symbols, an arithmetic of counters, bit-vector arithmetic and restricted lambda expressions. The UCLID verification engine can be configured for different kinds of verification tasks, including bounded model checking, correspondence checking, inductive invariant checking, property checking of quantifier-free first-order logic formulas involving counter arithmetic, and limited property checking of formulas with universal quantifiers. However, we use it as a correspondence checker and property checker. For more details of UCLID please refer to [14] and [15].

3.2 Correspondence Checking

Correspondence checking is a method that is first introduced for processor verification in [16]. Correspondence checking proves a correspondence between the implementation (detailed model) and the ISA (Instruction Set Architecture) specification of a processor employing symbolic simulation and property checking as follows.

As shown in Fig. 3, the implementation (Q_{impl}) is symbolically simulated with an arbitrary input combination for one step ($Q_{impl} \mapsto Q^1_{impl}$). Then, the pipeline is flushed for a number of steps (say, N2 steps) until all partially executed

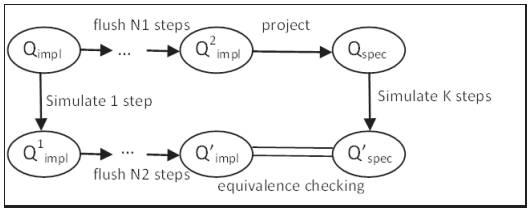

Fig. 3 Correspondence checking diagram.

instructions are completed and the programmer-visible parts of the implementation are saved ($Q^1_{impl} \mapsto Q'_{impl}$). After that, we re-initialize to the start state, and in order to have a state in the specification, which corresponds to the implementation, first, the implementation is flushed for a number of steps (say, N1 steps) ($Q_{impl} \mapsto Q^2_{impl}$), and then the state of the implementation is projected into that of the specification ($Q^2_{impl} \mapsto Q_{spec}$). Note that N1 and N2 are the maximum cycles that are required to finish execution of all the instructions that are on-the-fly.

After symbolically simulating the specification for K steps (K corresponds to the issue width of the processor) ($Q_{spec} \mapsto Q'_{spec}$), the programmer-visible components such as Program Counter (PC), Register File (RF) and Memory (Mem) in the Q'_{impl} should be equivalent to those of Q'_{spec}, which is the following correspondence property:

$$(Q'_{impl}.PC = Q'_{spec}.PC)\&(Q'_{impl}.RF = Q'_{spec}.RF)\& \\ (Q'_{impl}.Mem = Q'_{spec}.Mem) \quad (1)$$

4. Error Diagnosis

Our error diagnosis method consists of the following steps. Details will be presented in the following subsections.

1. Generate a list of variables of the erroneous model that are suspicious to be the cause of the error.
2. Insert multiplexers in all the candidate variables in the erroneous model to create an enriched model.
3. Perform the modified correspondence checking using the golden model and the erroneous model to find an erroneous sequence of instructions.
4. Perform the modified correspondence checking using the golden model and the enriched model with the erroneous sequence of instructions and the modified property to find the candidate variables for error correction.

4.1 Suspicious List Generation

Suspicious list is the initial list of variables that may be the cause of the error. The initial list contains all the controlling parts of the model. Controlling parts contain two kinds of variables. The first kind is the variables that are explicitly defined by the designer in the model. The second kind is the auxiliary variables that are added by our method and correspond to the controlling functions that route a value in the datapath. Those variables can be considered as select line of

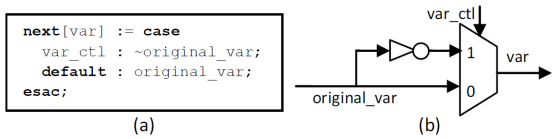

Fig. 4 Multiplexer insertion (a) code (b) equivalent hardware.

the multiplexers in datapath part of the processor. Note that both kinds of variables are Boolean.

Generally, the suspicious list can contain all the design variables; hence, it may be very large. One way to reduce the size of the list is by user guidance. The user, who has a good insight into the design, can specify portions of the design that are more suspicious to be erroneous.

Another way to systematically reduce the size of the list is to consider the changes from the previously verified implementation model. Assuming an incremental design method, the new changes to the previous model are error prone and should be verified first. For example, if we have added a timing error recovery mechanism to a verified processor model, the new errors are most probably due to the new changes; hence, should be considered first.

4.2 Multiplexer Insertion

Multiplexers are the heart of the automatic error diagnosis method. They are used in a similar way to the RTL and gate-level diagnosis methods, but they are used here in a higher level of abstraction and a little differently as we will explain later in this subsection.

For each variable in the suspicious candidate list, we add a multiplexer as shown in Fig. 4. The *var_ctl* is a new variable that is a pseudo input that can get an arbitrary value during the symbolic simulation. The input 0 of the multiplexer is the *original_var* and the other input of the multiplexer is complement of the *original_var*. That is because we are only interested in situations where *var_ctl* becomes 1 and a different value from the original one should be propagated in the design. Please note that all the control variables are Boolean (binary). Therefore, if the value of a variable is wrong, its complement should be correct.

In the previous RTL or gate-level diagnosis methods, when the *var_ctl* becomes 1, a new fresh variable is used instead of *~original_var*, in our method. This way, they create a new function for correcting the incorrect cases. We use, however, the complement of the existing function for correction, the same way as [4]. This way, we also avoid introducing a new variable for the input 1 of the multiplexer for the formal engine, which makes the model simpler.

