Abstract

Functional validation is one of the major bottlenecks in processor design methodology due to combined effects of increasing complexity and decreasing time-to-market. Increasing complexity of designs leads to larger set of design errors. Shorter time-to-market requires a faster validation scheme. Simulation using functional test vectors is the most widely used form of processor validation. While existing model checking based approaches have proposed several promising ideas for efficient test generation, many challenges remain in applying them to realistic pipelined processors. The time and resources required for test generation using existing model checking based techniques can be extremely large. This report presents an efficient test generation technique using SAT-based bounded model checking. To demonstrate the usefulness of this approach, we have applied this technique to generate test programs for validation of the VLIW MIPS processor.
Contents

1 Introduction ........................................................................................................................................... 3
2 Related Work ........................................................................................................................................ 3
3 Background ......................................................................................................................................... 4
   3.1 Model Checking ........................................................................................................................... 4
   3.2 SAT Solving ................................................................................................................................. 5
   3.3 SAT-based Bounded Model Checking ........................................................................................ 5
4 Functional Test Generation ............................................................................................................... 6
   4.1 Test Program Generation using Model Checking ..................................................................... 6
   4.2 Test Program Generation using SAT-based BMC ................................................................. 7
5 A Case Study ....................................................................................................................................... 8
   5.1 Experimental Setup .................................................................................................................... 8
   5.2 Test Generation: An Example .................................................................................................... 9
   5.3 Results ......................................................................................................................................... 10
6 Conclusion ......................................................................................................................................... 10
7 References .......................................................................................................................................... 11

List of Figures

Figure 4.1: Test generation using model checker ................................................................................. 6
Figure 4.2: SAT-base BMC test generation ......................................................................................... 7
Figure 5.1: MIPS processor architecture .......................................................................................... 8

List of Tables

Table 5.1: A generated test program: An example ............................................................................. 10
Table 5.2: Test generation time using SAT-based BMC ..................................................................... 10
1 Introduction

The complexity of microprocessors increases at exponential rate by Moore’s law\(^1\). Verification complexity is linearly proportional to design complexity [5]. Due to the exponential increase of verification complexity, verification of modern processors is widely acknowledged as a major bottleneck in design methodology. Some researchers [17] proposed that functional verification task consumes over 70% of the design efforts.

Functional validation of modern processors is generally performed by using random, directed or combined test programs based on simulation techniques. Directed test generation is more beneficial to reducing the validation time and overall effort since less number of tests is required than random tests to obtain the same coverage goal. One of the promising approaches in directed test generation is specification-driven test generation using model checking [26] as it generates test programs automatically without knowledge of implementation. Various properties (desired behaviors) are generated from specification and they are applied to a model checker along with the specification such that test programs are produced automatically for validation of an implementation\(^2\). However, this approach is not suitable for today’s complex pipelined processors since the time and memory requirements can be prohibitively large in many test generation scenarios due to the state explosion problem in model checking.

This report presents a new functional test generation technique that uses SAT (Satisfiability) based Bounded Model Checking (BMC) to reduce the time and memory required for test program generation. Modern Boolean SAT solvers combined with BMC have been successfully used in finding counterexamples of temporal properties. SAT-based BMC reduces search space for counterexamples by imposing a bounded length and then uses a SAT solver to generate a counterexample. Reduction of search space results in reduction of time and memory requirement for test generation. We applied the proposed technique to the multi-issue MIPS processor. Experimental results show that the required time and memory is several orders of magnitude less than the existing approach using model checking.

The rest of the report is organized as follows. Section 2 presents related work on test program generation and SAT-based BMC in the context of functional validation of pipelined processors. Section 3 presents basic verification techniques related to our approach. Section 4 describes the proposed test generation methodology followed by a case study in Section 5. Finally, Section 6 concludes the paper and discusses future work.

2 Related Work

For efficient test generation of IBM processors, Genesys-Pro [1] combines architecture and testing knowledge. Aharon et al. [2] used a formal model of processor architecture for directed test program generation. Fine and Ziv [29] have proposed coverage directed test generation based on Bayesian networks. Ur and Yadin [31] have

---

\(^1\) In 1965, the co-founder of Intel, Gordon Moore, predicted that the number of transistors per integrated circuit would double every 18 months.

\(^2\) The hardware design process is divided into several steps based on refinement level of abstractions. The next lower level of the specification on a certain abstraction level is called implementation.
presented a method for generation of assembler test programs that systematically probe the micro-architecture of a PowerPC processor. Iwashita et al. [14] use an FSM based processor modeling to automatically generate test programs. Campenhout et al. [7] have proposed a test generation algorithm that integrates high-level treatment of the data path with low-level treatment of the controller. Ho et al [27] have presented a technique for generating test vectors for verifying the corner cases of the design. Recently, Wagner et al. [16] 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.

