# Functional Test Generation Using Design and Property Decomposition Techniques

HEON-MO KOO and PRABHAT MISHRA
University of Florida

Functional verification of microprocessors is one of the most complex and expensive tasks in the current system-on-chip design methodology. Simulation using functional test vectors is the most widely used form of processor validation. A significant bottleneck in the validation of such systems is the lack of automated techniques for directed test generation. While existing model checking–based approaches have proposed several promising ideas for automated test generation, many challenges remain in applying them to industrial microprocessors. The time and resources required for test generation using existing model checking–based techniques can be prohibitively large. This article presents an efficient test generation technique using decompositional model checking. The contribution of the article is the development of both property and design decomposition procedures for efficient test generation of pipelined processors. Our experimental results using a multi-issue MIPS processor and an industrial processor based on Power Architecture$^{TM}$ Technology demonstrate several orders-of-magnitude reduction in validation effort by drastically reducing both test generation time and test program length.

Categories and Subject Descriptors: B.7.2 [**Hardware**]: Integrated Circuits—*Design Aids*; I.6.7 [**Computing Methodologies**]: Simulation and Modeling—*Simulation Support Systems*

General Terms: Verification, Algorithms

Additional Key Words and Phrases: Model checking, test generation, pipelined processor, design decomposition, property decomposition, functional validation

**ACM Reference Format:**

Koo, H-M. and Mishra, P. 2009. Functional test generation using design and property decomposition techniques. ACM Trans. Embedd. Comput. Syst. 8, 4, Article 32 (July 2009), 33 pages. DOI = 10.1145/1550987.1550995 http://doi.acm.org/10.1145/1550987.1550995

## 1. INTRODUCTION

Functional verification is widely acknowledged as a major bottleneck in microprocessor design methodology: up to 70% of the design development time

This work was partially supported by grants from Intel Corporation and NSF CAREER award 0746261.

Author's address: H.-M. Koo and P. Mishra, Department of Computer and Information Science and Engineering, University of Florida, Gainesville FL 32611, email: {hkoo, prabhat}@cise.ufl.edu



DOI 10.1145/1550987.1550995 http://doi.acm.org/10.1145/1550987.1550995

and resources are spent on functional verification [Fine and Ziv 2003]. Existing processor validation techniques employ a combination of simulation-based techniques and formal methods. Simulation-based validation is the most widely used form of processor verification using test programs. There are three types of test generation techniques: random, directed, and constrained-random. The directed tests can reduce overall validation effort, since shorter tests can obtain the same coverage goal compared to the random tests. A significant bottleneck in processor validation is the lack of automated tools and techniques for directed test generation.

Model checking–based test generation has been introduced as a promising approach for pipelined processor validation [Mishra and Dutt 2004; 2005] due to its capability of automatic test generation. In this approach, a set of properties are generated from the specification based on functional coverage. Then, the design and a property (negated version) are applied to the model checker to produce a counterexample (test). However, this approach is unsuitable for large designs due to state explosion problem in unbounded model checking (UMC). Therefore, it is necessary to reduce the counterexample search space in terms of the model of design as well as the applied properties.

As a complementary technique for test generation, the bounded model checking (BMC) is also promising by restricting the search space within a fixed number ($k$) of transitions, called bound. The basic idea is to unroll the model of design $k$ times, and then to convert the BMC problem into a propositional satisfiability problem. A satisfiability (SAT) solver is used to find a satisfiable assignment of variables that is converted into a counterexample. However, finding exact bound is a challenging problem, since the depth of counterexamples is unknown in general before applying the property. Choosing an incorrect bound increases test generation time and memory requirement. In the worst case, test generation may not be possible.

This article presents an efficient test generation technique that addresses both challenges mentioned earlier: (i) state explosion problem in UMC and (ii) bound determination in BMC. To address the first challenge, it is necessary to reduce the counterexample search space in terms of the model of the design as well as the applied properties. We propose an efficient test generation technique using both design and property decompositions to tackle the state explosion for complex designs. To address the second challenge, we have developed a method for determining the bound for each property based on the graph model of pipelined processors. We also present a novel algorithm for merging local (partial) counterexamples (generated due to decomposed design and properties) to generate the global counterexample (test).

The rest of the article is organized as follows. Section 2 briefly describes the existing model checking-based test generation approaches. Section 3 presents related work addressing test generation in the context of functional validation of pipelined processors. Section 4 describes our test generation methodology using decompositional model checking as well as SAT-based BMC. Section 5 presents case studies using two processors: test generation for a multi-issue MIPS processor and microarchitectural test generation for an industrial processor based on Power Architecture™ Technology. Finally, Section 6 concludes the article.

## 2. BACKGROUND AND PRELIMINARIES

Model checking is a formal method that can verify whether a temporal property is satisfied for a finite state concurrent system. Given a finite state system $M$ and a temporal property $p$, the model checking algorithm will check whether $M$ satisfies $p$, that is, $M \models p$. If the property holds, the algorithm will return true, otherwise a counterexample will be reported. Symbolic Model Verifier [SMV] is a popular model checking software that accepts SMV model of the design and a temporal logic property as inputs and verifies whether the design satisfies the property. To enable model checking, the design specification is abstracted to a transition graph model (Kripke structure [Clarke et al. 1999]). A Kripke structure $M$ is a quadruple $M = (S, I, T, L)$, where $S$ is the set of states, $I$ is the set of initial states, $T \subseteq S \times S$ is the transition relation, and $L : S \rightarrow P(A)$ is the labeling function, where A is the set of atomic propositions, and P(A) denotes the powerset over A. Labeling is a way to attach observations to the system: For a state $s \in S$, the set L(s) is made of the atomic propositions that hold in $s$.

The model checking problem can be considered as a reachability problem. The forward reachability algorithm starts at the initial state and calculates the next image, which is a set of states reachable in one step, based on the current image. The algorithm will stop in one of the following two cases: (i) the property is falsified for some states, so the counterexample will be generated, or (ii) all the state space is explored and no state violates the property. Generally, binary decision diagrams (BDDs) [Bryant 1986] are efficient data structures to represent and manipulate the transition relation of the finite-state model. However, they are not scalable to handle the large practical systems. As a complementary framework of BDDs, model checking algorithms based on Boolean SAT procedures [Prasad et al. 2005] have emerged as a promising approach, especially for the BMC.

The Boolean SAT problem is to figure out if the given Boolean formula has a satisfiable assignment. Usually this formula will be transformed into conjunctive normal form (CNF) and is checked by the SAT solver to determine the result. Several SAT solvers such as GRASP [Marques-Silva and Sakallah 1999] and Chaff [Moskewicz et al. 2001] adopt the popular DPLL algorithm. The DPLL algorithm can be improved by using the conflict clause forwarding and heuristic ordering of the variable assignment. BMC [Biere et al. 1999] is a technique that can prove if there is a counterexample for the property within a given bound. Given a model $M$, a safety property $p$, and a bound $k$, BMC will unfold the model $k$ times and encode it as the following logic formula:

$$BMC(M, p, k) = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg p(s_i) \tag{1}$$

This formula can be transformed to the CNF and checked by a SAT solver. If there is a satisfiable assignment, then the property is false and the assignment (counterexample) will be reported, which means $M \not\models_k p$. Otherwise, it means that in this model there is no counterexample with length $k$ for this property, written as $M \models_k p$.

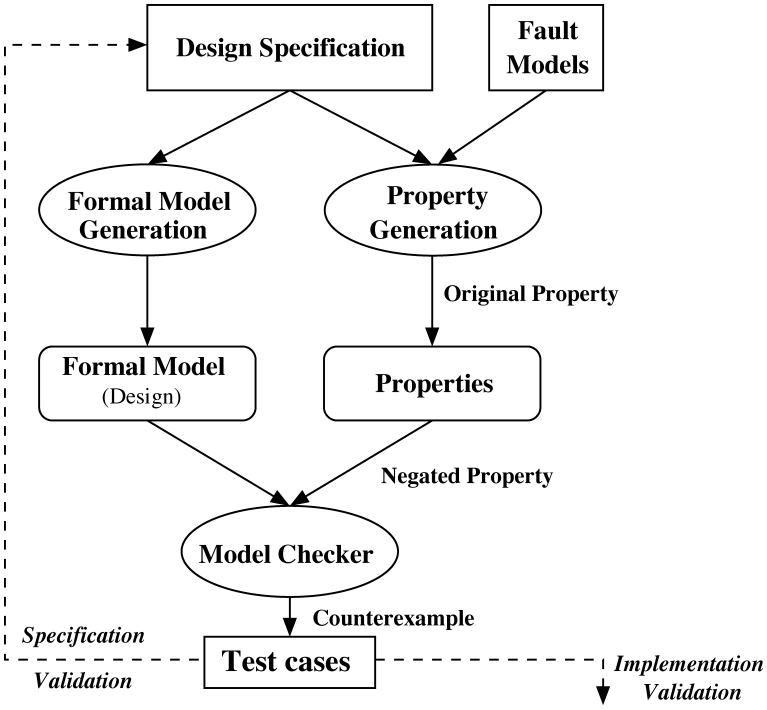


Fig. 1. Test generation using model checking.

## 2.1 Test Generation Using Model Checking

Figure 1 shows specification-driven test generation methodology using model checking [Mishra and Dutt 2004]. The design specification is translated to a formal model (e.g., SMV description [Clarke et al. 1999]). Next, the properties in the form of computation tree logic or linear temporal logic formulas can be generated based on the fault models [Mishra and Dutt 2005]. For example, in a pipelined processor, if the fault model is related to "pipeline interactions," one property will be generated for activating each pipeline interaction. Finally, the properties (negated version) are applied on the formal model using model checker to generate required test cases (counterexamples). Since we assume that both design and properties are correct and we use negated version of the properties, model checker will always generate a valid counterexample unless it faces state space explosion problem due to design/property complexity. The generated test cases can be used for validation of both specification and implementation.

Algorithm 1 outlines the three important steps in model checking-based test generation (shown in Figure 1): (i) formal model generation, (ii) property generation and negation, and (iii) test generation. As indicated earlier, one property is generated for activating each fault in the given fault model. This algorithm takes the model M generated from design specification and properties derived form the faults $F$ as inputs and generates test suite extracted from the counterexamples. For each fault $F_i$, one test case is generated. The algorithm iterates

**Algorithm 1**: *Test Generation using Model Checking*
**Inputs**: i) Design Specification, $S$
ii) Set of faults/interactions, $F$ (based on the coverage criterion)
**Outputs**: Test suite
Begin
TestSuite = $\phi$;
$M$ = CreateDesignModel ($S$);
**for** each fault $F_i$ in the set $F$
$P_i$ = CreateProperty($F_i$) ;
$\overline{P_i}$ = Negate($P_i$) ;
$test_i$ = ModelChecking($\overline{P_i}$, $M$) ;
TestSuite = TestSuite $\cup$ $test_i$ ;
**endfor**
**return** TestSuite ;
End

until all the faults in the fault model are covered. In each iteration, each fault $F_i$ is transformed to a temporal logic property $P_i$. The generated property is referred as original property that is supposed to verify the absence of that fault in the design. Next, each property is negated. The negated version of the property is referred as negated property. Finally, model checking is applied using the model $M$ and negated property $\overline{P_i}$ to produce the required counterexample. The generated test is supposed to activate the fault.