4.3 Erroneous Sequence Generation

Erroneous sequence of instructions are necessary to find the buggy locations as well as their corresponding fixes. The erroneous sequence can be generated by correspondence checking (see Sect. 3.2) between the erroneous implementation model and the golden model. Since the model is erroneous, the correspondence checking process generates a counter example. The generated counter example can be used to extract the sequence of instructions that is erroneous.

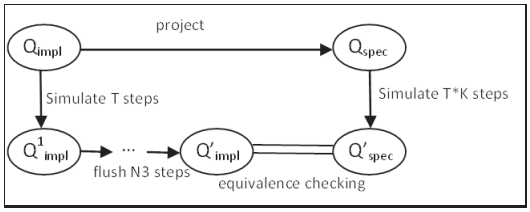

Fig. 5 Modified correspondence checking diagram.

The correspondence checking is a very costly operation, specially for the processors with deep pipelines or superscalar processors, since we need to flush the pipeline by performing symbolic simulation. Since our goal is to find the bugs, we begin from the initial state and do symbolic simulation until the pipeline becomes full (lets say for T cycles). This way, we generate random arbitrary sequence of instructions. Those sequences can reveal the bugs much faster than the original correspondence checking which needs lots of symbolic simulation for flushing the pipeline before projecting into the specification model. Note that we still need to flush the pipeline to make sure that execution of all the on-the-fly instructions are finished.

The initial state that we start simulating is not the reset state, but it is a state that there is no instruction on the fly. In that sense, all the programmer-visible parts, such as program counter, register file and memory, have arbitrary values. Moreover, the flags related to executing previously executed instructions have arbitrary values. Therefore, the effect of execution of the previous instructions is considered. Moreover, since we fill the pipeline with arbitrary instructions, all the possible dependencies among the instructions is considered. Theoretically, there may be cases that is not covered this way. However, intuitively, almost all the possible cases are considered for bug detection. Moreover, using the modified correspondence checking in our experiments, we did not miss detection of any bug.

The modifications to the original correspondence checking diagram (Fig. 3) is shown in Fig. 5. In Fig. 5, K is is the fetch width of the processor. Therefore, if we fetch instructions for T cycles, we need T*K cycles symbolic simulation in ISA model (Spec) to finish execution of all the instructions that are fetched. Note that Q_{impl} is a state in which no instruction is on the fly. Therefore, we can easily project PC, RF, and Mem, which have arbitrary values, to the ISA model. Also note that we use exactly the same property as the original correspondence checking method as shown in Sect. 3.2.

In the modified correspondence checking diagram, both Spec and Impl models are symbolically simulated in parallel and also $T \ll N3$. That is because, when we fetch for T cycles to fill the pipeline, $K * T$ instructions execute in parallel. In the worst case, because of the dependencies among the instructions, they may be executed one-by-one.

Therefore, $N3$, which is the maximum number of cycles to finish the execution of all the instruction on the fly, corresponds to $K * T$. Assuming super-scalar processors, which have $K > 1$, we can conclude that $T \ll N3$.

Therefore, the dominant part becomes $N3$. On the other hand, in the original correspondence checking digram (Fig. 3), first, we have to symbolically simulate for $N1$ cycles, then do projection, and finally symbolically simulate the two models (Spec and Impl) in parallel for $N2$ cycles. Therefore, the total symbolic simulation cycles roughly becomes $N1 + N2$. As $N1$ and $N2$ and $N3$ are of similar size, we can conclude that the modified correspondence checking method needs roughly half symbolic simulation cycles compared to the original correspondence checking method.

4.4 Candidate Variable Selection

After obtaining the erroneous sequence, we have to find a candidate location for correction. We perform the modified correspondence checking, similar to what we did for erroneous sequence generation, using the multiplexer-enriched model and the golden model, with the following differences.

First, as we want to find a solution to correct the error, we use the negation of the property (1), which we have used for erroneous sequence generation. This way, if the property fails, the counter example contains a solution. However, if the property satisfies, it means that no solution exists.

Second, we perform the modified correspondence checking, as introduced in the previous subsection. Moreover, we add the sequence of instructions along with proper constrains to the property that is checked. The constraints are necessary to define the exact condition that the error occurred as well as to ensure a valid correction. We will discuss more about the constraints in the next subsection. The general format of the property for candidate variable selection is as follows. Note that *Enr* is multiplexer-enriched model and *S pec* is the *IS A* specification model.

$$constraints \Rightarrow\sim ((Q'_{enr}.PC = Q'_{spec}.PC)\&(Q'_{enr}.RF = Q'_{spec}.RF)\&(Q'_{enr}.Mem = Q'_{spec}.Mem)) \quad (2)$$

If the property that we are checking fails, a counter example is generated that shows a possible solution for correction. The candidate location for correction is the variable whose select line of the multiplexer is activated in the counter example. By selecting that candidate location, we proceed to the correction process, which is explained in Sect. 5.

If the property holds, it shows that correction is not possible with the given constraints. As in this work we have considered correction by changing one location, it means that for the given erroneous sequence, we cannot make the result of execution on both *S pec* and *Enr* models the same by only activating select line of one multiplexer. Therefore, we may need to consider more candidate locations for correction at the same time. In other words, we have to change the constraint so that more than one select line of the multiplexers are allowed to be activated at the same time.