Model checking based techniques have been successfully applied to processor verification. Ho et al. [24] extract controlled token nets from a logic design to perform efficient model checking. Jacobi [6] has proposed a methodology to verify out-of-order pipelines by combining model checking with theorem proving for the verification of the pipeline. Like divide-and-conquer, compositional model checking [28] is used to verify a processor microarchitecture containing most of the features of a modern microprocessor. Clarke et al. [9], [10] have presented an efficient algorithm for generation of counterexamples and witnesses in symbolic model checking. Bjesse et al. [23] have used counterexample guided abstraction refinement to find complex bugs. Mishra [25] applied module level of design to reduce the test generation time for microprocessor validation.

Bounded Model Checking (BMC) [3], [4] was introduced to alleviate state explosion problem in model checking. BMC searches for counterexamples under a particular length instead of exhaustive searching. Using a SAT solver such as zChaff [21], SATO [15], or BerkMin[12], SAT-based BMC [8] converts a BMC problem into a propositional SAT problem and then prove or disprove it. Amla et al. [22] compared the performance between SAT-based BMC and BDD (Binary Decision Diagram)-based BMC. McMillan [19] has proposed a technique to use SAT solvers in unbounded model checking. Parthasarathy et al. [13] have presented a safety property verification framework using sequential SAT and bounded model checking.

To the best of our knowledge, the functional test generation technique using SAT-based BMC has not been proposed before in the context of pipelined processor validation.

3 Background

3.1 Model Checking

As an automated formal verification technique, model checking [11] proves mathematically that a design (implementation) satisfies a property. Specification is a collection of properties. Verification procedure using model checking includes:

- Formal modeling of a design
- Creating formal properties
- Proving or disproving

A design is modeled as a state transition graph, called a Kripke structure [30], which is a four-tuple model \( M = (S, S_0, R, L) \). \( S \) is a finite set of states. \( S_0 \) is a set of initial states, where \( S_0 \subseteq S \). \( R: S \rightarrow S \) is a transition relation between states, where for every state \( s \in S \),
there is a state \( s' \in S \) such that the state transition \((s, s') \in R, L: S \to 2^{AP}\) is the labeling function to mark each state with a set of atomic propositions \((AP)\) that hold in that state. A path in the structure, \( \pi \in M \) from a state \( s \), is a computation of the implementation which is an infinite sequence of states and transitions, \( \pi = s_0 s_1 s_2 \ldots \) such that \( s_0 = s \) and \( R(s_i, s_{i+1}) \) holds for all \( i \geq 0 \). Temporal behavior of the implementation is the computations represented by a set of paths in the structure.

Properties are expressed as propositional temporal logic that describes sequences of transitions on the computation paths of expected design behavior. A property is composed of three things as follows:

- Atomic propositions: variables in the design
- Boolean connectives: AND, OR, NOT, IMPLY, etc
- Temporal operators, assuming \( p \) is a state or path formula:
  - \( Fp \) (Eventually): True if there exists a state on the path where \( p \) is true
  - \( Gp \) (Always): True if \( p \) is true at all states on the path
  - \( Xp \) (Next): True if \( p \) is true at the state immediately after the current state
  - \( p_1 U p_2 \) (Until): True if \( p_2 \) is true in a state and \( p_1 \) is true in all proceeding states

For example, we will write a property \( G(req -> F(ack)) \) to describe a behavior that if \( req \) is asserted then the design must eventually reach a state where \( ack \) is asserted.

Given a formal model \( M = (S, S_0, R, L) \) of a design and a propositional temporal logic \( p \) of a property, the model checking problem is to find the set of all states in \( S \) that satisfy \( p \), \( \{ s \in S \mid M, s \models p \} \). The design satisfies the property if all initial states are in the set. If the property does not hold for the design, error trace from the error state to an initial state is given as a counterexample that helps designers debug the error. To achieve 100% confidence of correctness of the design, the specification should include all the properties that the design should satisfy. However, it is very hard to determine the completeness of specification. Therefore, model checking is widely used as an alternative to simulation-based validation.

### 3.2 SAT Solving

Boolean satisfiability (SAT) problem is to determine whether there exists a variable assignment such that a propositional formula evaluates to true. If there exists such an assignment, the formula is called *satisfiable*. Otherwise, the formula is said to be *unsatisfiable*. For example, the formula \((a \lor \neg b) \land (\neg b \lor c) \land (\neg c \lor \neg a)\) is satisfiable at \( a=1, b=0, \) and \( c=0 \). SAT solver is a tool to check satisfiability of a given Boolean formula represented in Conjunctive Normal Form (CNF).