For example, to activate a fault in the stall functionality of a decode (ID) unit in a pipelined processor, the system will generate the property *"assert G(ID._stall = 1)*. The negation of the property will be *assert G(ID._stall = 0)*. Once model checker receives the negated property and the processor model as inputs, it will generate a counterexample to stall the decode unit which can be used as a test case for the original property *assert G(ID._stall = 1)*. The counterexample contains a sequence of instructions (test program) from an initial state to a state where the negated version of the property fails. As mentioned earlier, this approach is unsuitable for large designs due to the state explosion problem in model checking. We propose an efficient test generation technique using both design and property decompositions to tackle the state explosion problem for complex designs.

### 2.2 Test Generation Using SAT-based BMC

BMC is a promising approach for test generation, since it restricts the search space that is reachable from initial states within a fixed number of transitions, called bound. Algorithm 2 describes the test generation procedure using BMC. This algorithm is similar to Algorithm 1 except that it has an additional step for determining exact bound for each property. Algorithm 2 iterates until all the faults in the fault model are covered. In each iteration, each fault $F_i$ is transformed to a temporal logic property $P_i$. Next, bound $k_i$ for each property is decided. The bound for each property is the minimum depth (from the initial state)

**Algorithm 2**: *Test Generation using BMC*
**Inputs**: i) Design Specification, $S$
ii) Set of faults/interactions, $F$ (based on the coverage criterion)
**Outputs**: Test suite
Begin
TestSuite = $\phi$;
$M$ = CreateDesignModel ($S$);
**for** each fault $F_i$ in the set $F$
$P_i$ = CreateProperty($F_i$) ;
$bound_i$ = DetermineBound($M$, $\overline{P_i}$) ;
$test_i$ = BounedModelChecking($\overline{P_i}$, $M$, $bound_i$) ;
TestSuite = TestSuite $\cup$ $test_i$ ;
**endfor**
**return** TestSuite ;
End

to find the required counterexample. Finally, SAT-based BMC takes model $M$, negated property $P_i$, and bound $k_i$ as inputs and generates a counterexample.

As mentioned earlier, it is a major challenge to determine the exact bound for BMC. If the bound is known in advance, SAT-based BMC is typically more effective for falsification than model checking because search for counterexample is faster and SAT capacity reaches beyond BDD capacity [Biere et al. 1999]. Choosing an incorrect bound increases test generation time and memory requirement. In the worst-case, test generation may not be feasible. For example, we can increase the bound iteratively starting from a small bound until a counterexample is found. This approach is advantageous for shallow counterexamples, but disadvantageous for deep counterexamples due to accumulation of iterative running time. Alternatively, a large bound can be used such that all counterexamples are found. This approach loses the benefits of BMC due to search in a large number of irrelevant states. Therefore, the efficiency of test generation closely depends on the techniques of deciding the bound. We propose a method for determining the exact bound for each property based on the graph model of pipelined processors.

## 3. RELATED WORK

Traditionally, validation of microprocessors has been performed by applying a combination of random and directed test programs using simulation techniques. There are many successful test generation frameworks in industry today. Genesys-Pro [Adir et al. 2004], used for functional verification of IBM processors, combines architecture and testing knowledge for efficient test generation. In Piparazzi [Adir et al. 2003], a model of microarchitectural processor and the user's specification are converted into a constraint satisfaction problem and the dedicated constraint satisfaction problem solver is used to construct an actual test program. Many techniques have been proposed for directed test

program generation based on an instruction tree traversal [Aharon et al. 1995], microarchitectural coverage [Koo et al. 2006; Ur and Yadin 1999], and functional coverage using Bayesian networks [Fine and Ziv 2003]. Recently, Gluska [2006] described the need for coverage-directed test generation in coverage-oriented verification of the Intel Merom microprocessor. None of these techniques can automatically generate directed tests based on a comprehensive functional coverage metric. In other words, these techniques either generate constrained random testcases automatically, or they generate directed tests for specific scenarios in semiautomated fashion. Our approach can automatically generate the required directed tests to achieve a given functional coverage goal.

Several formal model-based test generation techniques have been developed for validation of pipelined processors. In FSM-based test generation, FSM coverage is used to generate test programs based on reachable states and state transitions [Campenhout et al. 1999; Iwashita et al. 1994; Kohno and Matsumoto 2001; Ho et al. 1995]. Since complicated microarchitectural mechanisms in modern processor designs include interactions among many pipeline stages and buffers, the FSM-based approaches suffer from the state space explosion problem. To alleviate the state explosion, Utamaphethai et al. [2000] have presented an FSM model partitioning technique based on microarchitectural pipeline storage buffers. Shen and Abraham [2000] have proposed an RTL abstraction technique that creates an abstract FSM model while preserving clock accurate behaviors. Wagner et al. [2005] have presented a Markov model-driven random test generator with activity monitors that provides assistance in locating hard-to-find corner case design bugs and performance problems. Due to the state space explosion problem, these techniques are not applicable for directed test generation in pipelined processors. Our approach addresses the state space explosion problem by using design an property decompositions during model checking-based test generation.

Functional validation can be performed by using two approaches: model-driven approach or model checking–based approach. Model-driven methods [Mathaikutty et al. 2007b; Mathaikutty et al. 2007a; Patel et al. 2007] capture the essence of the design as a functional (specification) model and tests generated from this model are employed in the validation of the system implementation. Model checking [Clarke et al. 1999] has been successfully used in verification of proving processor properties. Ho et al. [1998] extract controlled token nets from a logic design to perform efficient model checking. Jacobi [2002] used a methodology to verify out-of-order pipelines by combining model checking for the verification of the pipeline control, and theorem proving for the verification of the pipeline functionality. Compositional model checking is used to verify a processor microarchitecture containing most of the features of a modern microprocessor [Jhala and McMillan 2001]. Parthasarathy et al. [2004] have presented a safety property verification framework using sequential SAT and BMC. Model checking-based techniques are also used in the context of falsification by generating counterexamples. Clarke et al. [1995] have presented an efficient algorithm for generation of counterexamples and witnesses in symbolic model checking. Bjesse et al. [2004] have used counterexample guided abstraction refinement to find complex bugs. Automatic test generation techniques

using model checking have been proposed in software [Gargantini and Heitmeyer 1999] as well as in hardware validation [Mishra and Dutt 2002]. However, traditional model checking-based techniques do not scale well due to the state space explosion problem. To reduce the test generation time and memory requirement, Mishra and Dutt [2004; 2005] have proposed a design decomposition technique at the module level when the original property contains variables for only a single module. However, their technique does not handle properties that have variables from multiple modules. Such properties are common in test generation. Our framework allows such input properties by decomposing the properties as well as the model of the pipelined processor.

As a complementary technique of model checking, Biere et al. [1999] introduced BMC combined with SAT solving. The recent developments in SAT-based BMC techniques have been presented in Prasad et al. [2005]. BMC is an incomplete method that cannot guarantee a true or false determination when a counterexample does not exist within a given bound. However, once the exact bound of a counterexample is known, large designs can be falsified very fast, since SAT solvers [Goldberg and Novikov 2002; Moskewicz et al. 2001] do not require exponential space, and searching counterexample, in an arbitrary order consumes much less memory than breadth first search in model checking. Amla et al. [2005] have analyzed the performance of bounded and unbounded algorithms using a set of industrial benchmarks. The capacity increase of the BMC technique has become attractive for industrial use. An Intel study [Copty et al. 2001] showed that BMC has better capacity and productivity over UMC for real designs taken from the Pentium-4 processor. SAT-based BMC can be used as a test generation engine due to its capacity and performance if the bound is selected appropriately. A major challenge in these approaches is how to determine the exact bound. Incremental SAT solvers [Jin and Somenzi 2005; Whittemore et al. 2001; Strichman 2001] try to mitigate the impact of choosing an incorrect initial bound by exploiting similarity and forwarding conflict clauses, but they are disadvantageous for deep counterexamples due to the accumulation of iterative running time. We propose a method to determine the bound for each test generation scenario, thereby making SAT-based BMC useful for directed test generation in pipelined processors.

## 4. TEST GENERATION USING DESIGN AND PROPERTY DECOMPOSITIONS

This section presents our proposed approach that uses both design and property level decompositions. Figure 2 shows our functional test generation methodology. The design model can be generated from the architecture specification or can be developed by the designers. Similarly, the properties can be generated from the specification based on a functional coverage metric such as graph coverage or pipeline interaction coverage. Additional properties can be added based on interesting scenarios and corner cases. For efficient test generation, we decompose the properties as well as the design model. Our framework uses both UMC and BMC to generate partial counterexamples for the partitioned designs and decomposed properties. These partial counterexamples are integrated to construct the final test program.

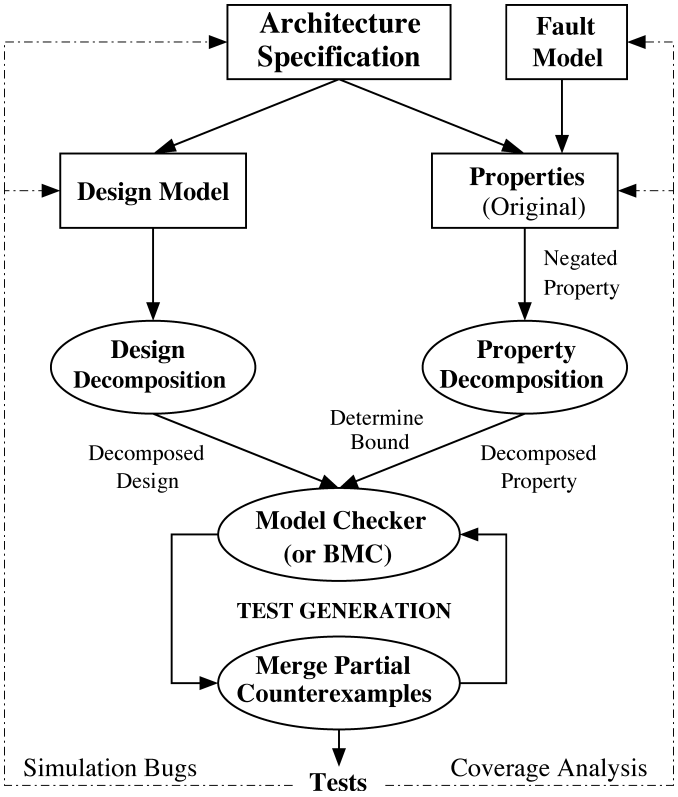


Fig. 2. Proposed test generation methodology using design and property decompositions.

This methodology has seven important steps: (i) design model generation, (ii) generation and negation of properties, (iii) design decomposition, (iv) property decomposition, (v) determination of bound for each property, (vi) test generation using model checking as well as SAT-based BMC, and (vii) merging partial (local) counterexamples to generate the global counterexample (test). The last two steps work in an integrated fashion (discussed in Section 4.6) to ensure that the generated testcases can activate the intended faults (errors). The remainder of this section describes each of these seven steps in detail.

Algorithm 3 outlines the major steps in our test program generation methodology shown in Figure 2. This algorithm takes the processor model $M$ and a set of desired faults (e.g., pipeline interaction faults) $F$ as inputs and generates a set of test programs. Each interaction is converted and negated into a temporal logic property. The exact bound for BMC is determined for each property. The design model, the negated version of the property and the required bound are applied to our decompositional model checking framework to generate the test program for the property. The algorithm iterates over all the faults based on the functional coverage and corner cases. Section 4.1 describes a graph-based modeling of pipelined processors. The property generation based on pipeline interaction coverage is described in Section 4.2. The design and

Table I. Design and Property Decomposition Scenarios

| Design | Property | Comments |
|---|---|---|
| Original | Original | Traditional model checking |
| Original | Partitioned | Merging of counterexamples is not always possible |
| Partitioned | Original | Similar to traditional model checking |
| Partitioned | Partitioned | Our approach, both property and design decompositions |

property decomposition techniques are described in Sections 4.3 and 4.4, respectively. Section 4.5 presents a technique to determine the exact bound for finding counterexamples for a given property. Model checking as well as SAT-based BMC are used to generate partial counterexamples for the partitioned modules and properties.