The combination of the error sequence generation (previous subsection) and candidate variable selection (this subsection) is similar to our previous method [4]. Intuitively, in [4] we were looking for a sequence of instructions that was executed incorrectly on *Impl* model, while it was executed correctly on the *Enr* model. Moreover, Our previous method involved in handling three models (*S pec*, *Impl*, and *Enr*) at the same time. However, in this work we have divided our previous diagnosis process into two processes, each involving two models, instead of three. As a result, the overall process is done more efficiently as confirmed by our experimental results.

If we rewrite formulas (1) and (2) as P_{impl} and $constraints \Rightarrow\sim P_{enr}$ respectively, the combination of them becomes as follows.

$$P \cong\sim P_{impl} \Rightarrow (constraints \Rightarrow\sim P_{enr}) \quad (3)$$

The above formula means that if P_{impl} fails and a counter example (or erroneous sequence) is generated, then we try to find candidate variable. Formula (3) can be rewritten according to the Boolean algebra as follows:

$$P \cong\sim (\sim P_{impl}\&(constraints\&P_{enr})) \quad (4)$$

$$P \cong\sim (constraints\&(\sim P_{impl}\&P_{enr})) \quad (5)$$

The above formula is the same as the formula in [4] with additional constraints. Actually, part of the constraints is related to the erroneous sequence that is generated by P_{impl} formula, as explained in the next subsection.

4.5 Adding Constraints

The constraints, which are specified for the formal engine, are not only important for success of the method for correction, but also crucial for the effective and efficient correction of bugs. We have defined the following types of constraints in our method to guide our backend formal engine.

1. Related to the erroneous sequence
2. Related to the desired solution
3. Related to the integrity of the solution

Constraints related to the erroneous sequence ensure that the exact condition for the erroneous behavior is provided for correction. These constraints include: specifying the sequence of instructions, the data dependencies among the instructions, and execution status of branch instructions; i.e. the branches are taken or not.

If we do not provide the exact conditions, the input sequence becomes a superset of the erroneous condition. Therefore, it is possible that we miss the correction for that specific erroneous case; hence, not correcting the design. For example, the error may be in executing a load instruction after a register-register instruction only when the load address depends on the output of the register-register instruction. If we do not specify that specific data dependency between those two instructions, we are dealing with a sequence of two instructions that is sometimes erroneous and

sometimes correct. In that sense, we are dealing with a more general case than specifying the data dependency between those two instructions.

The constraints that are related to the erroneous sequence are automatically extracted from the counter example that is generated in Sect. 4.3.

Constraints related to the desired solution are related to which locations should be considered for correction and under what conditions. For example, in this work we are interested in correction at one location. Therefore, we define appropriate cardinality constraints that allows only select line of one of the multiplexers to become active. Similarly, if we want to correct M locations, we have to add cardinality constraints that allows select line of M multiplexers to become active. Moreover, if we are trying to fix a specific location, which is introduced in the diagnosis, we add constraints to prevent other multiplexers on other locations to become active. The constraints that are related to the desired solution depend on the correction method approach and they are generated automatically.

Constraints related to the integrity of the solution are very important. These constraints make sure that a correct and valid solution is provided as follows. We are not interested in any solution that involves out-of-order commit of instructions. Therefore, we add appropriate constraints to prevent it. Note that although instructions may be executed out-of-order in a processor, commit is almost always done in-order.

Another such constraint is to ensure execution of an instruction that is followed by a branch instruction that is not taken. Similarly, we have constraints to ensure not executing an instruction after a branch instruction that is taken.

The constraints that are related to the integrity of the solution are initially defined based on the previous experiments with similar designs or similar bugs that are corrected/not corrected. Those constraints may be gradually evolved as we do more experiments. Once such constraints are defined, they can be automatically generated and reused for all the other bugs or other similar designs. As we will show in our experimental results, these constraints can greatly affect the correction process.

5. Error Correction

Error correction deals with constructing a function, which we name it the correcting function, for select line of the multiplexers that are suggested after diagnosis. Assuming that we want to correct the *original_var* in Fig. 4, we have:

$$var = original\_var \hat{} var\_ctl \tag{6}$$

Assuming the functions of each of the above variables:

$$original\_var = F_{org}(V_i \cdots V_j) \tag{7}$$
$$var\_ctl = F_{correcting}(V_m \cdots V_n) \tag{8}$$
$$var = F_{new}(V_s \cdots V_t) \tag{9}$$
$$\{V_s \cdots V_t\} = \{V_m \cdots V_n\} \cup \{V_i \cdots V_j\} \tag{10}$$

In this work, we have assumed that the designer has not missed any variable when constructing the original (erroneous) function. In other words, the error is not due to missing an input variable. Therefore, the set of input variables for F_{org} and F_{new} are the same; i.e. $\{V_s \cdots V_t\} = \{V_i \cdots V_j\}$. Consequently, we have $\{V_m \cdots V_n\} \subseteq \{V_i \cdots V_j\}$. We have considered the general case of $\{V_m \cdots V_n\} = \{V_i \cdots V_j\}$ in this work. However, after the correction, we may find out that some of the variables are don't care; hence, can be safely removed.

The correction is performed in an iterative manner as follows. Details will be given in the following subsections.

1. Generate a counter example.
2. Extract the correcting function from the counter example.
3. Validate the correcting function and resolve the conflicts.
4. Certify that the design is corrected.
5. If the design is not corrected, continue the above process, otherwise finish the correction process.