### 3.3 SAT-based Bounded Model Checking

Bounded Model Checking (BMC) was proposed to overcome the state explosion problem in conventional model checking. Instead of exhaustive search, BMC searches for a counterexample of a particular length \( k \), called bound or maximum length of

---

3 CNF is a conjunction of clauses, each clause is a disjunction of literals, and a literal is a Boolean variable or its negation. For example, the formula \((a \lor \neg b) \land (\neg b \lor c) \land (\neg c \lor \neg a)\) conforms the CNF.
counterexamples. In SAT-based BMC, the BMC problem is encoded into the satisfiability problem and a SAT solver is used as verification engine instead of a model checker. To perform verification, SAT-based BMC includes following steps:

- Unfold design and property up to the bound $k$
- Encode the bounded design and property into a CNF formula
- Apply the CNF formula to a SAT solver
- If satisfiable, then the property does not hold for the design and the satisfiable assignment of variables is converted to a counterexample
- If unsatisfiable, the property may or may not hold for the design

The CNF formula is satisfiable if and only if a violated state is reachable within the bound $k$. The resulting satisfiable assignment of variables is translated into an error trace from a valid initial state to the violated state. If the bound $k$ is equal to or larger than the diameter and the CNF formula is unsatisfiable, then the design satisfies the property because there is no counterexample in the state space. However, if the bound $k$ is smaller than the diameter and the CNF formula is unsatisfiable, then SAT-based BMC cannot prove or disprove the correctness of design since states beyond the bound $k$ still remain unchecked.

4 Functional Test Generation

This section presents the existing technique using model checking and a test generation technique using SAT-based BMC. The proposed approach does not suffer from the state explosion problem in the existing test generation using model checking.

4.1 Test Program Generation using Model Checking

![Figure 4.1: Test generation using model checker](image)

Figure 4.1 shows the existing test generation technique using model checking. Specification is described in a formal model using model checking language and a

---

4Diameter is the maximum length of the shortest path between any two states in the state space of a given design.
property is expressed in temporal logic. A model checker exhaustively searches all reachable states of the specification to check out whether the property holds or not. If the model checker detects no violation, then the property holds. If it finds any reachable state that does not satisfy the property, it produces a counterexample. This feature of model checking can be quite effectively exploited for test generation. If a property holds true, then its negated property will be false, making model checker generate a counterexample to the design specification. The produced counterexample is exactly one of the test cases for the original property. As described in Figure 4.1, specification and a negated property are applied to a model checker and the model checker generates a counterexample. If specification of a pipelined processor is applied, the sequence of fetched instructions in the counterexample is a test program for validation of the property before negation. The main advantage is that model checking enables automatic test generation by using counterexamples of model checker.

Existing model checking based test generation approaches have proposed several promising ideas for efficient test generation. However, many challenges still remain in applying them to realistic pipelined processors due to the capacity limitation of the model checking tool. In addition, the time and resources required for test generation can be extremely large due to the state explosion problem in model checking.

4.2 Test Program Generation using SAT-based BMC

![Diagram of SAT-based BMC test generation]

The proposed test program generation technique is applicable to a specification-driven test generation methodology where test programs are generated from specification without knowing the details of its implementation. Figure 4.2 describes the proposed SAT-based BMC test generation technique. Microprocessor specification is modeled
using a model checking language. A property is expressed in temporal logic and then negated to generate a counterexample. Based on the bound length of counterexamples, the processor model and the negated property are converted into a CNF formula. In other words, model checking problem is converted into SAT problem. A SAT solver takes the CNF formula as input and verifies that the formula is satisfiable. Since the design does not satisfy the negated property, the SAT solving results in satisfiable and produces satisfiable assignment of variables. The satisfiable assignment of variables is decoded into a counterexample. A test program is the required sequence of fetching instructions in the counterexample. Simulator takes the test program as input and generates expected behavior of an implementation of the microprocessor.

For example, to generate a test program for verifying “read after write (RAW) stall at Decode stage”, we write the negated property that “RAW stall at Decode stage is never exercised”. The SAT-based BMC tool takes this negated property, specification of a microprocessor, and the bound and then produces a counterexample of bounded length that activates the “RAW stall at Decode stage.” Primary inputs of the counterexample show an instruction sequence to test the property. By restricting search space for counterexamples, SAT-based BMC is advantageous in reducing time and memory requirement for functional test generation.

5 A Case Study

We present our experimental setup followed by a test generation example. Next, we compare our test generation technique with the existing unbounded model checking approach in terms of test generation time.