**Algorithm 3**: *Test Generation using Design and Property Decompositions*
**Inputs**: i) Design Specification, $S$
ii) Set of faults/interactions $F$ based on functional coverage and corner cases
**Outputs**: Test programs
Begin
TestPrograms = $\phi$
$M$ = CreateDesignModel ($S$);
**for** each fault $F_i$ in the set $F$
$P_i$ = CreateProperty($F_i$)
$bound_i$ = DecideBound($P_i$)
$\overline{P_i}$ = Negate($P_i$)
$test_i$ = DecompositonalModelChecking($\overline{P_i}$, $M$, $bound_i$)
TestPrograms = TestPrograms $\cup$ $test_i$
**endfor**
**return** TestPrograms
End

Integration of these partial counterexamples is a major challenge due to the fact that the relationships among decomposed modules and subproperties may not be preserved at the top level. We propose a time step-based integration of partial counterexamples to construct the final test program. Section 4.6 presents our test generation technique based on decompositional model checking. Section 4.7 presents a conflict resolution technique during merging of partial counterexamples.

It is important to note that the property and design decompositions are not independent. Table I shows four possible scenarios of design and property decompositions. The first scenario indicates the traditional model checking where original property is applied to the whole design. The second scenario implies that the decomposed properties are applied to the whole design. In certain applications, this may improve overall model checking efficiency. However, in general this procedure is not applicable, since merging counterexamples may not generate the expected result. Consider an example property that can be used

for generating a test to activate two simultaneous unit stalls. The property can be decomposed to generate two subproperties. These subproperties may generate counterexamples to stall the respective units in a pipelined processor, but the combined test program may not simultaneously stall both the units. Furthermore, we cannot avoid the state explosion problem without design decomposition. The third scenario is meaningless, since design decomposition is not useful if the original property is not applicable to the partitioned design components. The last scenario depicts our approach where both design and properties are partitioned.

### 4.1 Generation of Design Model from the Specification

The first step in our test generation methodology is to generate formal model of the design from the architecture specification. Modeling plays a central role in the generation of efficient test programs. Ideally, the design should be decomposed into components such that there is very little interaction among the partitioned components. For a pipelined processor the natural partition is along the pipeline boundaries. In other words, the partitioned pipelined processor can be viewed as a graph where nodes consist of units (e.g., fetch, decode) or storages (e.g., memory or register file) and edges consist of connectivity among them. Typically, instruction is transferred between units, and data is transferred between units and storages. This graph model is similar to the pipeline level block diagram available in a typical architecture manual. The graph model can be extracted from the architecture description language [Mishra and Dutt 2008] specification of pipelined processors. This section presents graph models for MIPS and e500 processors.

#### Example 1: Modeling of MIPS Processor

For illustration, we use a simplified version of the multi-issue MIPS processor [Hennessy and Patterson 2003]. Figure 3 shows the graph model of the processor that can issue up to four operations (an integer ALU operation, a floating-point addition operation, a multiply operation, and a divide operation). In the figure, rectangular boxes denote units, dashed rectangles are storages, bold edges are instruction-transfer (pipeline) edges, and dashed edges are data-transfer edges. A path from a root node (e.g., Fetch) to a leaf node (e.g., WriteBack) consisting of units and pipeline edges is called a pipeline path. For example, one of the pipeline path is {*Fetch, Decode, IALU, MEM, WriteBack*}. A path from a unit to main memory or register file consisting of storages and data-transfer edges is called a data-transfer path. For example, {*MEM, DataMemory, MainMemory*} is a data-transfer path.

#### Example 2: Modeling of e500 Processor

Figure 4 shows a functional graph model of the four-wide superscalar commercial e500 processor-based on the Power Architecture™ Technology[1] [e500

[1]The Power Architecture and Power.org wordmarks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org

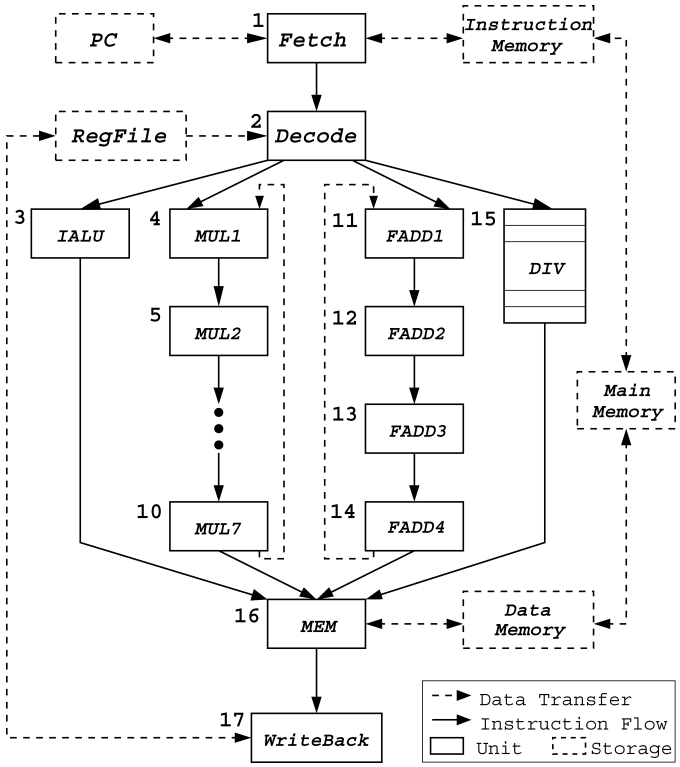


Fig. 3. Graph model of the MIPS processor.

Manual 2005] with seven pipeline stages. We have developed a processor model based on the microarchitectural structure, the instruction behavior, and the rules in each pipeline stage that determine when instructions can move to the next stage and when they cannot. The microarchitectural features in the processor model include pipelined and clock-accurate behaviors such as multiple issue for instruction parallelism, out-of-order execution and in-order completion for dynamic scheduling, register renaming for removing false data dependency, reservation stations for avoiding stalls at Fetch and Decode pipeline stages, and data forwarding for early resolution of read-after-write (RAW) data dependency.

### 4.2 Generation and Negation of Properties

Today's test generation techniques and formal methods are very efficient to find logical bugs in a single module. Hard-to-find bugs arise often from the intermodule interactions among many pipeline stages and buffers of modern processor designs. In this article, we primarily focus on such hard-to-verify interactions among modules in a pipelined processor. If we consider the graph model of the pipelined processor, the pipeline interactions imply the interactions between the nodes in the graph model. In this article all the properties are specified using temporal logic, since the functional coverage model ("pipeline interaction") that we have adopted in this article does not require complex property

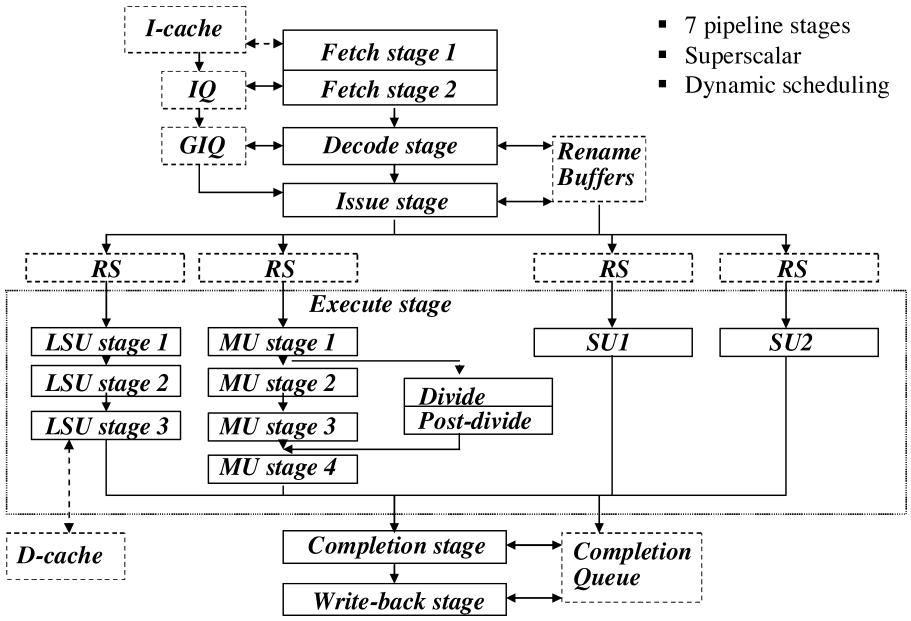


Fig. 4. Instruction pipeline of e500 processor.

specifications supported in Property Specification Language (PSL). When a different or complex coverage model is employed PSL-based specification may be required. However, the overall flow presented in this article will remain the same in the presence of properties specified using PSL, since the existing model checkers support PSL property specifications [Tuerk et al. 2007].

We first define the possible pipeline interactions based on the number of nodes in the graph model and the average number of activities in each node. For example, an IALU node can have four activities: operation execution, stall, exception, and no operation (NOP). In general, the number of activities for a node will be different based on what activity we would like to test. For example, execution of ADD and SUB operations can be treated as the same activity because they go through the same pipeline path. Separation of them into different activities will refine the functional tests but increase the test generation complexity. Furthermore, the number of activities may vary for different nodes.

CLAIM 1. *In a graph model with n nodes where each node can have on average r activities, a total of* $((1 + r)^n - 1)$ *properties are required to verify all interactions.*

PROOF. Since we generate one property for each interaction, the total number of properties is the same as the total of interactions. To compute the total number of interactions, we can compute the summation of all the scenarios. We also include the scenario for no interaction in this computation. If we consider no interactions, there are $(n \times r)$ test programs necessary. In the presence of one interaction, we need $({}_nC_2 \times r^2)$ test programs for possible combination of two nodes. Here, ${}_nC_i$ denotes the ways of choosing $i$ nodes from $n$

nodes. Based on this model, the total number of interactions will be as shown in Eq. (2). Therefore, the total number of interactions (properties) is equal to $(n \times r +_n C_2 \times r^2 + \ldots +_n C_n \times r^n)$, which is equal to $((1+r)^n - 1)$. □

$$\sum_{i=1}^{n} {}_nC_i \times r^i \tag{2}$$

Although the total number of interactions can be extremely large, in reality, the number of simultaneous interactions can be small and many other realistic assumptions can reduce the number of properties to a manageable one. The generated properties are expressed in linear temporal logic (LTL) [Clarke et al. 1999], where each property consists of temporal operators ($G, F, X, U$) and Boolean connectives ($\wedge$, $\vee$, $\neg$, and $\rightarrow$). We generate a property for each pipeline interaction from the specification. Since pipeline interactions at a given cycle are semantically explicit and our processor model is organized as structure-oriented modules, pipeline interactions can be converted in the form of a property such as $F(p_1 \wedge p_2 \wedge \ldots \wedge p_n)$ that combines activities $p_i$ over $n$ modules using logical *AND* operator. The atomic proposition $p_i$ is a functional activity at a node $i$ such as operation execution, stall, exception or NOP. The property is true when all the $p_i$'s ($i = 1$ to $n$) hold at some time step. Since we are interested in counterexample generation, we need to generate the negation of the property first. The negation of the properties can be expressed as:

$$\begin{aligned} \neg X(p) &= X(\neg p) \\ \neg G(p) &= F(\neg p) \\ \neg F(p) &= G(\neg p) \\ \neg pUq &= pR\neg q \end{aligned} \tag{3}$$

For example, the negation of $F(p_1 \wedge p_2 \wedge \ldots \wedge p_n)$, interaction fault, can be described as $G(\neg p_1 \vee \neg p_2 \vee \ldots \vee \neg p_n)$ whose counterexamples will satisfy the original property.