5.1 Counter Example Generation

In this subsection, we briefly discuss about how the counter example is generated. Then, in the subsequent subsections we discuss about how to extract the correcting function from the generated counter example.

For extracting the correcting function, we use the counter example that is generated by the modified correspondence checking method, as explained in Sect. 4.4, with the following considerations.

1. After the diagnosis is finished and the candidate variable is known, we use the same counter example that is generated for candidate variable selection.
2. If the generated correcting function is not valid, as will be explained later, we add additional constraints to exclude the current invalid solution and run the modified correspondence checking again to obtain a new counter example.
3. If the generated correcting function cannot correct the design, we add additional constraints to exclude the current solution and run the modified correspondence checking again to obtain a new counter example. Note that the new counter example may suggest another candidate location for correction.
4. Finally, if no counter example is generated after performing the modified correspondence checking, we add additional constraint to exclude the current sequence of instructions and generate a new sequence of instruction for correction as explained in Sect. 4.3.

5.2 Extracting the Correcting Function

As mentioned before, we assume that the correcting function has the same input variables as the original erroneous

function. Therefore, its corresponding truth-table model contains 2^n rows; where n is the number of function inputs. Each row of the truth-table represents the results of the correcting function, which can be 0 or 1.

Therefore, there are generally 2^{2^n} possible correcting functions, which can be very large for large n. However, there are two main things to be considered. First, in most cases there are more than one correcting function that can actually correct the design. That is because, in general, there are a lot of don't cares in every logic function, including the correcting function. As a result, the actual number of possible correcting functions is very smaller than the general case. Second, as we are using formal methods, the formal engine, with our guidance through the constraints, explores all the possible functions for a feasible one. Therefore, both quality of the constraints as well as efficiency of the formal engine affect the overall process of correction.

In this work, we have not explicitly specified the truth-table in the design. However, the truth-table partially exists in the generated counter example as follows.

At each symbolic simulation cycle, the input variables of the truth-table have a specific value of either 0 or 1. Moreover, the result of the truth-table (or function) exists as the value of the select line of the multiplexer on the candidate variable. Therefore, for each symbolic simulation cycle we can extract one row of the truth-table. As a result, for a counter example of M cycles, we can extract maximum M rows of the truth-table; since input variables of the truth-table may have exactly the same combination of values in several cycles.

Extraction of information of one cycle is straightforward. If we consider the combination of values of all the input variables of the truth-table in each cycle, they correspond to exactly one row of the truth-table. The result of the function, which is represented in the truth-table, at that row is also available in the same cycle. Moreover, we can construct a minterm for each cycle. If we consider a variable itself when its value is 1, and the complement of the variable when its value is 0, we can easily construct the corresponding minterm at each cycle by conjuncting all the corresponding variables or their complement.

For example, consider a simple example for correction as shown in Fig. 6. Assume that the correcting function is *var_ctl*. Furthermore, assume that it is a function of variables $a1$, $a2$, and $a3$. The counter example is 4 cycles long. As it can be seen, only 3 rows of the table are determined in this example. The remaining rows are not known in this counter example and we assume that they are 0 for generating the correcting function.

5.3 Validity of the Correcting Function

As explained before, we extract the truth-table rows from the generated counter example. A common situation in this process is that we may extract a row multiple times. For example, in Fig. 6 cycle 1 and cycle 2 correspond to the same row. In this case, if the result of the function is different, we call it a conflicting case and it is not acceptable. For example, in Fig. 6 the result of the correcting function (*var_ctl*) is the same (0) and it is ok. But, if it was 0 in one cycle and 1 in the other cycle, it was a conflicting case.

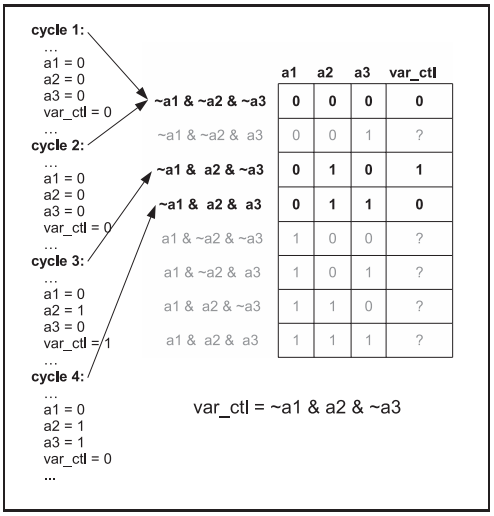

Fig. 6 Example of extracting a truth-table.

value1	value2	final value
0	0	0
1	1	1
0	?	0
?	0	0
1	?	1
?	1	1
?	?	?
0	1	X
1	0	X

Fig. 7 Aggregating the values.

According to the above definition, we call an extracted truth-table *valid*, if there is no conflicting case, or we can resolve the conflicts. Resolving a conflict means to determine whether the fix is possible for the specified erroneous sequence of instructions assuming the conflict did not exist. In this case, we assume the same logic value 1 for all the conflicting cases and validate the resolution by performing concrete simulation, as will be explained in the following subsection. In other words we exclude the conflicting cycles for truth-table generation.

Note that because of limited number of symbolic simulation cycles, each extracted truth-table may have a number of rows with unknown values (shown with ? mark in Fig. 6). As, we may not be able to find the correcting function with only one counter example, we may extract several valid truth-tables from several counter examples and aggregate the results in the final truth-table of the correcting function. In this case, there should be also no conflicting case, or the conflicting cases can be resolved as explained above. The rules for aggregating the values for the same row of a truth-table are shown in Fig. 7.

