Functional Test Generation using Property Decompositions for Validation of Pipelined Processors

Heon-Mo Koo
hkoo@cise.ufl.edu

Prabhat Mishra
prabhat@cise.ufl.edu

Department of Computer and Information Science and Engineering
University of Florida, Gainesville, FL 32611, USA.

Abstract

Functional validation is a major bottleneck in pipelined processor design. 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 paper presents an efficient test generation technique using decompositional model checking. The contribution of the paper 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 demonstrate several orders-of-magnitude reduction in memory requirement and test generation time.

1 Introduction

The complexity of pipelined processors is increasing at an exponential rate due to the combined effect of technological advances and availability of increasingly complex applications. To accommodate such faster computation requirements, today's processors employ many sophisticated micro-architectures including deeply pipelined superscalar architectures. Functional validation of such processors is widely acknowledged as a major bottleneck in microprocessor design methodology. Existing processor validation techniques employ a combination of simulation based techniques and formal methods. Simulation is the most widely used form of processor validation. Two types of test programs are used during simulation: random and directed. The directed test vectors are generated based on certain coverage metric such as pipeline coverage, functional coverage, and so on. Directed tests are very promising in reducing the validation time and effort since several orders of magnitude less number of directed tests are required compared to random tests to obtain the same coverage goal. Various techniques have been developed in the past to generate directed test programs. Certain heuristics and design abstractions are used to generate directed testcases. However, due to the bottom-up nature and localized view of these heuristics the generated testcases may not yield a good coverage.

Specification driven test generation has been introduced as a promising top-down validation technique for pipelined processors [16, 17]. A language based specification is used to capture the processor architecture. Various properties are generated from the specification based on pipeline coverage. These properties are used to generate the counterexamples using model checking. The generated counterexample is converted to a test program that consists of instruction sequences. This approach is not suitable for today's pipelined processors since the time and memory requirements can be prohibitively large in many test generation scenarios. We present an efficient test generation technique that uses design level as well as property level decompositions. This paper makes three important contributions: i) it develops a procedure for decomposing a temporal logic property into multiple smaller properties, ii) it presents an algorithm for merging the counterexamples generated by decomposed properties, and iii) it develops an integrated framework to support both design and property decompositions for efficient test generation of pipelined processors.

The rest of the paper is organized as follows. Section 2 presents related work addressing test generation in the context of functional validation of pipelined processors. Section 3 describes our test generation methodology followed by a case study in Section 4. Finally, Section 5 concludes the paper.

2 Related Work

Traditionally, validation of a microprocessor 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. For example, Genesys-Pro [1], used for functional verification of IBM processors, combines architecture and testing knowledge for efficient test generation. Many techniques have been proposed for generation of directed test programs [2, 20].

Ur and Yadin [21] have presented a method for generation of assembler test programs that systematically probe the micro-architecture of a PowerPC processor. Iwashita et al. [9] use an FSM based processor modeling to automatically generate test programs. Campenhout et al. [5] have proposed a test generation algorithm that integrates high-level treatment of the datapath with low-level treatment of the controller. Ho et al. [18] have presented a technique for generating test vectors for verifying the corner cases of the design. Recently, Wagner et al. [11] 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.

3-9810801-0-6/DATE06 © 2006 EDAA

Model checking based techniques have been successfully used in processor verification. Ho et al. [15] extract controlled token nets from a logic design to perform efficient model checking. Jacobi [4] 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 [19]. Parthasarathy et al. [8] have presented a safety property verification framework using sequential SAT and bounded model checking. Model checking based techniques are also used in the context of proving properties or test generation by generating counterexamples. Clarke et al. [7] have presented an efficient algorithm for generation of counterexamples and witnesses in symbolic model checking. Bjesse et al. [14] have used counterexample guided abstraction refinement to find complex bugs.