5.1 Experimental Setup

![MIPS processor architecture](image)

Figure 5.1: MIPS processor architecture
We applied the proposed test generation technique on a multi-issue MIPS architecture [18]. Figure 5.1 shows a simplified version of the architecture with four pipeline paths. We used Cadence SMV [20] model checker along with zChaff SAT-solver to perform all the experiments since Cadence SMV has an interface to zChaff. The interface encodes the design and the temporal property into a CNF formula according to the bound and it also decodes satisfiable assignment of variables into a counterexample. We made few simplifications to the MIPS processor for the purpose of comparing test generation time between model checking technique and the proposed SAT-based BMC approach. For example, if 32 32-bit registers are used in the register file, the existing approach does not produce any counterexample even for a simple property with no pipeline interactions. We used 16 or 8 2-bit registers for the following experiments to ensure that the model checking approach can generate counterexamples. All the experiments were run on a 1 GHz Sun UltraSparc with 8G RAM.

5.2 Test Generation: An Example

A multiple exception scenario at the same time is considered as one of pipeline interaction behaviors. Three functional units are in exception at clock cycle 7: overflow exception in IALU, divide-by-zero exception in DIV unit, and a memory exception in the MEM unit. The original property, $P$, is shown as below:

$$P: F( (clk=7) \& (MEM\text{-}exception = 1)$$
$$\& (IALU\text{-}exception = 1)$$
$$\& (DIV\text{-}exception = 1))$$

The negated property, $P'$, is shown as below:

$$P': G( (clk\neg{}=7) \mid (MEM\text{-}exception \neg{} = 1)$$
$$\mid (IALU\text{-}exception \neg{} = 1)$$
$$\mid (DIV\text{-}exception \neg{} = 1))$$

The negated property and the design described in SMV language are applied to SMV tool using the bound 7 as follows:

```
smv –bmc –l 7 –satsolver zchaff Mips.smv
```

The interface between SMV and zChaff SAT-solver encodes them into CNF formula. The SAT-solver tries to find a satisfiable assignment of variables. Since the negated property does not satisfy the design, the SAT solver generates a satisfiable assignment and the interface decodes it into a counterexample. The generated counterexample includes primary input requirement of Fetch unit, that is, a test program to validate the original property. The generated test program is shown in Table 5.1. The first column presents the time at fetching an instruction. The second column through fifth column shows a test program in the form of VLIW instructions. The proposed approach required 9.31 seconds to generate the final test program whereas the model checking approach did not complete in 3 hours when we used 16 2-bit registers in the register file. The test program in Table 5.1 exercises three exceptions: MEM exception by a load
operation with memory address zero, IALU exception by add operation with value 2 for both source operands (result 4 does not fit in a 2-bit register), and DIV exception by a divide operation with second source operand valued zero.

<table>
<thead>
<tr>
<th>Fetch cycle</th>
<th>Instructions ([0] for ALU, …, [3] for DIV)</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ADDI R2 R0 #2 N0P N0P N0P</td>
<td>R2 has 2</td>
</tr>
<tr>
<td>2</td>
<td>NOP N0P N0P N0P</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>NOP N0P N0P N0P</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>LD R1 0(R0) N0P N0P</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>ADD R3 R2 R2 N0P DIV R3 R0 R0 R0 has 0</td>
<td></td>
</tr>
</tbody>
</table>

### 5.3 Results

Table 5.2 describes the comparison of test generation time between unbounded model checking and SAT-based BMC approaches. We used the bound length 5 which is the minimum depth of counterexamples to generate test programs in our experiments. The bound was decided by running the existing model checking approach. The first column shows the type of interaction properties used for test generation. 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 test generation time (in seconds). Since the existing approach using unbounded model checking cannot finish in majority of the cases when more registers are used, we used only 8 2-bit registers. The experimental results show the proposed approach takes several orders of magnitude less test generation time than unbounded model checking approach.

<table>
<thead>
<tr>
<th>Interaction Properties</th>
<th>Unbounded MC</th>
<th>SAT-based BMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>384.70</td>
<td>0.86</td>
</tr>
<tr>
<td>Two Modules</td>
<td>499.23</td>
<td>1.18</td>
</tr>
<tr>
<td>Three Modules</td>
<td>671.13</td>
<td>1.21</td>
</tr>
<tr>
<td>Four Modules</td>
<td>928.89</td>
<td>1.38</td>
</tr>
</tbody>
</table>

### 6 Conclusion

Functional verification is widely acknowledged as a major bottleneck in microprocessor design methodology. This report presented an efficient test program generation technique using SAT-based BMC for functional validation of pipelined processors. Our experimental results using a multi-issue MIPS processor demonstrate that the proposed approach reduces the test generation time by several orders of magnitude.

In the previous section, we assumed that the depth of counterexamples was known in advance. However, in most cases, the bound length is unknown. Therefore, one of
challenges in test generation using SAT-based BMC is to find the smallest bound for each property to minimize the test generation time. Our future work includes development of an efficient way of deciding the bound of counterexamples for test generation of pipelined processors.

7 References

[33] www-cad.eecs.berkeley.edu/~kenmc mil/smv. Cadence SMV.