5.4 Resolving Conflicts

As explained before, during extraction of the rows of the truth-table for the correcting function, we may find that one row of the truth-table can be both 0 and 1, according to the counter example, in different cycles. The reason for this situation is that the formal engine can freely assign the control of the multiplexers to either 0 or 1 in different cycles without any kind of restriction. In other words, in general, the select line of the multiplexer is a function of all the design variables in all the cycles. However, we are extracting a deterministic function of n variables, which can be either 0 or 1 for each combination of variables' values.

One way to resolve the conflict is to use formal methods. However, this is a costly operation mainly because of using a symbolic simulator. Therefore, we use a concrete simulator to determine whether for the specific erroneous sequence of instructions, correction is possible after resolving the conflict or not. The reason that we can use a concrete simulator is that we are using the counter example, and in a counter example all the values of inputs as well as the variables in different cycles are assigned concrete values; hence, there is no symbolic or unknown value.

After performing the concrete simulation on the design after adding the generated correcting function, if correction is possible, we proceed to certifying the correction, as presented in the following subsection. However, if after resolving the conflicts and adding the generated corrected function to the design, the erroneous sequence cannot be executed correctly, the generated correcting function cannot correct the bug in general. Consequently, we can skip the costly process of certification that is based on formal methods.

The overall process of resolving a conflict is as follows.

1. Set the conflicting value to 1 (ignore the conflicting cycles)
2. Create the correcting function and apply it to the model accordingly
3. Determine the first cycle that a conflict has happened
4. Extract all the values of variables and inputs at the first conflicting cycle from the counter example
5. Perform concrete simulation from the first conflicting cycle until execution of all the instructions are finished
6. Determine the cycle that the execution of all the instructions are finished in the counter example
7. Compare the state of the processor before and after conflict resolution by comparing the corresponding final states from the counter example and the concrete simulation results. If we have reached to the same state after conflict resolution, conflict is successfully resolved. Otherwise, correction is not possible in this case.

As shown in Fig. 8, assuming the first conflicting cycle is *Cycle i*, we begin concrete simulation from that cycle since before that, two models behave exactly the same. We continue concrete simulation until *Cycle k* that execution of all the instructions are finished. Then, we compare the state of the processor to the last cycle in the counter example that the execution of all the instructions are finished (*Cycle j*).

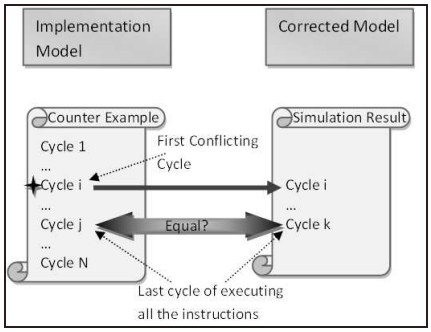

Fig. 8 Overview of conflict resolution.

5.5 Certification of Correction

We need to certify that the correction could actually rectify the error. That is because each generated counter example only guarantees the fix for that particular case. Even if we consider more counter examples, the correction cannot be guaranteed. Therefore, all the correction methods, including our method, need to validate the design after fix. To certify the correctness of the design after fix, we employ formal methods and perform correspondence checking between the possibly corrected design and the golden model. This way, we guarantee that the fix has been actually successful and the bug is corrected.

6. Experimental Results

6.1 Experimental Setup

We have chosen an out-of-order superscalar processor as our case study, as shown in Fig. 9. In this architecture, N instructions (N=3,4 in our case study) are fetched, decoded, and sent to the reservation stations at every cycle. The processor instructions are abstracted to 7 instructions: general register-register (RR), general register-immediate (RI), memory load (LD), memory store (ST), conditional branch (BR), load linked (LL), and store conditional (SC). The implementation model of the processor for N=3, is about 725 lines of UCLID code and contains 28 control variables, and for N=4, it is about 1050 lines of UCLID code and contains 36 control variables.

LL-SC instruction pair is a very powerful mechanism for synchronization among cores and atomic execution, which is implemented in many RISC processors such as MIPS, PowerPC, Alpha, and ARM. The usual way of using LL-SC instruction pair is for implementation of atomic Read-Modify-Write (RMW) operation. We have used the MIPS implementation of LL-SC instruction pair as described in [17]. Details of our implementation in UCLID can be found in [4].

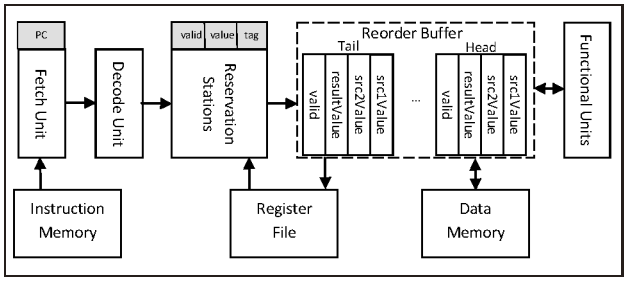

Fig. 9 General architecture of a superscalar processor.

We have used UCLID v.1 [15] and Minisat 2.2 [18] in our experiments as formal verification engines. Note that UCLID uses Minisat as its SAT solver engine. Also, note that UCLID v.1 does not support Minisat SAT solver, and we have added the support to UCLID.