The work by Mishra et al. [16] on graph-based functional test program generation using model checking is closest to our approach. They proposed a module level decomposition technique to reduce the test generation time. The basic idea of their algorithm is to decompose one processor level property into multiple module level properties and apply them to respective modules. This assumes that the original property contains variables for only one module. In other words, their technique does not handle properties that have variables from different modules. Such properties are common in test generation. For example, a property to generate a test program to stall multiple units will contain variables from multiple modules. Our framework allows such input properties. We present a method to perform decomposition of temporal properties. Moreover, we integrate both module level and property level decompositions to further reduce memory requirement and test generation time. The biggest challenge in performing test generation in the presence of property decomposition is how to merge the counterexamples to obtain the final test program. We present an algorithm for merging the test programs. The contribution of this paper is an efficient test generation technique that allows both design and property decompositions for validation of pipelined processors.

3 Test Generation using Design and Property Decompositions

Figure 1 shows our functional test program generation methodology. In this methodology, the designer uses an Architecture Description Language (ADL) to specify the processor architecture. We specify the processor architecture using EXPRESSION ADL [3]. Our methodology is independent of the ADL. We can use any ADL that captures both the structure and the behavior of the processor. The graph based model of the pipelined processor is generated from the ADL specification as described in Section 3.1. The properties are generated from the ADL specification based on the graph coverage. The property generation and decomposition technique is described in Section 3.2. Finally, Section 3.3 presents our test generation technique using design and property decompositions.

It is important to note that the property and design decompositions are not independent. Table 1 shows four possible scenarios of design and property decompositions. The first scenario indicates the traditional model checking where original property is applied to whole design. The second case 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 of counterexamples may not generate the expected result. For example, two sub-properties may generate counterexamples to stall the respective units in a pipelined processor but the combined test program may not simultaneously stall both the units. The third scenario is similar to the first scenario 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.

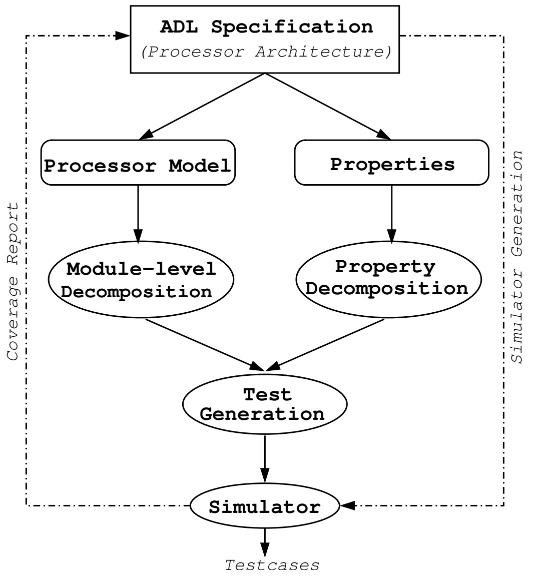

Figure 1. Test Program Generation Methodology

Table 1. Design and property decomposition scenarios

D	P	Comments
0	0	Traditional model checking
0	1	Merging of counterexamples is not always possible
1	0	Similar to traditional model checking
1	1	Our approach, both property and design decompositions

D: Design, P: Property, 0: Original, 1: Decomposed/partitioned.

3.1 Modeling of a Pipelined Processor

Decomposition of a design 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 etc.) or storages (e.g., memory or register file) and *edges* consist of connectivity among them. Typically, instruction is transfered between units, and data is transfered between units and storages. This graph model is similar to the pipeline level block diagram available in a typical architecture manual. The ADL specification captures the block diagram view (structure) of the pipelined processor. The graph model of the pipelined processor is generated from the ADL specification.