### 4.3 Design Decomposition

It is important to note that the design decomposition is dependent on the property decomposition. As discussed in Section 4.1, the pipelined processor can be partitioned into modules. However, we need to change the partitioning policy based on the properties. It is hard to decompose the properties when they are spread across multiple modules or in the complicated forms such as $pUq$, $F(p \rightarrow G(q))$, $G(p \rightarrow F(q))$, and so on. For example, a property related to checking data-forwarding path is not decomposable based on a module-level partitioning, but it may be decomposable based on a pipeline path-level partitioning.

We consider three partitioning techniques: module-level, path-level, and stage-level partitioning. Module (or node) level partitioning gives the lowest level of granularity in the graph model. For example, in Figure 3, the integer-ALU pipeline path {*Fetch, Decode, IALU, Mem, WriteBack*} is treated as one of the path level partitions. Similarly, the multiplier path, the floating-point adder path, and the divider path are other examples of path-level partitioning for the

MIPS processor in Figure 3. Stage-level partitioning is determined by the distance from the root node (e.g., *Fetch*). In general, various forms of design and property partitioning are possible and different graph clustering algorithms can be used to find different design partitions for a given property decomposition. Section 4.6 describes two design partitioning techniques using illustrative examples (Examples 3 and 4).

## 4.4 Property Decomposition

Various combinations of temporal operators and Boolean connectives are possible to express desired properties in temporal logic. If the properties are decomposable, the partial counterexamples generated from the decomposed properties can be used for generating a counterexample of the original property. However, not all properties are decomposable, and in certain situations, decompositions are not beneficial compared to traditional model checking-based test generation. In this section, we describe how to decompose these properties (already negated) with respect to generation of a counterexample. We assume that a set of counterexamples always exist for the property, since it is the negated version of the original property and the design is assumed to be correct.

4.4.1 *Decomposable Properties.* The following types of properties allow simple decompositions. Lemmas 4.1 through 4.4 prove that the decomposed properties can be used for test generation.

$$\begin{aligned} G(p \wedge q) &= G(p) \wedge G(q) \\ F(p \vee q) &= F(p) \vee F(q) \\ X(p \vee q) &= X(p) \vee X(q) \\ X(p \wedge q) &= X(p) \wedge X(q) \end{aligned} \tag{4}$$

LEMMA 4.1. *Counterexamples of the decomposed properties $G(p)$ and $G(q)$ can be used to generate a counterexample of $G(p \wedge q)$.*

PROOF. Let $C_{G(p)}$ denote the set of counterexamples for $G(p)$ that satisfies $F(\neg p)$, $C_{G(q)}$ denote the set of counterexamples for $G(q)$ that satisfies $F(\neg q)$, and $C_{G(p \wedge q)}$ denote the set of counterexamples for $G(p \wedge q)$ that satisfies $F(\neg p \vee \neg q)$. Since $F(\neg p \vee \neg q) = F(\neg p) \vee F(\neg q)$, so the sets $C_{G(p)}$ and $C_{G(q)}$ are subsets of $C_{G(p \wedge q)}$, that is, $C_{G(p)} \cup C_{G(q)} \equiv C_{G(p \wedge q)}$. Therefore, any counterexample of the decomposed properties $G(p)$ or $G(q)$ can be used as a counterexample of $G(p \wedge q)$. □

LEMMA 4.2. *Counterexamples of the decomposed properties $F(p)$ and $F(q)$ can be used to generate a counterexample of $F(p \vee q)$.*

PROOF. Since $G(\neg p \wedge \neg q) = G(\neg p) \wedge G(\neg q)$, the set $C_{F(p \vee q)}$ is equal to the intersection set between $C_{F(p)}$ and $C_{F(q)}$, that is, $C_{F(p)} \cap C_{F(q)} \equiv C_{F(p \vee q)}$. Therefore, a common counterexample between $F(p)$ and $F(q)$ can be used as a counterexample of $F(p \vee q)$. □

LEMMA 4.3. *Counterexamples of the decomposed properties $X(p)$ and $X(q)$ can be used to generate a counterexample of $X(p \wedge q)$.*

PROOF. Since $X(\neg p \vee \neg q) = X(\neg p) \vee X(\neg q)$, the sets $C_{X(p)}$ and $C_{X(q)}$ are subsets of $C_{X(p \wedge q)}$, that is, $C_{X(p)} \cup C_{X(q)} \equiv C_{X(p \wedge q)}$. Therefore, any counterexample of the decomposed properties $X(p)$ or $X(q)$ can be used as a counterexample of $X(p \wedge q)$. □

LEMMA 4.4. *Counterexamples of the decomposed properties $X(p)$ and $X(q)$ can be used to generate a counterexample of $X(p \vee q)$.*

PROOF. Since $X(\neg p \wedge \neg q) = X(\neg p) \wedge X(\neg q)$, the set $C_{X(p \vee q)}$ is equal to the intersection set between $C_{X(p)}$ and $C_{X(q)}$, $C_{X(p)} \cap C_{X(q)} \equiv C_{X(p \vee q)}$. Therefore, a common counterexample between $X(p)$ and $X(q)$ can be used as a counterexample of $X(p \vee q)$. □

4.4.2 *Nondecomposable Properties.* It is important to note that the property decomposition is not possible in various scenarios when the combination of decomposed properties is not logically equivalent to the original property. For example, $F(p \wedge q) \neq F(p) \wedge F(q)$, and $G(p \vee q) \neq G(p) \vee G(q)$. However, with respect to test generation, the counterexamples of the decomposed properties can be used to generate a counterexample of the original property as described below.

The property $F(p \wedge q)$ is true when both $p$ and $q$ hold at the same time step. But $F(p) \wedge F(q)$ is true even when $p$ and $q$ hold at different time steps. Therefore, $F(p \wedge q) \neq F(p) \wedge F(q)$. However, we can use $F(p)$ or $F(q)$ for test generation to activate the property $F(p \wedge q)$ based on the following Lemma 4.5.

LEMMA 4.5. *Counterexamples of the decomposed properties $F(p)$ and $F(q)$ can be used to generate the counterexample of $F(p \wedge q)$.*

PROOF. Since the relation between $F(p \wedge q)$ and $F(p) \wedge F(q)$ is $F(p \wedge q) \rightarrow F(p) \wedge F(q)$, $C_{F(p \wedge q)} \supset (C_{F(p)} \cup C_{F(q)})$. Therefore, any counterexample of the decomposed properties $F(p)$ or $F(q)$ is a counterexample of $F(p \wedge q)$. □

The property $G(p \vee q)$ is true when either $p$ or $q$ holds at every time step. But $G(p) \vee G(q)$ is true either when $p$ holds at every time step or when $q$ holds at every time step. Therefore, $G(p \vee q) \neq G(p) \vee G(q)$. In this case, the counterexamples of the decomposed properties $G(p)$ and $G(q)$ cannot directly be used to generate a counterexample of $G(p) \vee G(q)$, since $G(p) \vee G(q) \rightarrow G(p \vee q)$, that is, $(C_{G(p)} \cap C_{G(q)}) \supset C_{G(p \vee q)}$. In other words, not all common counterexamples of $G(p)$ and $G(q)$ can be used as a counterexample of $G(p \vee q)$. Furthermore, it is hard to know whether the common counterexamples of $G(p)$ and $G(q)$ belong to $C_{G(p \vee q)}$. However, introducing the notion of clock allows the decomposed properties to produce a counterexample of $G(p \vee q)$ as described in Lemma 4.6.

LEMMA 4.6. *Counterexamples of $G(p)$ and $G(q)$ can be used to generate a counterexample of $G(p \vee q)$ by introducing a specific time step.*

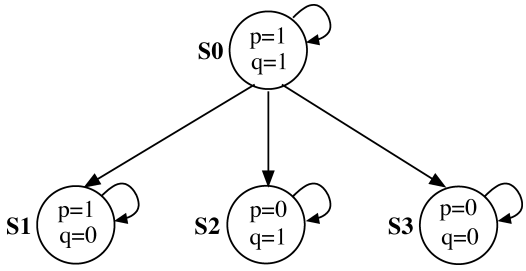


Fig. 5. An example of Kripke structure model.

PROOF. The relation between $G(p \vee q)$ and $G(p) \vee G(q)$ with time step is $G((clk \neq t_s) \vee (p \vee q)) = G((clk \neq t_s) \vee p) \vee G((clk \neq t_s) \vee q)$ because both sides are evaluated to be true when $(clk \neq t_s)$, or when $(clk = t_s)$ and $p = true$ or $q = true$. Therefore, $C_{G((clk \neq t_s) \vee (p \vee q))} \equiv (C_{G((clk \neq t_s) \vee p)} \cap C_{G((clk \neq t_s) \vee q)})$. □

For example, Figure 5 describes a Kripke structure [Clarke et al. 1999] with four states $s0, s1, s2$, and $s3$, where $s0$ is the only initial state. The structure has three transitions: $(s0, s1)$, $(s0, s2)$, $(s0, s3)$, and self-loop in each state. There are two local variables $p$ for $module1$ and $q$ for $module2$ : $p$ holds on states $\{s0, s1\}$ and $q$ holds on states $\{s0, s2\}$. Assuming the original property $F(p = 0 \wedge q = 0)$, we add a specific time step as $F(clk = t_s \wedge p = 0 \wedge q = 0)$[2] and its negation will be $G(clk \neq t_s \vee p = 1 \vee q = 1)$. Let us assume that $t_s$ = 2. The following shows a set of counterexamples of $G(clk \neq 2 \vee p = 1 \vee q = 1)$ for the entire model:

$$C_M = \{(s0, s0, s3), (s0, s3, s3)\}$$

The following shows a set of counterexamples of $G(clk \neq 2 \vee p = 1)$ for $module1$:

$$C_{m1} = \{(s0, s0, s2), (s0, s0, s3), (s0, s2, s2), (s0, s3, s3)\}$$

The following shows a set of counterexamples of $G(clk \neq 2 \vee q = 1)$ for $module2$:

$$C_{m2} = \{(s0, s0, s1), (s0, s0, s3), (s0, s1, s1), (s0, s3, s3)\}$$

We can see that $C_{m1} \cap C_{m2} = \{(s0, s0, s3), (s0, s3, s3)\}$ is the same as $C_M$. Therefore, the decomposed properties can be used by introducing the specific time step.

Based on Lemma 4.6, the interaction fault $G(\neg p_1 \vee \neg p_2 \vee \ldots \vee \neg p_n)$ is converted into $G((clk \neq t_s) \vee \neg p_1 \vee \neg p_2 \vee \ldots \vee \neg p_n))$. The decomposed properties $G((clk \neq t_s) \vee \neg p_1)$, $G((clk \neq t_s) \vee \neg p_2)$, $\ldots$, $G((clk \neq t_s) \vee \neg p_n)$ are repeatedly applied to model checker until a common counterexample is found as described in Section 4.6. The counterexample is one of the interactions that satisfies the property $F((clk = t_s) \wedge p_1 \wedge p_2 \wedge \ldots \wedge p_n))$. In this decomposition scenario, the time step $(t_s)$ should be decided to guarantee that a counterexample exist within the given bound $(t_s)$. As described in the analysis of BMC techniques [Amla et al. 2003], deciding the bound is a challenging problem because the depth of counterexamples is unknown in most cases. Section 4.5 describes a

[2]The $clk$ variable is used to count time steps, and $t_s$ is a specific time step during model checking.

mechanism for deciding the bound ($t_s$) that enables test generation using SAT-based BMC.