We have developed a program in C++ around 9200 lines of code to automate the diagnosis and correction of errors. Our program reads the original UCLID models, inserts multiplexers and truth-tables, collects the output results of running UCLID, analyzes the counter examples, performs concrete simulation of UCLID models, resolves conflicts, and generates the corrected models in UCLID format.

All the experiments are performed on a PC with Intel Pentium Dual Core 2.5 GHz processor (only one core is used) with 2 GB main memory running Linux kernel 2.6.39 32-bit.

6.2 Results

In the first experiment, we show the effectiveness of the new proposed method to reduce the model size for verification as well as the verification time compared to our previous method [4]. We have chosen the same processor model (N=3) and the same set of bugs of [4] for this experiment. We have added multiplexers on all the controlling variables in the reorder buffer, assuming existence of designers guidance. This way, the total number of variables in the suspicious list is reduced around 15%. Results are shown in Table 1. Note that our previous method cannot handle the processor with N=4 due to size of the model, which is not handled by the backend formal engine.

As shown in Table 1, we could correct all the bugs as our previous method, while reducing the model size as well as the total correction time by 6x and 12.6x on average, respectively. Note that Model size (column 4) is the number of clauses in the CNF formula that is generated for the SAT solver engine. Also note that the average reduction in correction time is among the first four bugs that are corrected and bugs 5-6 are excluded because of timeout in the correction process.

For the bugs 5 and 6, both our previous method and the new one could not correct the error within the specified time limit. Note that in both not-corrected cases, our method could correctly diagnose the bug. This means that the bug location could be correctly determined, but an appropriate correcting function was not found. As we investigated, in both cases the method have suggested some rows of the truth-table for correcting function to be 1, while they should not. This means that those rows are don't cares for those counter examples. However, in general case, those rows should be 0 to be able to correct the design.

In the second experiment we have used the processor model with N=4. We have selected 8 different variables in the reorder buffer (ROB) of the processor and injected 10 bugs on them as shown in Table 2. We selected all the 20 variables of ROBs as initial suspicious list for multiplexer insertion. We have done the experiments in three different manners as will be explained later. The results of applying our correction method is shown in Table 2.

In Table 2, #Vars is the number of input variables of the correcting function, #Candidates is the number of candidate locations that are tried, #Sequences is the number of erroneous sequences that are generated for correction, and #UCLID runs is the number of times that UCLID is run. Total time reduction shows reducing the total time for correction comparing *constraint*2 vs. *constraint*1, and *simulation* vs. *constraint*2. Finally, the average reduction in total correction time is reported for *constraint*2 (vs. *constraint*1), and *simulation* (vs. *constraint*2). Note that for all the experiments, the total time for correction is due to UCLID runtime and our analysis program runs less than a second for all the cases.

The three manners that we have used in this experiment are as follows. *constraint*1 represents our primary set of constraints for correction as explained before. In *constriant*2 manner, we added some additional constraints related to the integrity of the solution (as explained in Sect. 4.5) that forces the formal engine to more strictly follow the execution behavior of instructions like the specification model. In the *simulation* manner, we have also used our concrete simulation engine for resolving the conflicts, as explained in Sect. 5.4.

The results for *constraint*1 manner show that 4 bugs out of 10 could not be corrected, although the bug was diagnosed correctly. As we examined, among the 20 suspicious variables for correction, the actual buggy location has been selected as first or second candidates in 8 cases out of ten. For two cases the buggy location was selected as third candidate. This shows that even in the cases that we could not correct the bug, the diagnosis has done very well.

We analyzed the four cases that we could not correct and the following results obtained. In two cases, the reason for not correcting was due to our valid truth-table detection mechanism. As we explained before, when we detect a conflicting case, we make the extracted truth-table invalid, exclude the conflicting case, and try to obtain a valid truth-table. The problem is that sometimes the excluded case is actually necessary to correct the error.

The other two cases that we could not correct were due the counter example that was generated by our backend formal engine (UCLID). The problem was that due to some unknown reason, most of the generated counter examples were

Table 1 Correction results for N=3.

Bug No	Brief Description	Correction Method	Model Size	Model Size Reduction	Total Time	Total Time Reduction	Fixed?
1	Adding an erroneous term	[4]	4,242,928		51m 19s		Yes
		new	2,688,667	1.6x	4m 50s	10.6x	Yes
2	Removing several correct terms	[4]	599,185		9m 55s		Yes
		new	338,260	1.8x	32s	18.6x	Yes
3	Removing a correct term	[4]	4,606,825		33m 29s		Yes
		new	530,767	8.7x	3m 25s	9.8x	Yes
4	Adding a negation operator	[4]	4,443,136		27m 41s		Yes
		new	530,845	8.4x	2m 27s	11.3x	Yes
5	Changing an operator type	[4]	4,004,443		> 2h		No
		new	540,497	7.4x	> 1h	N/A	No
6	Multiple above changes	[4]	4,606,825		> 2h		No
		new	569,385	8.1x	> 1h	N/A	No
			Average Reduction	6.0x	Average Reduction	12.6x	

Table 2 Results of correction for N=4.