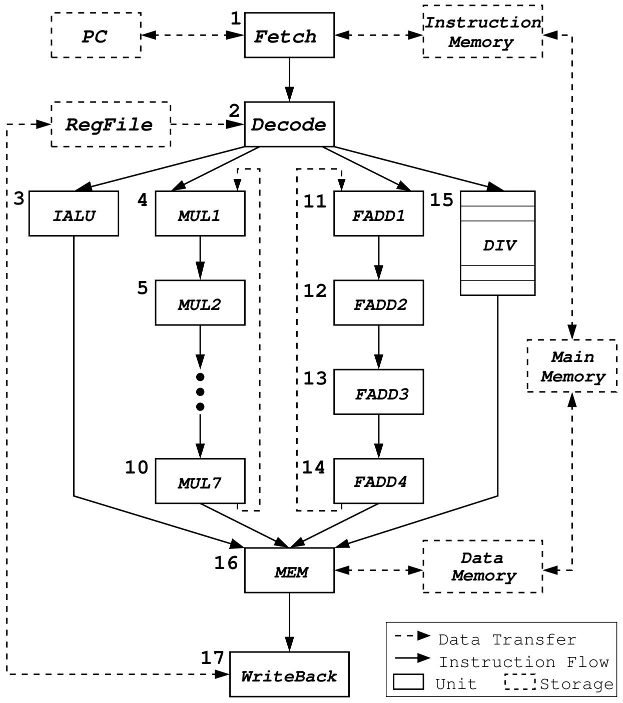

Figure 2. Graph Model of the MIPS Processor

For illustration, we use a simplified version of the multi-issue MIPS processor [12]. Figure 2 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.

3.2 Generation and Decomposition of Properties

We first explain how to generate properties from the ADL specification. Next, we describe how to decompose such properties for efficient test generation. Today's test generation techniques as well as formal methods are very efficient in module level validation. The harder problem is to verify the inter-module interactions. In this paper, 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. We first define the possible pipeline interactions based on the number of nodes in the graph model and the average number of activities in a node. For example, an IALU node can have three activities: executing an operation (active), stalled, and in exception. In general, the number of activities for a node will be different based on what activity we would like to test. For example, executing an ADD or SUB operations can be treated as separate activities. Furthermore, the number of activities may be different for different nodes. Consider a graph model with n nodes where each node can have on average r activities. We require a total of $r(1-r^n)/(1-r)$ properties to verify all interactions. We omit the proof for the interest of space. The basic idea of the proof is that if we consider no interactions, there are $n \times r$ test programs necessary. In the presence of one interaction we need r^2 test programs for each possible combination of two nodes. Once we combine all possible interactions the equation will be:

$$\sum_{i=1}^{n} n_{C_i} \times r^i \qquad (1)$$

Here, n_{C_i} represents the ways of choosing i nodes from a set of n nodes. 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. For each such interaction we generate a property from the ADL specification.

The generated properties are expressed in propositional temporal logic. 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 [6]:

$$\begin{array}{ll} \neg AX(p) = EX(\neg p) & \neg EX(p) = AX(\neg p) \\ \neg AG(p) = EF(\neg p) & \neg EG(p) = AF(\neg p) \\ \neg AF(p) = EG(\neg p) & \neg EF(p) = AG(\neg p) \\ \neg A(pRq) = E(\neg pU\neg q) & \neg E(pRq) = A(\neg pU\neg q) \end{array}$$
$$\neg A(pUq) = E(\neg qU\neg p \wedge \neg q) \vee EG(\neg q)$$

In the remainder of this section, we describe how to decompose these properties (already negated) for efficient model checking. Each property consists of temporal operators (G, F, X, U) and Boolean connectives ($\wedge$, $\vee$, $\neg$, and $\rightarrow$). There are various combinations of temporal operators and Boolean connectives where decompositions are not possible e.g., $F(p \wedge q) \neq F(p) \wedge F(q)$ and $G(p \vee q) \neq G(p) \vee G(q)$. In certain situations, such as pUq, $F(p \rightarrow F(q)$, or $F(p \rightarrow G(q)$, decompositions are not beneficial compared to traditional model checking. The following combinations allow simple property decompositions.

$$\begin{array}{ll} G(p \wedge q) = G(p) \wedge G(q) & F(p \vee q) = F(p) \vee F(q) \\ X(p \vee q) = X(p) \vee X(q) & X(p \wedge q) = X(p) \wedge X(q) \end{array}$$

If we introduce the notion of clock (time step) in the property then more decompositions are allowed as shown below[1]. Note that the left and right hand side of the decomposition are not logically equivalent but they produce functionally equivalent counterexamples.

$$G((clk \neq t_s) \vee (p \vee q)) \approx G((clk \neq t_s) \vee p) \vee G((clk \neq t_s) \vee q)$$