For certain properties such as $pUq$, $F(p \rightarrow F(q))$, $F(p \rightarrow G(q))$, $G(p \rightarrow G(q))$, or $G(p \rightarrow F(q))$, decompositions are not beneficial compared to traditional model checking because it is very difficult to decide a specific time step between their decomposed properties. Although many property decompositions are not possible, it is important to note that the scenarios described in this section are sufficient to generate the test programs in the context of pipeline interactions.

An important consideration during property decomposition is how to specify and handle the different types of variables in the property. In general, the properties are described as pairs of module name and variable name. An interaction fault property $p_i$ can be either a local variable in a single module or a global variable over multiple modules. If $p_i$ is a local variable, it is converted into $(m_i.p_i)$ where $m_i$ is the corresponding module. If $p_i$ is a global variable, $p_i$ is decomposed into subproperties of corresponding modules. For example, for the property $G(\neg p_1 \vee \neg p_2)$, if $p_1$ is an interface variable between $m_1$ and $m_2$, and $p_2$ is a local variable of $m_2$, then the property is converted as $G(\neg m_1.p_1 \vee (\neg m_2.p_1 \vee \neg m_2.p_2))$. Decomposition of global variables is based on the decomposed modules of a processor model and their interfaces.

### 4.5 Determination of Bound for BMC

Determination of exact bound is a challenging problem, since the depth of counterexamples is unknown in general before applying the property. Choosing an incorrect bound increases test generation time and memory requirement. Use of a large bound loses the benefits of BMC due to search in a large number of irrelevant states. Therefore, the efficiency of test generation closely depends on the techniques of deciding the bound. We propose a method for determining the bound for each property based on the graph model of pipelined processors. The longest computation path in the pipeline corresponds to the counterexample bound to generate tests for all interaction scenarios. For example, in the graph model of the MIPS processor in Figure 3, the maximum bound is determined by the length of {FE→ DE→ IALU→ MEM→ DataMemory→ MainMemory → DataMemory→ MEM→ WB} if cache miss takes more time than any other pipeline paths. However, this bound is overconservative in most test scenarios because many interactions do not include this longest path. Therefore, using bound for each interaction is more efficient for test generation.

In the context of BMC, it is important to note that the relationship between diameter [Biere et al. 1999] and the exact bound for counterexample generation in our approach. The diameter [Biere et al. 1999] of a design is typically defined as the maximum distance between any two of its states $s_i$ and $s_j$, such that $s_j$ is reachable from $s_i$. The distance from $s_i$ to $s_j$ is the minimum number of time-steps to transition the design from $s_i$ to $s_j$. However, in our approach, we are only interested in finding a counterexample. Therefore, we are interested in the maximum distance between the initial state and the other reachable

states. Let us define this maximum distance as *init2anyDiameter*. Clearly, a bounded check of depth greater or equal to the *init2anyDiameter* of the design is guaranteed to generate a counterexample. Therefore, the exact bound $bound_i$ for each property $p_i$ should be smaller or equal to *init2anyDiameter*. In other words, the following relationship holds:

$$bound_i \leq init2anyDiameter \leq diameter. \tag{5}$$

Bound for each interaction is determined by the longest temporal distance from the root node to the nodes under consideration. For example, bound for the property *IALU, FADD2, and FADD3 in normal execution at the same time* will be 5 because FADD3 has the longest temporal distance from Fetch stage. If a property includes stall or exception activity, the temporal distance between the root node and leaf node (WB) is added to consider the causal node of the stall or exception. After deciding counterexample bound, either traditional model checking or SAT-based BMC can be used for generating counterexamples by taking design model, negated properties, and bound as inputs. It is important to note that the determination of bound is dependent on the initial state. A reset sequence is typically required to bring the design to the initial state. In the absence of such a reset sequence, it may not be possible to determine the bound for each property.

### 4.6 Test Generation Using Design and Property Decompositions

Algorithm 4 presents our decompositional model checking procedure (invoked from Algorithm 3) for design and property decompositions. The basic idea is to apply the decomposed properties (subproperties) to the corresponding design partition using model checking, and compose the generated partial counterexamples to construct the final test program. This algorithm accepts a property $P_i$ (already negated in Algorithm 3), a design $D$, and search bound $bound_i$ as inputs and produces the required test program. The design is decomposed based on the property decomposition and the techniques described in Section 4.3. Similarly, the property is decomposed based on the design decomposition and the techniques described in Section 4.4. The algorithm uses three lists to maintain the decomposed properties: *TaskList* for the present clock cycle *clk*, *NextList* for the next cycle, that is, $clk - 1$, and *AllList* for all properties. Each entry in the *TaskList* and the *NextList* contain a collection of subproperties that are applicable to corresponding design partitions. Therefore, each list can have up to $n$ entries where $n$ is the number of design partitions in the processor model. The tasks in the TaskList need to be performed in the current time step (*clk*). The tasks in the NextList will be performed in the next time step ($clk - 1$). AllList contains all the entries of TaskList for each time step. This information is used to resolve the conflict among subproperties, as described in Section 4.7. Initially these lists are empty.

The algorithm generates one test program for each property set $DP_i$ that consists of one or more subproperties based on their applicability to different modules or partitions in the design, as discussed in Section 4.2. The algorithm adds the subproperties to the *TaskList* and *AllList* based on the partitions

to which these properties are applicable. The algorithm iterates over all the subproperties in the *TaskList*. It removes an entry (say $k$-th location) from the *TaskList* which is the output requirement $outR_k$ of $k$-th partition. In general, this entry can be a list of subproperties (due to simultaneous output requirements from multiple children nodes) that need to be applied to partition $M_k$. These subproperties are composed to create the intermediate property $P_i^k$ using *MergeRequirements* described in Section 4.7. After negation of $P_i^k$, the property $\overline{P_i^k}$ is applied to the corresponding partition $M_k$ using the model checker to generate a counterexample.

The generated counterexample is analyzed to find the input requirements $inp_k$ for the partition $M_k$. If these are primary inputs (inputs of the root node in the graph model), then they are stored in *PrimaryInputs* list. Otherwise, for each parent node $M_r$ to which $inp_k$ is applicable, we extract the output requirements for $M_r$. This output requirement is added to the $r$-th entry of the *NextList* as well as the *AllList*. Finally, if the tasks for the current time step is completed (*TaskList* empty), *NextList* is copied to the *TaskList* and the time step *clk* is reduced by one. This process continues until both the lists are empty. Using a precise upper bound for the original property, $\overline{P_i}$ enables the *clk* to be zero and two lists empty at the same time. However, if one chooses the diameter [Biere et al. 1999] as the upper bound, two lists will be empty before the *clk* becomes zero. In both of these cases, it will ensure that we have obtained the primary input assignments for all the subproperties. These assignments are converted into a test program consisting of an instruction sequence.

For illustration, consider a simple property $P_1$ to verify a multiple stall scenario consisting of IALU (3rd module) and DIV (15th module) nodes in Figure 3 at clock cycle 5. We assume the module level partitioning of the design for this example. The property can be decomposed into two subproperties $P_1^3$ (IALU not stalled in cycle 5) and $P_1^{15}$ (DIV not stalled in cycle 5). This implies that *TaskList* will have two entries before entering the while loop: $TaskList[3] = P_1^3$ and $TaskList[15] = P_1^{15}$. At the first iteration of the while loop, $P_1^3$ will be applied to $M_3$ (IALU) using model checker; the generated counter example will be analyzed to find the output requirement for the Decode unit (2nd module in Figure 3) in clock cycle 4; and the requirement will be added to $NextList[2]$. During second iteration of the while loop, $P_1^{15}$ ($TaskList[15]$) will be applied to $M_{15}$ (DIV), the generated counter example will be analyzed to find the output requirement for the Decode unit in clock cycle 4, and the requirement, will be added to $NextList[2]$. At this point, the *TaskList* is empty and the *NextList* has only one entry with two requirements which is copied to the *TaskList*. At the third iteration of the while loop, these two requirements are composed into an intermediate property and applied to $M_2$ (Decode) that generates requirements for Fetch node. Finally, the fourth iteration applies the corresponding property to the Fetch unit that generates the primary input assignments. These assignments are converted to a test program. The following two examples show test generation using module level as well as pipeline path-level partitioning of the processor model.

**Algorithm 4**: *DecompositionalModelChecking*
**Inputs**: i) Property $P_i$, ii) Design $D$, and iii) $bound_i$
**Outputs**: Test program
Begin
  TaskList = $\phi$; NextList = $\phi$; AllList = $\phi$;
  PrimaryInputs = $\phi$; $clk = bound_i$
  $\{P_i^1, P_i^2, ..., P_i^m\}$ = DecomposeProperty($P_i$)
  $\{M_1, M_2, ..., M_n\}$ = DecomposeDesign($D$)
  **for** each design partition $M_j$ /* $P_i^j$ is applicable to $M_j$ */
    TaskList[$j$] = AllList[$clk$][$j$] = $P_i^j$
  **endfor**
  **while** TaskList is not empty and $clk > 0$
    $outR_k$ = RemoveEntry(TaskList[$k$])
    $\overline{P_i^k}$ = MergeRequirements($outR_k$, AllList, $clk$)
    $\overline{P_i^k}$ = Negate($P_i^k$)
    Counterexample = ModelChecking($\overline{P_i^k}$, $M_k$, $clk$)
    $inpR_k$ = input requirements for $M_k$ from Counterexample
    **if** $inpR_k$ are not primary_inputs
      **for** each applicable parent node $M_r$ of $M_k$
        $outR_r$ = Extract output requirements for $M_r$ from $inpR_k$
        NextList[$r$] = NextList[$r$] $\cup$ $outR_r$
        AllList[$clk$][$r$] = AllList[$clk$][$r$] $\cup$ $outR_r$
      **endfor**
    **else** PrimaryInputs = PrimaryInputs $\cup$ $inpR_k$
    **endif**
    **if** TaskList is empty
      $clk$=$clk - 1$;
      TaskList = NextList;
      NextList = $\phi$
    **endif**
  **endwhile**
  $test_i$ = ExtractInstructions(PrimaryInputs)
  **return** $test_i$
End

Example 3: Test Generation Using Module-Level Partitioning

Consider a multiple exception scenario at clock cycle 7 consisting of an overflow exception in IALU, divide by zero exception in DIV unit, and a memory exception in the MEM unit. The desired property $P$ is shown as:

```
P: F( (clk=7) & (MEM.exception = 1) & (IALU.exception = 1)
              & (DIV.exception = 1))
```

The negated property, $P'$ is:

```
P': G( (clk~=7) | (MEM.exception ~= 1) | (IALU.exception ~= 1)
                | (DIV.exception ~= 1))
```

$P'$ is decomposed into three subproperties:

```
P1: G((clk~=7) | (MEM.exception ~= 1))
P2: G((clk~=7) | (IALU.exception ~= 1))
P3: G((clk~=7) | (DIV.exception ~= 1))
```

The subproperties $P1$, $P2$, and $P3$ will be applied to MEM, IALU, and DIV modules using SMV model checker. The model checker will come up with a counterexample in each case as input requirements for the respective modules. For example, the counterexamples for $P1$, $P2$, and $P3$, respectively, are: ($C_{P1}$) *load* operation with memory address zero, ($C_{P2}$) *add* operation with the maximum value for both source operands, and ($C_{P3}$) *divide* operation with second source operand value zero. These requirements are converted into properties and applied to the respective parent modules. In this case, $P1'$ (from $C_{P1}$) is applied to IALU, and $P23'$ (combine $C_{P2}$ and $C_{P3}$)[3] is applied to the Decode unit in the next step. In each case, clock cycle value is reduced by one:

```
P1':  G((clk~=6) | (aluOp.opcode ~= LD) | (aluOp.src1Val ~= 0))
P23': G((clk~=6) | (decOp[0].opcode ~= ADD) | (decOp[0].src1Val ~= 2)
                 | (decOp[0].src2Val ~= 2) | (decOp[3].opcode =~ DIV)
                 | (decOp[3].src2Val ~= 0))
```

The outcome of the property $P1'$ will be applied to Decode unit (generates $P1''$ say) whereas the outcome of the $P23'$ will be applied to Fetch unit (generates primary inputs $PI_i$) in time step 5. In time step 4, $P1''$ will be applied to Fetch unit that generates the primary inputs $PI_j$. The primary inputs $PI_i$ and $PI_j$ are combined based on their time step (clock cycle) to generate the final test program:

```
Fetch Instructions ([0] for ALU... [3] for DIV)
Cycle  [0]             [1]  [2]  [3] //R0 is 0
 1     ADDI R2 R0 #2   NOP  NOP  NOP //R2 = 2
 2     NOP             NOP  NOP  NOP
 3     NOP             NOP  NOP  NOP
 4     LD   R1 0(R0)   NOP  NOP  NOP
 5     ADD  R3 R2 R2   NOP  NOP  DIV R3 R0 R0
```

[3]Note that when multiple children create requirements for the parent (e.g., $P23'$), conflicts can occur. In such cases, alternative assignments need to be evaluated for the conflicting variable, as described in Section 4.7.