Bug No	Brief Description	Correction Manner	#Vars	#Candidates	#Sequences	#UCLID runs	Total Time Total Time	Total Time Reduction	Fixed?
1	Adding an erroneous term	constraint1	7	1	1	5	3m 28s	-	Yes
		constraint2	7	1	2	5	3m 24s	1.9%	Yes
		simulation	7	1	2	5	3m 25s	−0.5%	Yes
2	Removing a correct term	constraint1	7	1	1	5	4m 8s	-	Yes
		constraint2	7	1	2	6	5m 22s	−29.8%	Yes
		simulation	7	1	2	5	5m 4s	5.6%	Yes
3	Adding a “not” operator	constraint1	7	2	1	13	10m 43s	-	Yes
		constraint2	7	1	5	11	6m 32s	39.0%	Yes
		simulation	7	1	5	11	6m 29s	0.8%	Yes
4	Removing a “not” operator	constraint1	7	3	4	22	15m 8s	-	Yes
		constraint2	7	1	4	16	12m 14s	19.2%	Yes
		simulation	7	1	4	12	10m 41s	12.7%	Yes
5	Adding two “not” operators	constraint1	7	3	2	22	15m 26s	-	Yes
		constraint2	7	1	1	4	3m 13s	79.2%	Yes
		simulation	7	1	1	4	3m 15s	−1.0%	Yes
6	Changing an “and” operator to “nor”	constraint1	7	3	8	60	43m 31s	-	Yes
		constraint2	7	1	2	5	3m 30s	92.0%	Yes
		simulation	7	1	2	5	3m 31s	−0.5%	Yes
7	Adding a “not” operator	constraint1	7	6	14	167	> 1h	-	No
		constraint2	7	2	2	14	14m 15s	-	Yes
		simulation	7	2	2	7	11m 46s	17.4%	Yes
8	Adding a “not” operator	constraint1	7	4	11	106	> 1h	-	No
		constraint2	7	1	1	3	2m 15s	-	Yes
		simulation	7	1	1	3	2m 17s	−1.5%	Yes
9	Removing a “not” operator	constraint1	7	7	19	129	> 1h	-	No
		constraint2	7	2	1	13	13m 50s	-	Yes
		simulation	7	2	1	4	10m 53s	21.3%	Yes
10	Negating the result of a term	constraint1	7	4	16	121	> 1h	-	No
		constraint2	7	5	9	98	> 1h	-	No
		simulation	7	9	9	101	> 1h	-	No
						Average Reduction	constraint2	33.6%	
							simulation	6.0% (14.3%)*	

*only cases that #UCLID runs is changed

not activating the buggy location. That was because the first or second instruction in the erroneous sequence of instructions were branch instructions that were taken. Therefore, the bug could not be reached. This means that the formal engine needed more guidance with additional constraints to suggest a solution for fixing the bug.

Considering the above analysis and the reasons for not correcting the bug, we changed the constraints to better guide the formal engine for correction. The changes that we made was to specify a more strict condition for UCLID to more precisely follow the execution of branch instructions, as well as the instructions before and after the branch considering the branch has been taken or not-taken. The results of new constraints are shown in the corresponding rows for the correction manner *constraint*2 in Table 2.

The results of *constraint*2 show substantial improvement over *constraint*1 as follows. The results of new constraints also show the importance of guidance for efficient

and effective correction.

1. With the new constraints, 3 out of the 4 previously uncorrected bugs were successfully corrected.
2. For the other 6 bugs that we could correct with *constraint*1 manner, we could also correct all of them with the new constraints while reducing the correction time by 33.6% on average. Note that only in one case (Bug #2) the correction time increased due to more strict constraints. But, in all the other cases, the correction time decrease.
3. The diagnosis is also improved compared to *constraint*1 manner. As it can be seen in all the cases that are corrected (bugs #1-#9), only 1 or 2 candidates are tried. That is an average 1.2 candidates compared to average of 3 in the *constraint*1 manner.

As explained in Sect. 5.3, there are cases that during the extraction of the truth-table, conflicts happen. To show the effect of using concrete simulation for conflict resolution, as explained in Sect. 5.4, we did the experiments of *constraint*2 manner employing our concrete simulator. The results are shown as *simulation* manner in Table 2.

The results of using concrete simulation for conflict resolution shows two kinds of behaviors among the corrected cases (Bugs #1-#9). For almost half of the corrected bugs (5 out of 9) the concrete simulator did not changed the number of UCLID runs. For the remaining bugs (4 out of 9), the technique has been effective to reduce the number of UCLID runs and consequently the total time for correction.

For the former case, which the number of UCLID runs is not changed, the total time changes around 1%. That negligible change is due to small change in different UCLID run times.

For the latter case, which the number of UCLID runs is reduced, total time is reduced on average by 14.3%. However considering all the 9 bugs together, the reduction of total time becomes 6.0% on average. The results confirm that using the concrete simulator is effective.

As it can be seen from the results, concrete simulation has been less effective than the improved constraints (Constraint2) to reduce the correction time. The first reason is that the conflicting cases does not happen so often. The other reason is that the concrete simulation is performed only when conflicting cases happen. Therefore, it does not affect the selection of the candidates as well as the erroneous sequence generation, which are very important for efficient correction. However, constraints directly affect both candidate selection as well as erroneous sequence generation. Therefore, constraints have been more effective to reduce the total correction time than the concrete simulation.

Regarding bug #10 that is not corrected, as we investigated we found out that not correcting is due to adding a row of the truth-table as 1, that is not correct in general. This means that the row of the truth-table has been actually don't care for the erroneous sequences that we have tried, however, that row should be 0 to be able to correct the design.