Although we only use a few decomposition scenarios, it is important to note that these scenarios are sufficient for generating the properties where node interactions are considered. Moreover, the property decomposition is dependent on the design decomposition. For example, consider a design which has two

[1]The clk variable is used to count time steps, and t_s is a specific time step.

partitions: d_1 and d_2. We cannot decompose a property into two sub-properties p_1 and p_2, if it is not possible to apply p_1 and p_2 to the partitions d_1 and d_2. In other words, if p_1 contains variables from both partitions, it is not possible to apply it to one partition of the design. As discussed in Section 3.1, the pipelined processor is partitioned into modules. However, we can change the partition based on the properties. For example, a property may not be decomposable based on a module level partitioning but it may be decomposable based on a pipeline path level partitioning as described in Section 4.3.

```
Algorithm 1: Test Generation
Inputs: i) Processor model M as a composition of modules
        ii) Set of global properties P where each property is
            decomposed into multiple module level properties
Outputs: Test programs to verify the pipeline interactions.
Begin
  TaskList = $\phi$; FutureList = $\phi$; TestPrograms = $\phi$
  for each property $P_i$ in set P
    for each sub property $P_i^j$
      TaskList[j] = $P_i^j$ /* $P_i^j$ is applicable to $M_j$ */
    endfor
    PrimaryInputs = $\phi$
    while TaskList is not empty
      $items$ = RemoveEntry(TaskList)
      $P_i^k$ = ComposeRequirements($items$);
      Apply $P_i^k$ on module $M_k$ using model checker
      $inp_k$ = input requirements for $M_k$ from counterexample
      if $inp_k$ are not primary_inputs
        for each applicable parent node $M_r$ of $M_k$
          $out_r$ = Extract output requirements for $M_r$
          FutureList[r] = FutureList[r] $\cup$ $out_r$
        endfor
      else PrimaryInputs = PrimaryInputs $\cup$ $inp_k$
      endif
      if TaskList is empty
        TaskList = FutureList; FutureList = $\phi$
      endif
    endwhile
    $test_i$ = GenerateTest(PrimaryInputs).
    TestPrograms = TestPrograms $\cup$ $test_i$
  endfor
  return TestPrograms
End
```

3.3 Functional Test Generation

Algorithm 1 presents our test generation procedure using design and property decompositions. The basic idea of the algorithm is to apply the components of the properties (sub-properties) to appropriate modules and compose their responses to construct the final test program. This algorithm accepts design M and properties P as inputs and produces the test programs. It uses two lists to maintain the current (TaskList) and future (FutureList) tasks. Both lists have exactly the same structure. Each entry in the list contains a collection of sub-tasks that is applicable to a particular module. Therefore, each list can have up to n entries where n is the number of modules (or partitions) in the design. The tasks in the TaskList need to be performed in the current time step (clock cycle). The tasks in the FutureList will be performed in the next clock cycle. Initially both lists are empty.

For each property P_i, the algorithm generates one test program. Each property consists of one or more sub-properties based on their applicability to different modules or partitions in the design as discussed in Section 3.2. The algorithm adds the sub-properties to the TaskList based on the module to which this property is applicable. The algorithm iterates over all the tasks (sub-properties) in the TaskList. It removes an entry (say k'th location) from the TaskList. In general, this entry can be a list of sub-tasks (due to simultaneous output requirements from multiple children nodes) that need to be applied to module M_k. These subtasks are composed to create the intermediate property P_i^k. The property P_i^k is applied to the module M_k using a model checker. The model checker generates a counterexample. The generated counterexample is analyzed to find the input requirements inp_k for the module M_k. If these are primary inputs then they are stored in PrimaryInputs list; otherwise for each parent node M_r, where inp_k is applicable, extract the output requirements for M_r. This output requirement is added to the r'th entry of the FutureList. Finally, if the tasks for the current timestamp is completed (TaskList empty), FutureList is copied to the TaskList and this process continues until both the lists are empty. This implies we have obtained the primary input assignments for all the sub-properties. These assignments are converted into a test program consisting of instruction sequences.