Example 4: Test Generation Using Path-Level Partitioning

The example shown above assumes a module-level partitioning of the processor model. However, it is not always possible to decompose a property based on module-level partitioning. For example, if we are trying to determine whether two feedback (data-forwarding) paths shown in Figure 3 are activated at the same time, it is not possible to decompose this property at module level because the implication relation between *feedOut* and *feedIn* (in the following property) will be lost.

```
/* Original Property */
 P: F((clk=9) & (FADD4.feedOut -> X(FADD1.feedIn))
              & (MUL7.feedOut -> X(MUL1.feedIn)))

/* Property after Negation*/
 P': G(((clk~=9 | ~FADD4.feedOut) | (clk~=10 | ~FADD1.feedIn)) |
        ((clk~=9 | ~MUL7.feedOut) | (clk~=10 | ~MUL1.feedIn)))

/* Properties after Decomposition*/
 P1: G((clk~=9 | ~FADD4.feedOut) | (clk~=10 | ~FADD1.feedIn))
 P2: G((clk~=9 | ~MUL7.feedOut) | (clk~=10 | ~MUL1.feedIn))
```

To enable property decomposition in this example, we need to partition the design differently. The floating-point adder path (FADD1 to FADD4) should be treated as a design partition $F_{path}$. Similarly, the multiplier path (MUL1 to MUL7) should be treated as another partition $M_{path}$. This new partitioning is applied for test generation. First, $P1$ and $P2$ can be applied on $F_{path}$ and $M_{path}$, respectively, that generates counterexamples $C1$ and $C2$. Next, $C1$ and $C2$ are combined and the corresponding property is applied to the Decode unit to generate the counterexample $C3$. Next, the property corresponding to $C3$ is applied to the Fetch unit that generates the primary input requirements. Finally, these primary input requirements are converted into the required test program.

### 4.7 Merging Partial Counterexamples

The output requirement ($outR_r$ in Algorithm 4) generated from a single child node can be directly used for the corresponding module ($M_r$) simply by negating the output requirement. In case of multiple children, the input requirements generated from children nodes need to be merged appropriately into the required property for the parent node. However, this is nontrivial since the input requirements can be conflicting due to the fact that the model checker assigns arbitrary values to the variables that do not have influence on falsification of the children nodes. For example, in Figure 4, four reservation station ($RS$) modules share the parent module Issue. Counterexamples (input requirements of each $RS$) generated from four $RS$s at the time step $clk = t_s + 1$ should be combined for creating the required property of Issue module at $clk = t_s$. However, they may require different output values for the same variable of the module Issue.

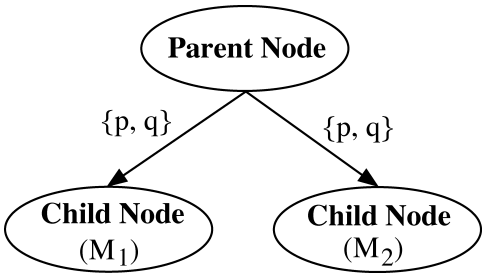


Fig. 6. An example structure with two children nodes.

In case of output requirement conflict, we adjust input requirements of the children nodes by excluding the current input requirement, called false requirement. For example, consider the scenario shown in Figure 6. Assume that the output variables of the parent node are $p$ and $q$, the input requirement of one child is $(p = 1 \wedge q = 0)$ that is generated by $G((clk \neq (t_s + 1)) \vee \neg(M_1.p = 1))$ at child node $M_1$, and the input requirement of the other child is $(p = 0 \wedge q = 1)$ that is generated by $G((clk \neq (t_s + 1)) \vee \neg(M_2.q = 1))$ at child node $M_2$. Obviously, there is no way to assign output $p$ and $q$ to satisfy these two conflicting inputs. We refine the subproperties of children nodes to resolve the conflict requirements by excluding the false requirement. The desired subproperties stored in $AllList[t_s + 1]$ for children nodes, and they are modified by adding the negated version of the conflict requirement:

$$F((clk = (t_s + 1)) \wedge (M_1.p = 1) \wedge \neg(M_1.p = 1 \wedge M_1.q = 0))$$
$$F((clk = (t_s + 1)) \wedge (M_2.q = 1) \wedge \neg(M_2.p = 0 \wedge M_2.q = 1)).$$

To generate the input requirements of the $module1$, the above properties are negated:

$$G((clk \neq (t_s + 1)) \vee \neg(M_1.p = 1) \vee (M_1.p = 1 \wedge M_1.q = 0))$$
$$G((clk \neq (t_s + 1)) \vee \neg(M_2.q = 1) \vee (M_2.p = 0 \wedge M_2.q = 1)).$$

These subproperties do not allow the counterexample $(p = 1 \wedge q = 0)$ any more. The generated counterexample will be $(p = 1 \wedge q = 1)$ as the input requirements of $module1$ and $module2$. As a result, we can merge them into the output requirement of the parent node as $(p = 1 \wedge q = 1)$ at $clk = t_s$. If there is an interface variable $r$ between the parent and its child $module2$, it does not cause the output requirement conflict of the parent node, since the input requirement of $module1$ does not influence the variable $r$. If there is another child node $module3$ that has the interface variables $p$ and $r$, we need to adjust three input requirements of $module1$, $module2$, and $module3$ to resolve any conflict among them. It is possible that there is no common variable assignments for shared input variables among children nodes, since their output requirements may be generated from false input requirements from the subsequent stages (child nodes of $M_1$ and $M_2$). In this case, we need to refine the subproperties of grandchildren nodes stored in $AllList[t_s + 2]$. The procedure of subproperty refinement continues until the conflict is resolved or $clk$ is equal to $bound_i$, that is, upper bound to search for a test program.

Although, this procedure of iterative conflict resolution may continue forever in the worst case, our experimental studies show that number of such iterations is small: 10 to 15, on average. Our observation is that in the majority of these cases the conflict happened in the fork node (e.g., Issue unit in e500 or Decode unit in MIPS) and it could be avoided if we maintain "don't cares". In other words, when we compose the partial counterexamples in each stage, we can carry forward "don't care" values to the parent stages. To enable this, we need to enable/modify the existing model-checking techniques to produce don't care values for the variables where exact assignment is not required to produce the counterexample. We have also implemented techniques for learning from previous mistakes, for example, conflict in the previous stage for the same property or from a similar property. For example, the test 9 of e500 processor (in Table IV) uses the knowledge of test 8. In other words, for test 9 instead of searching for a counterexample from the initial state, it started from the failure state of test 8. To safeguard against the scenarios where the iteration may continue indefinitely (the worst-case scenario), we use a threshold to terminate the repeated conflict generation for the same scenario.

## 5. EXPERIMENTS

We applied our test generation methodology on a multi-issue MIPS architecture [Hennessy and Patterson 2003] and a superscalar commercial e500 processor [e500 Manual 2005]. We performed various test generation experiments for validating the pipeline interactions by varying different design partitions and property decompositions. In this section, we present experimental results in terms of time and memory requirement in test generation.

### 5.1 Test Generation Using Model Checking and Module-Level Decomposition

Our test generation technique using UMC and module-level decomposition is applied on a multi-issue MIPS architecture, as shown in Figure 3. We have used NuSMV [NuSMV] for running incremental BMC experiments. We have used Cadence SMV [SMV] model checker to perform all other experiments. We made few simplifications in the MIPS processor model to compare with existing approaches. For example, if thirty-two 32-bit registers are used in the register file, the UMC approach can not produce any counterexample even for a simple property with no pipeline interaction due to the memory depletion during model checking. For comparison, we used eight 2-bit registers for the following experiments to ensure that the existing approaches can generate counterexamples. All the experiments were run on a 1GHz Sun UltraSparc with 8G RAM.

Table II presents the results of the comparison of test generation techniques for MIPS processor. The first column defines the type of properties used for test generation based on number of module interactions. For example, "None" implies properties applicable to only one module; "Two Modules" implies properties that include two module interactions, and so on. Each row presents the average memory requirement (M: MB, K:KB) for the BDD nodes (or CNF clauses) used as well as test generation time (in seconds). For example, the first row presents the average time and memory requirement for 68 ($n = 17$, $r = 4$,

Table II. Comparison of Test Generation Techniques for MIPS Processor

| Module Interactions | Cadence SMV (UMC) | | Cadence SMV (Cex-based Abstraction) | | | NuSMV (Incr. BMC) | | Our Approach (UMC+Dec.) | |
|---|---|---|---|---|---|---|---|---|---|
| | BDD | Time | BDD | Clauses | Time | Clauses | Time | BDD | Time |
| None | 6 M | 165 | 23K | 33K | 0.37 | 135K | 8.47 | 3K | 0.06 |
| 2 Modules | 11M | 215 | 54K | 138K | 5.89 | 201K | 9.99 | 6K | 0.12 |
| 3 Modules | 21M | 240 | 76K | 139K | 7.67 | 268K | 11.57 | 9K | 0.19 |
| 4 Modules | | >1hr | | | >1hr | 334K | 14.08 | 11K | 0.28 |
| 5 Modules | | >1hr | | | >1hr | 401K | 16.65 | 15K | 0.35 |
| 6 Modules | | >1hr | | | >1hr | 467K | 18.76 | 21K | 0.51 |

"Time" is in seconds, and memory requirement is in BDD nodes (UMC) or CNF clauses (BMC).

and i = 1 in Equation (2)) single module properties. Similarly, the second row shows the average values for test generation of 2,186 two module interactions, and so on. The second and third columns present the average memory (number of BDD nodes) and time (in seconds) requirement for doing UMC using Cadence SMV. The next three columns present the average memory (number of BDD nodes as well as number of CNF clauses) and time requirement for doing counterexample-based abstraction [Amla and McMillan 2004] using Cadence SMV. The seventh and eighth columns present the average memory and time requirement for doing incremental BMC using NuSMV. The next two columns presents the results using our approach—UMC with design and property decompositions. Clearly, our approach requires several orders of magnitude less memory and test generation time compared to existing UMC-based approaches. Our approach is also beneficial compared to existing BMC-based approaches by providing at least an order-of-magnitude reduction in both test generation time and memory requirement.