7. Conclusions

In this paper, we have presented automatic correction of design bugs in modern processors employing formal methods combined with concrete simulation. Given the cycle-accurate formal specification of an erroneous processor and its reference instruction-set architecture (ISA) model, we employ a multiplexer-based method to find the candidate locations for correction. After that, we find the minterms, or rows of a truth-tables that are 1, of a correcting functions in the candidate location. We have used the truth-table model of the correcting function, which is a very general model. Moreover, we have not explicitly represented the truth-table in the design, which makes the correction model scalable. The possible correction is then formally verified to make sure it is actually correcting the design. We have shown the effectiveness of our method by correcting the bugs in a complex out-of-order superscalar processors. Our experimental results show that our method reduces the model size for correction by 6.0x and total correction time by 12.6x, on average, compared to our previous method. Our future work is to optimize the correcting function and represent the whole corrected function in a more compact way, as well as extending the work to correct in multiple locations.

References

[1] A. Smith, A. Veneris, and A. Viglas, "Design diagnosis using boolean satisfiability," Asia and South Pacific Design Automation Conference (ASP-DAC), pp.218–223, Jan. 2004.
[2] G. Fey, S. Staber, R. Bloem, and R. Drechsler, "Automatic fault localization for property checking," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol.27, no.6, pp.1138–1149, June 2008.
[3] A.M. Gharehbaghi and M. Fujita, "Formal verification guided automatic design error diagnosis and correction of complex processors," International High Level Design Validation and Test Workshop (HLDVT), pp.121–127, Nov. 2011.
[4] A.M. Gharehbaghi and M. Fujita, "Error model free automatic design error correction of complex processors using formal methods," Asian Test Symposium (ATS), pp.143–148, Nov. 2012.
[5] A. Kuehlmann, D.I. Cheng, A. Srinivasan, and D.P. Lapotin, "Error diagnosis for transistor-level verification," Design Automation Conference (DAC), pp.218–224, June 1994.
[6] J.C. Madre, O. Coudert, and J.P. Billon, "Automating the diagnosis and the rectification of design errors with priam," International Conference on Computer-Aided Design (ICCAD), pp.30–33, Nov. 1989.
[7] A. Veneris and I.N. Hajj, "Design error diagnosis and correction via test vector simulation," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol.18, no.12, pp.1803–1816, Dec. 1999.
[8] Y.S. Yang, S. Sinha, A. Veneris, and R.K. Brayton, "Automating logic rectification by approximate SPFDs," Asia and South Pacific Design Automation Conference (ASP-DAC), pp.402–407, May 2007.
[9] Y. Chen, S. Safarpour, J. Marques-Silva, and A. Veneris, "Automated design debugging with maximum satisfiability," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol.29, no.11, pp.1804–1817, Nov. 2010.
[10] V. Boppana, I. Ghosh, R. Mukherjee, J. Jain, and M. Fujita, "Hierarchical error diagnosis targeting RTL circuits," International Conference on VLSI Design (VLSID), pp.436–441, Jan. 2000.

[11] V. Boppana, R. Mukherjee, J. Jain, M. Fujita, and P. Bollineni, "Multiple error diagnosis based on xlists," Design Automation Conference (DAC), pp.660–665, June 1999.
[12] K.H. Chang, I. Wagner, V. Bertacco, and I.L. Markov, "Automatic error diagnosis and correction for RTL designs," International High Level Design Validation and Test Workshop (HLDVT), pp.65–72, Nov. 2007.
[13] M.N. Velev and P. Gao, "Automated debugging of counter-examples in formal verification of pipelined microprocessors," Asia and South Pacific Design Automation Conference (ASP-DAC), pp.689–694, Jan.-Feb. 2012.
[14] R.E. Bryant, S.K. Lahiri, and S.A. Seshia, "Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions," in Computer Aided Verification, ed. E. Brinksma and K.G. Larsen, Lecture Notes in Computer Science, vol.2404, pp.106–122, Springer-Verlag, July 2002.
[15] "Uclid ver. 1.0." available online at: http://www.cs.cmu.edu/˜uclid/.
[16] J. Burch and D. Dill, "Automatic verification of pipelined microprocessor control," Computer Aided Verification, ed. D. Dill, Lecture Notes in Computer Science, vol.818, pp.68–80, Springer-Verlag, June 1994.
[17] C. Price, MIPS IV instruction set. MIPS Technologies Inc., 1995.
[18] "Minisat ver. 2.2." available online at: http://minisat.se/MiniSat.html.

Amir Masoud Gharehbaghi received his B.S., M.S., and Ph.D. degrees in computer engineering from Sharif University of Technology in 1997, University of Tehran in 2000, and Sharif University of Technology, Tehran, Iran in 2007, respectively. He is currently a project assistant professor in Department of Electrical Engineering and Information Systems, The University of Tokyo. Previously, he was a postdoctoral researcher in the VLSI Design and Education Center (VDEC), The University of Tokyo, Tokyo, Japan. His research interests include post-silicon debug, high-level design validation, and microprocessor verification.

Masahiro Fujita received his Ph.D. degree in Engineering from the University of Tokyo in 1985 and shortly after joined Fujitsu Laboratories Ltd. From 1993 to 2000 he had been assigned to Fujitsu's US research office and directed the CAD research group. In March 2000, he joined the department of Electronic Engineering in the University of Tokyo as a professor, and now a professor at VLSI Design & Education Center in the University of Tokyo. He has been involved in many research projects on various aspects of formal verification.