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 2 at clock cycle 5. This property can be decomposed into two sub-properties 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; generated counter example will be analyzed to find the output requirement for the Decode unit (2nd module in Figure 2) in clock cycle 4; the requirement will be added to FutureList[2]. During second iteration of the while loop P_1^{15} (TaskList[15]) will be applied to M_{15} (DIV); generated counter example will be analyzed to find the output requirement for the decode unit in clock cycle 4; the requirement will be added to FutureList[2]. At this point, the TaskList is empty and the FutureList 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. Section 4.2 shows a test generation example for a multiple exception scenario using module-level design decompositions. Section 4.3 shows another test generation example using pipeline path level partitioning of the design.

4 A Case Study

We performed various test generation experiments for validating the pipeline interactions by varying different design partitions and property decompositions. In this section we present our experimental setup followed by two test generation scenarios for different design partitions. Next, we compare our test generation technique with two other approaches: i) *naive* approach where the original property is applied to the whole design, and ii) *existing* approach based on Mishra et al. [16].

4.1 Experimental Setup

We applied our methodology on a multi-issue MIPS architecture [12]. Figure 2 shows a simplified version of the architecture. We have chosen MIPS processor for two reasons. First, it has been well studied in academia and there are HDL implementations available for the processor that can be used for validation purposes. Second, it has many interesting features, such as fragmented pipelines and multi-cycle functional units that are representative of many commercial pipelined processors including TI C6x and PowerPC.

We used SMV [10] model checker to perform all the experiments. We made few simplifications to the MIPS processor for the naive approach to work. For example, if 32 32-bit registers are used in the register file, the naive approach does not produce any counterexample even for a simple property with no node interactions. We used 8 2-bit registers for the following experiments to ensure that the naive approach can generate counterexamples. All the experiments were run on a 1 GHz Sun UltraSparc with 8G RAM.

4.2 Test Generation: An Example

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 original property (SMV description), P, is shown below:

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

The negated property, P’, is shown below:

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

P’ is decomposed into three sub-properties:

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

Based on Algorithm 1, the sub-properties 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 module. For example, the counterexamples for P1, P2, and P3 respectively are: (C_{P1}) a load operation with memory address zero, (C_{P2}) an add operation with value 2 for both source operands (result 4 does not fit in a 2’bit register), and (C_{P3}) a 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})[2] is applied to Decode unit in the next step. In each case, clock cycle value is reduced by one as shown below:

```
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 (generate primary inputs PI_i) in timestep 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
```

4.3 Test Generation using Path-level Partitioning

The example shown above assumes a module-level partitioning of the design. However, it may not always be 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 2 are activated at the same time, it is not possible to decompose this property (shown below) based on module level decompositions.

```
/* 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 -> X(˜FADD1.feedIn))
               | (MUL7.feedOut -> X(˜MUL1.feedIn)))
/* Properties after Decomposition*/
P1: G((clk˜=9) | (FADD4.feedOut -> X(˜FADD1.feedIn))
P2: G((clk˜=9) | (MUL7.feedOut -> X(˜MUL1.feedIn))
```

It is obvious that when a decomposed property contains variables from multiple modules, it is not possible to apply them to individual modules. To enable property decomposition in the above 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}. The remaining design can be partitioned as modules for this example.

[2]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.

Algorithm 1 can be applied using this new partitioning 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 applied to the Decode unit, and so on. Finally, the primary input requirements can be converted to generate the final test program.

4.4 Results

In this section we compare our approach with two other approaches: i) naive approach where the original property is applied to the whole design, and ii) the approach presented by Mishra et al. [16] which is closest to our approach. We refer the second approach as the *existing* approach.

Table 2. Comparison of Test Generation techniques

Module	Naive Approach		Existing Approach		Our Approach	
Interactions	BDD	Time	BDD	Time	BDD	Time
None	6 M	165	3 K	0.06	3 K	0.06
Two Modules	11M	215	NA	NA	6 K	0.12
Three Modules	21M	240	NA	NA	9 K	0.19
Four Modules	27M	290	NA	NA	11K	0.28

NA: Not Applicable.