### 5.2 Test Generation Using SAT-based BMC and Cluster-Level Decomposition

We applied our test generation technique using SAT-based BMC and cluster-level decomposition on a MIPS architecture. We used Cadence SMV as model checker and zChaff [Moskewicz et al. 2001] as SAT solver. Section 5.1 (Table II) presented results of our approach using design and property decompositions in the context of UMC. This section presents results of our approach using design and property decompositions in the context of both UMC and BMC. All the experimental setup and properties remain the same, as in Section 5.1.

Table III compares our approach with the existing UMC and BMC-based approaches. The first and second columns in Table III are identical to the first and third columns in Table II. The third column presents the test generation time using BMC of Cadence SMV. In this case we used the maximum bound of 45 to ensure we get all the counterexamples. The next three columns present the test generation times using our approach with module-level decomposition in three ways: UMC, BMC with maximum bound, and BMC using bound for each property. The next three columns are similar to these three columns except that the final three columns consider cluster-level decompositions. The maximum bound of 45 was used assuming that the longest length is taken by memory operations, that is, the summation of the IALU pipeline path length (5)

Table III. Comparison of Test Generation Techniques for MIPS Processor

| Module Interactions | SMV (UMC) | SMV (BMC) | Our Approach using Decompositions | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | Module-level Decomp. | | | Cluster-level Decomp. | | |
| | | | | BMC | | | BMC | |
| | | | UMC | Max. $k$ | Each $k$ | UMC | Max. $k$ | Each $k$ |
| None | 165 | 5.63 | 0.06 | 2.24 | 0.42 | 0.21 | 3.87 | 0.22 |
| 2 Modules | 215 | 7.42 | 0.12 | 6.41 | 1.38 | 1.81 | 4.31 | 0.43 |
| 3 Modules | 240 | 7.74 | 0.19 | 6.75 | 1.45 | 8.06 | 5.72 | 0.52 |
| 4 Modules | >1hr | 8.79 | 0.28 | 7.63 | 1.97 | 37.13 | 6.98 | 0.64 |
| 5 Modules | >1hr | 9.29 | 0.35 | 9.03 | 2.18 | 83.25 | 8.31 | 0.62 |
| 6 Modules | >1hr | 9.58 | 0.51 | 10.70 | 2.50 | 126.01 | 9.04 | 0.68 |

and data-transfer path length (40). As expected, Table III shows that the test generation time grows with the increase of the number of module interactions. Bound for each property reduces approximately 90% of the test generation time compared to using BMC with maximum bound. An interesting observation is that UMC with module-level decomposition provides better performance than SAT-based BMC. This is because unfolding the processor model and converting it to the SAT problem takes some time. In other words, UMC might be better when properties are simple and can be decomposed at the module level. As expected, if the bound is small, BMC without decomposition (existing approach) is feasible but will take an order-of-magnitude more test generation time compared to our approach with individual bound for each property (BMC with decompositions).

### 5.3 Test Generation for e500 Processor

We also applied our decompositional model-checking technique on a superscalar e500 processor. We described the processor using SMV language. Our processor model includes microarchitectural structure and clock-accurate behaviors of the processor. We represented one clock cycle as two time steps (low and high at each cycle) so that the processor model accommodates the behaviors of read and write at the same cycle in the first-in-first-out queues and reservation stations. We performed various test generation experiments for validating the pipeline interactions and corner cases. In this section, we present a subset of the test sequences generated by our test generation framework. Next, we describe how the generated test programs are used in processor-validation framework.

5.3.1 *Results.* Table IV shows a subset of the directed test cases that we have generated for e500 processor using our approach. The first column indicates the testcase ID. The second column briefly describes the testcase. The third column indicates the number of iterations (UMC/BMC runs) required to produce the final counterexample. The fourth column indicates the number of instructions in the final test program. The next column presents the test generation time (in seconds) and indicates the total time required to obtain the counterexample. The total time includes the time for performing all the UMC/BMC runs for obtaining and merging partial counterexamples. In majority of the test generation scenarios, the number of UMC/BMC runs (iterations) due to conflicting assignments is small (10 to 15). The worst-case scenario was for test

Table IV. Various Test Cases Generated by Our Framework for e500 Processor

| ID | Test Cases | Iterations (# runs) | Length (#inst.) | Time (sec) |
|---|---|---|---|---|
| 1 | Instruction dual issue | 52 | 15 | 30 |
| 2 | Renaming *src1* operand | 42 | 12 | 25 |
| 3 | Read operand from forwarding path (RAW) | 35 | 9 | 20 |
| 4 | Reservation station reads operand from forwarding path | 28 | 7 | 15 |
| 5 | Read operand from renaming reg. (RAW) | 37 | 10 | 20 |
| 6 | Read operand from GPR (RAW) | 40 | 11 | 25 |
| 7 | Renaming for WAW (no stall) | 34 | 8 | 20 |
| 8 | Stall at Decode stage due to IQ full | 59 | 14 | 35 |
| 9 | Stall at Decode stage due to CQ full, then released queue full at the next clock cycle | 112 | 34 | 61 |
| 10 | CQ full, then full again | 130 | 35 | 70 |
| 11 | CQ full, then empty, and then full again | 527 | 95 | 290 |
| 12 | Retire only one instruction in Completion | 48 | 12 | 28 |
| 13 | "lwz" instruction at LSU_stage3 | 25 | 7 | 15 |
| 14 | "add" at Fetch2 & "mulhw" at MU_stage2 simultaneously | 30 | 6 | 18 |
| 15 | "addi" at Completion, "mulhw" at MU_stage1, & "lwz" at LSU_stage1 at the same clock | 44 | 12 | 25 |
| 16 | "mulhw" at Completion, "add" & "addi" waits in completion queue, & "lzw" at LSU_stage3 | 72 | 12 | 40 |
| 17 | "lwz" and "add" at Completion, "mulhw" at MU_stage3, "addi" at CQ, "lwz" at LSU_stage1 | 61 | 14 | 35 |
| 18 | "mulhw" & "add" retire, "mulhw" at MU_stage4, "addi" at CQ, & "lwz" at LSU-stage2 | 80 | 15 | 45 |

11 where a total of 527 UMC runs were required and most of the runs (351 runs, 67%) were in the iterations between Issue and RS modules. As discussed in Section 4.7, we have employed several techniques to reduce these iterations including reuse of the knowledge of a previously generated test. For example, when generating the counterexample for test 9, we use the knowledge of test 8. In other words, test 9 would have taken 98 seconds (instead of 61 seconds) if the test 8 knowledge was not used.

The test program for Case 11 validates the feature of completion queue (CQ) by piling data up and down in the first-in-first-out queue. Test programs for Case 3 through 6 exercise operand read from four different resources, as shown in Figure 7, which can be generated at microarchitecture level but very difficult at ISA level. In terms of efficiency, only several seconds were spent on test generation except for Case 11 where test generation took a few minutes. The test Cases 13 through 18 show various interaction scenarios. For example, test Case 13 only activates one node whereas test Case 15 considers three node interactions at the same clock cycle.

5.3.2 *Microarchitectural Validation using Test Programs.* Micro-architectural design errors, such as performance bugs, are hard to be exposed by architectural test generation. Furthermore, they may not be detected by ISA functional simulation. For example, test generation for uncovering incorrect stalls in pipeline stages require timing information of instruction flow and those bugs are only visible during the clock-accurate

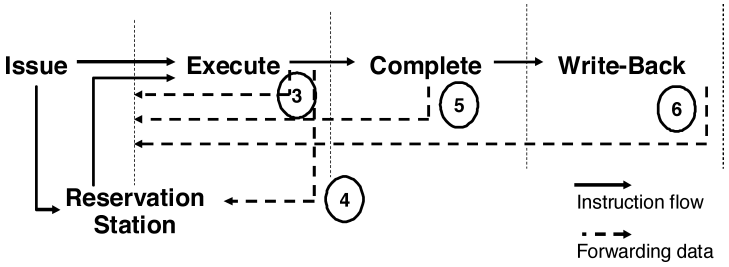


Fig. 7. Four different data-forwarding mechanisms.

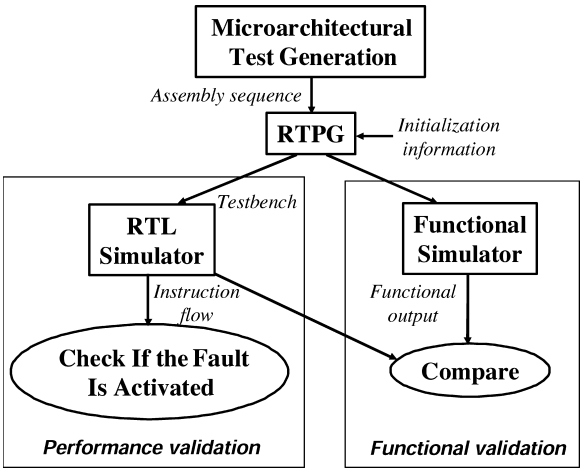


Fig. 8. Microarchitectural validation flow.

simulation. Therefore, microarchitectural validation plays an important role in ensuring the correctness of performance as well as functionality of the processor designs.

We have performed microarchitectural validation by using the existing methodology in an industrial settings that includes an internal random test pattern generator (RTPG) tool. Figure 8 shows the validation flow. We converted the assembly test sequences generated by our method into the input format of the RTPG tool that produces testbenches for RTL simulation. The simulator shows how instructions go through the pipeline stages on a cycle-by-cycle basis as well as whether the stored results in register files and memory are correct. Capturing when and which instructions move from one stage to the next ensures that the generated tests exercise the target microarchitectural artifacts. We compared the validation effort for activating these microarchitectural features using the existing validation methodology in an industrial setting and our approach. On an average, each of our test case took less than 100 clock cycles whereas the existing random/pseudo-random tests took approximately 100,000 clock cycles to activate the target fault. As a result, our approach reduced the overall validation effort by several orders of magnitude.

### 5.4 Discussion: Applicability and Limitations

This article studied the design and property decompositions in the context of directed test generation for pipelined processors. As a result, it presented various domain-specific (directed test generation) and application-specific (pipelined processors) heuristics and optimizations. Clearly, our approach is applicable only for test (counterexample) generation purposes. In other words, its domain is limited to automated generation of directed tests. However, the application is not limited to only pipelined processors. The approaches presented in this article can be used for directed test generation for any software/hardware designs where the interaction between the components are limited. In other words, the modeling, decomposition and test generation techniques are applicable to any design that is decomposition friendly in the context of counterexample (test) generation. For example, in a SoC design if each component interacts heavily with all the other components in the design, the design and property decomposition may not be feasible. Similarly, in software-based systems if too many global variables are used for interaction between modules and/or there is strong coupling between components, decomposition will not be feasible. In practice, the interaction between components (modules) are well defined and limited in nature due to design-for-verification and other requirements. As a result, a vast majority of designs (hardware, software, or hardware+software) are decomposition friendly and our approach is applicable for directed test generation for those designs.