Table 2 presents the results of the comparison of test generation techniques. The first column defines the type of 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 average values for the BDD nodes used as well as test generation time (in seconds) for one property. For example, the first row considers 85 (n=17, r=5, and i=1 in Equation (1)) single module properties, and different properties take different amount of BDD nodes and test generation time. The table shows the average value for both BDD nodes and test generation time. The existing approach is only applicable to the first row since it cannot handle multiple simultaneous properties or property decompositions. As mentioned earlier, the naive approach cannot finish in majority of the cases when more registers are used. As a result we used only 8 2-bit registers. In spite of this simplification naive approach takes several orders of magnitude more memory and test generation time.

5 Conclusions

Functional verification is widely acknowledged as a major bottleneck in microprocessor design methodology. This paper presented an efficient test generation technique based on decomposition of both design and properties for functional validation of pipelined processors. This paper made three important contributions. First, it developed a procedure for decomposing propositional temporal logic properties based on various partitioning of pipelined processors. Second, it presented an algorithm for merging the counterexamples generated by decomposed properties. Finally, it presented an integrated framework for efficient test generation that supports both design and property decompositions. Our experimental results using a multi-issue MIPS processor demonstrate that our technique reduces the memory requirement and improves the test generation time by several orders of magnitude.

The increasing complexity of today's pipelined processors implies an increase in both number of pipeline components and behaviors of each component (variables n and r in Equation (1)). As a result, the number of test programs required to verify such interactions are increasing at an exponential rate. It is necessary to develop efficient tools, techniques and methodologies to perform efficient test generation as well as reduce the number of test programs. This paper presented an efficient test generation technique based on design and property decompositions. Our future work includes development of test compaction techniques to reduce the number of test programs for validation of pipeline interactions.

References

[1] A. Adir et al. Genesys-pro: Innovations in test program generation for functional processor verification. *Design & Test*, 2004.

[2] A. Aharon et al. Test program generation for functional verification of PowerPC processors in IBM. *DAC*, 1995.

[3] A. Halambi et al. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. *DATE*, 1999.

[4] C. Jacobi. Formal verification of complex out-of-order pipelines by combining model-checking and theorem-proving. *CAV*, 2002.

[5] D. Campenhout et al. High-level test generation for design verification of pipelined microprocessors. *DAC*, 1999.

[6] E. Clarke, O. Grumberg and D. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[7] E. Clarke et al. Efficient generation of counterexamples and witnesses in symbolic model checking. *DAC*, 1995.

[8] G. Parthasarathy et al. Safety property verification using sequential SAT and bounded model checking. *Design & Test*, 2004.

[9] H. Iwashita et al. Automatic test pattern generation for pipelined processors. *ICCAD*, 580–583, 1994.

[10] www-cad.eecs.berkeley.edu/˜kenmcmil/smv. *Cadence SMV*.

[11] I. Wagner, V. Bertacco and T. Austin. Stresstest: An automatic approach to test generation via activity monitors. *DAC*, 2005.

[12] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2003.

[13] M. Behm et al. Industrial experience with test generation languages for processor verification. *DAC*, 36–40, 2004.

[14] P. Bjesse and J. Kukula. Using counter example guided abstraction refinement to find complex bugs. *DATE*, 2004.

[15] P. Ho and A. Isles and T. Kam. Formal verification of pipeline control using controlled token nets and abstract interpretation. *ICCAD*, 529–536, 1998.

[16] P. Mishra and N. Dutt. Graph-based functional test program generation for pipelined processors. *DATE*, 182–187, 2004.

[17] P. Mishra and N. Dutt. *Functional Verification of Programmable Embedded Architectures – A Top-Down Approach*. Springer, 2005.

[18] R. Ho et al. Architecture validation for processors. *ISCA*, 1995.

[19] R. Jhala and K. L. McMillan. Microarchitecture verification by compositional model checking. *CAV*, 2001.

[20] S. Fine and A. Ziv. Coverage directed test generation for functional verification using bayesian networks. *DAC*, 286–291, 2003.

[21] S. Ur and Y. Yadin. Micro architecture coverage directed generation of test programs. *DAC*, 175–180, 1999.