## 6. CONCLUSIONS

Functional verification is widely acknowledged as a major bottleneck in microprocessor design methodology. Compared to the random or constrained-random tests, the directed tests can reduce overall validation effort, since shorter tests can obtain the same coverage goal. However, there is a lack of automated techniques for directed test generation. This article presented a directed test generation technique for validation of performance as well as functionality of the modern microprocessors. Our methodology is based on decompositional model checking where the processor model as well as the properties are decomposed and the model checking is applied on smaller partitions of the design using decomposed properties. We introduced the notion of time steps to enable decomposition of the properties into smaller ones based on their clock cycles. We have developed an efficient algorithm to merge the partial counterexamples generated by the decomposed properties to create the final test program corresponding to the original property. Our experimental results using MIPS and e500 processor architectures demonstrate the efficiency of our method by generating complicated microarchitectural tests. Since the proposed technique is generic, its framework can be used for validation of other industrial-strength processors. Furthermore, this work can be an excellent complement to the current RTPG validation methodology without modification of the existing validation flow.

Our future work includes test generation for safety as well as liveness properties for multiprocessor SoC designs. We also plan to develop efficient test

compaction techniques to reduce the number of functional test programs for validation of multiprocessor SoC architectures.

ACKNOWLEDGMENT

We would like to acknowledge the contributions of Dr. Jayanta Bhadra and Dr. Magdy Abadir for giving us the opportunity to apply our technique on e500 processor architecture.

REFERENCES

ADIR, A., ALMOG, E., FOURNIER, L., MARCUS, E., RIMON, M., VINOV, M., AND ZIV, A. 2004. Genesyspro: Innovations in test program generation for functional processor verification. *IEEE Des. Test Comput. 21*, 2, 84–93.

ADIR, A., BIN, E., PELED, O., AND ZIV, A. 2003. Piparazzi: A test program generator for micro-architecture flow verification. In *Proceedings of High-Level Design Validation and Test Workshop (HLDVT)*. IEEE, Los Alamitos, CA, 23–28.

AHARON, A., GOODMAN, D., LEVINGER, M., LICHTENSTEIN, Y., MALKA, Y., METZGER, C., MOLCHO, M., AND SHUREK, G. 1995. Test program generation for functional verification of PowerPC processors in IBM. In *Proceedings of Design Automation Conference (DAC)*. ACM, New York, 279–285.

AMLA, N., KURSHAN, R., MCMILLAN, K., AND MEDEL, R. 2003. Experimental analysis of different techniques for bounded model checking. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, Berlin, 34–48.

AMLA, N. AND MCMILLAN, K. 2004. A hybrid of counterexample-based and proof-based abstraction. In *Proceedings of the 5th International Formal Methods in Computer-Aided Design (FMCAD)*. Springer, Berlin, 260–274.

AMLA, N., DU, X., KUEHLMANN, A., KURSHAN, R., AND MCMILLAN, K. 2005. An analysis of SAT-based model checking techniques in an industrial environment. In *Proceedings of Correct Hardware Design and Verification Methods (CHARME)*. Springer, Berlin, 254–268.

BIERE, A., CIMATTI, A., AND CLARKE, E. M. 2003. Bounded model checking. *Adv. Comput. 58*.

BIERE, A., CIMATTI, A., CLARKE, E., AND ZHU, Y. 1999. Symbolic model checking without BDDs. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, Berlin, 193–207.

BJESSE, P. AND KUKULA, J. 2004. Using counter example guided abstraction refinement to find complex bugs. In *Proceedings of Design Automation and Test in Europe (DATE)*. IEEE, Los Alamitos, CA, 156–161.

BRYANT, R. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comput. C-35*, 8, 677–691.

CAMPENHOUT, D., MUDGE, T., AND HAYES, J. 1999. High-level test generation for design verification of pipelined micro-processors. In *Proceedings of the Design Automation Conference (DAC)*. ACM, New York, 185–188.

CADENCE SMV. http://www-cad.eecs.berkeley.edu/~kenmcmil/smv.

CLARKE, E., GRUMBERG, O., AND PELED, D. 1999. *Model Checking*. MIT Press, Cambridge, MA.

CLARKE, E., GRUMBERG, O., MCMILLAN, K., AND ZHAO, X. 1995. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Proceedings of the Design Automation Conference (DAC)*. ACM, New York, 427–432.

COPTY, F., FIX, L., FRAER, R., GIUNCHIGLIA, G., KAMHI, G., TACCHELLA, A., AND VARDI, M. 2001. Benefits of bounded model checking at an industrial setting. In *Proceedings of Computer-Aided Verification (CAV)*. Springer, Berlin, 436–453.

FINE, S. AND ZIV, A. 2003. Coverage directed test generation for functional verification using Bayesian networks. In *Proceedings of the Design Automation Conference (DAC)*. ACM, New York, 286–291.

FREESCALE. PowerPCTM e500 Core Family Reference Manual. http://www.freescale.com/files/32bit/doc/ref manual/e500CORERM.pdf 2005.

GARGANTINI, A. AND HEITMEYER, C. 1999. Using model checking to generate tests from requirements specifications. *ACM SIGSOFT Software Engin. Notes. 24*, 146–162.

GOLDBERG, E. AND NOVIKOV, Y. 2002. BerkMin: A fast and robust SAT-solver. In *Proceedings of Design Automation and Test in Europe (DATE)*. IEEE, Los Alamitos, CA, 142–149.

GLUSKA, A. 2006. Practical methods in coverage-oriented verification of the Merom microprocessor. In *Proceedings of the Design Automation Conference (DAC)*. ACM, New York, 332–337.

HALAMBI, A., GRUN, P., GANESH, V., KHARE, A., DUTT, N., AND NICOLAU, A. 1999. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proceedings of Design Automation and Test in Europe (DATE)*. IEEE, Los Alamitos, CA, 485–490.

HO, P., ISLES, A., KAM, T. 1998. Formal verification of pipeline control using controlled token nets and abstract interpretation. In *Proceedings of International Conference on Computer-Aided Design (ICCAD)*. ACM, New York, 529–536.

HO, R., YANG, C., HOROWITZ, M. A., DILL, D. 1995. Architecture validation for processors. In *Proceedings of International Symposium on Computer Architecture (ISCA)*. ACM, New York.

IWASHITA, H., KOWATARI, S., NAKATA, T., HIROSE, F. 1994. Automatic test program generation for pipelined processors. In *Proceedings of International Conference on Computer-Aided Design (ICCAD)*. ACM, New York, 580–583.

JACOBI, C. 2002. Formal verification of complex out-of-order pipelines by combining model-checking and theorem-proving. In *Proceedings of Computer-Aided Verification (CAV)*. Springer-Verlag, Berlin, 309–323.

JHALA, R. AND MCMILLAN, K. L. 2001. Micro-architecture verification by compositional model-checking. In *Proceedings of Computer-Aided Verification (CAV)*. Springer-Verlag, Berlin, 396–410.

JIN, H. AND F. SOMENZI, F. 2005. An incremental algorithm to check satisfiability for bounded model-checking. In *Proceedings of the International Workshop on Bounded Model-Checking (BMC'04)*. Elsevier, 51–65.

KOO, H. AND MISHRA, P. 2006a. Functional test generation using property decompositions for validation of pipelined processors. In *Proceedings of Design Automation and Test in Europe (DATE)*. IEEE, Los Alamitos, CA.

KOO, H. AND MISHRA, P. 2006b. Test generation using SAT-based bounded model-checking for validation of pipelined processors. In *Proceedings of the ACM Great Lakes Symposium on VLSI (GLSVLSI)*. ACM, New York.

KOO H., MISHRA, P., BHADRA, J., AND ABADIR, M. 2006. Directed micro-architectural test generation for an industrial processor: A case study. In *Proceedings of Micro-processor Test and Verification (MTV)*. IEEE, Los Alamitos, CA.

HENNESSY, J. AND PATTERSON, D. 2003. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, St. Louis, MO.

KOHNO, K. AND MATSUMOTO, N. 2001. A new verification methodology for complex pipeline behavior. In *Proceedings of the Design Automation Conference (DAC)*. ACM, New York, 816–821.

MARQUES-SILVA, J. AND SAKALLAH, K. 1999. Grasp: A search algorithm for propositional satisfiability. *IEEE Trans. Comput. 48*, 5, 506–521.

MATHAIKUTTY, D., KODAKARA, S., DINGANKAR, A., SHUKLA, S., AND LILJA, D. 2007. Design fault directed test generation for micro-processor validation. In *Proceedings of Design Automation and Test in Europe (DATE)*. IEEE, Los Alamitos, CA.

MATHAIKUTTY, D., AHUJA, S., DINGANKAR, A., AND SHUKLA, S. 2007. Model-driven test generation for system level validation. In *Proceedings of High-Level Design Validation and Test (HLDVT)*. IEEE, Los Alamitos, CA.

MISHRA, P. AND DUTT, N. 2002. Automatic functional test program generation for pipelined processors using model-checking. In *Proceedings of High-Level Design Validation and Test (HLDVT)*. IEEE, Los Alamitos, CA, 99–103.

MISHRA, P. AND DUTT, N. 2004. Graph-based functional test program generation for pipelined processors. In *Proceedings of Design Automation and Test in Europe*. IEEE, Los Alamitos, CA, 182–187.

MISHRA, P. AND DUTT, N. 2005. Functional coverage driven test generation for validation of pipelined processors. In *Proceedings of Design Automation and Test in Europe*. IEEE, Los Alamitos, CA, 678–683.

MISHRA, P. AND DUTT, N. (EDS.). 2008. In *Processor Description Languages*. Morgan Kaufmann, St. Louis, MO.

MOSKEWICZ, M., MADIGAN, C., ZHAO, Y., ZHANG, L., AND MALIK, S. 2001. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference (DAC)*. ACM, New York, 530–535.

NUSMV. http://nusmv.irst.itc.it/.

PARTHASARATHY, G., IYER, M., CHENG, K. T., AND WANG, L. 2004. Safety property verification using sequential SAT and bounded model-checking. *IEEE Des. Test Comput. 21*, 2, 132–143.

PATEL, H. AND SHUKLA, S. 2007. Model-driven validation of SystemC designs micro-processor. In *Proceedings of the Design Automation Conference (DAC)*. ACM, New York, 29–34.

PRASAD, M., BIERE, A., AND GUPTA, A. 2005. A survey of recent advances in SAT-based formal verification. *Int. J. Softw. Tools Tech. Trans. 7*, 2, 156–173.

PROPERTY SPECIFICATION LANGUAGE. http://vhdl.org/ieee-1850/.

SHEN, J. AND ABRAHAM, J. 2000. An RTL abstraction technique for processor micro-architecture validation and test generation. *J. Electron. Test. 16*, 1-2, 67–81.

STRICHMAN, O. 2001. Pruning techniques for the SAT-based bounded model-checking problem. In *Proceedings of Correct Hardware Design and Verification Methods (CHARME)*. Springer-Verlag, Berlin, 58–70.

TUERK, T., SCHNEIDER, K., AND GORDON, M. 2007. Model-checking PSL using HOL and SMV. In *Proceedings of the International Haifa Verification Conference (HVC 2006)*. Springer, Berlin, 1–15.

UR, S. AND YADIN, Y. 1999. Micro-architecture coverage directed generation of test programs. In *Proceedings of the Design Automation Conference (DAC)*. ACM, New York, 175–180.

UTAMAPHETHAI, N., BLANTON, R. D. S., AND SHEN, J. P. 2000. Effectiveness of micro-architecture test program generation. *IEEE Des. Test 17*, 4, 38–49.

WAGNER, I., BERTACCO, V., AND AUSTIN, T. 2005. Stresstest: An automatic approach to test generation via activity monitors. In *Proceedings of the Design Automation Conference (DAC)*. ACM, New York, 783–788.

WHITTEMORE, J., KIM, J., AND SAKALLAH, K. 2001. SATIRE: A new incremental satisfiability engine. In *Proceedings of the Design Automation Conference (DAC)*. ACM, New York, 542–545.

Received May 2008; revised November 2008; accepted December 2008