High-Level Test Generation for Design Verification of Pipelined Microprocessors

David Van Campenhout, Trevor Mudge, and John P. Hayes

Department of Electrical Engineering and Computer Science
The University of Michigan, Ann Arbor, MI 48109-2122, USA
{davidvc, tnm, jhayes}@eecs.umich.edu

Abstract

This paper addresses test generation for design verification of pipelined microprocessors. To handle the complexity of these designs, our algorithm integrates high-level treatment of the datapath with low-level treatment of the controller, and employs a novel "pipeframe" organization that exploits high-level knowledge about the operation of pipelines. We have implemented the proposed algorithm and used it to generate verification tests for design errors in a representative pipelined microprocessor.

Keywords: design verification, sequential test generation, high-level test generation, pipelined microprocessors.

1. Introduction

Design verification is considered one of the most serious bottlenecks for multimillion-gate microprocessor designs. There are two broad approaches to hardware design verification: formal and simulation-based. Formal methods try to verify the correctness of a system by using mathematical proofs. Simulation-based design verification tries to uncover design errors by detecting a circuit's faulty behavior when deterministic or pseudo-random tests (simulation vectors) are applied [3, 9, 23]. The effectiveness of verification test suites is quantified by coverage metrics that include code coverage measures from software testing [5], finite-state machine coverage [14], architectural event coverage [23], and observability-based metrics [11]. A shortcoming of all these metrics is that the relationship between the metric and the detection of classes of design errors is not well specified or understood.

An alternative verification approach draws on the similarity between hardware design verification and physical fault testing [1, 4, 24]. In this approach, synthetic error models are derived from empirical design error data, and physical fault testing techniques are adapted to generate test sets for the synthetic errors. Due to the gap in abstraction level between the implementation and the specification, the test generation problem must be solved for a very large sequential circuit. Circuits of the size and complexity of pipelined microprocessors far exceed the capabilities of current gate-level sequential test generation algorithms.

This paper addresses test generation targeted at synthetic errors for design verification of pipelined microprocessors, and proposes a new test generation method that exploits high-level knowledge about these designs. We review relevant previous work in Section 2. Our high-level model for pipelined processors is presented in Section 3. The iterative organization of the proposed high-level test generation algorithm is described in Section 4. The algorithm is described in Section 5. We present experimental results in Section 6, and give some concluding remarks in Section 7.

2. Related Work

Design verification of pipelined microprocessors with respect to their instruction set architecture (ISA) specification has been tackled using formal methods. In some early work [6] the implementation and the specification are symbolically simulated and the outcome is compared using a given mapping that relates corresponding time points for the outputs. This method assumes that the width of the datapath can be reduced to cope with the state explosion problem. Burch and Dill [8] abstract the datapath using a quantifier-free first-order logic with uninterpreted functions. Their method constructs the next-state functions of the implementation and the specification, which are then checked for equivalence. Levitt and Olukotun [19] develop a methodology for verifying the control logic of pipelined microprocessors that uses the same datapath abstraction. They iteratively merge the two deepest stages of the pipeline and check whether the newly obtained pipeline is still equivalent to the previous one. High-level information, similar to that in our approach, is central to achieving a high degree of automation.

A class of hybrid verification techniques [12, 14, 20, 21] that combine simulation with formal verification has recently been proposed. These techniques construct a reduced FSM model of the implementation. A test set is generated that achieves full state-transition coverage on the reduced FSM model. That test set is transformed so that it can be applied to the implementation. The implementation and the specification are then simulated for the transformed test set.

Typical fault-oriented test generation techniques for gate-level sequential circuits [10] iteratively apply a test generation algorithm for combinational circuits using the iterative logic array (ILA) model of the circuit. Circuits of the size and complexity of pipelined microprocessors still exceed the capabilities of current gate-level sequential test generation methods. One approach to address this complexity is to perform test generation at a higher level of abstraction. Lee and Patel [18] describe a method to generate tests for microprocessors, modeled as a datapath and a set of control behaviors. Test generation is split into path selection and value selection phases. Iwashita et al. [16] describe a technique for generating instruction sequences to excite given "test cases", such as hazards, in pipelined processors. Test generation at the microarchitectural level is presented in [9, 15], but is not suitable for targeting concrete structural errors.

3. Pipelined Processor Model

An important element of microprocessor structure is the distinction between *data* and *control*. The merits of treating datapaths and controllers differently have been recognized in many other domains such as high-level synthesis, formal verification, etc. In today's design methodologies, controllers are usually described by behav-

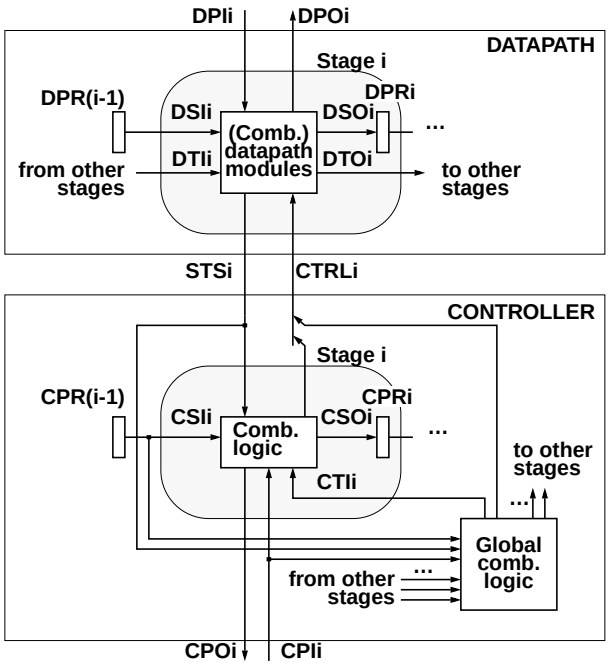
ioral HDL code (case statements). These descriptions are then synthesized into gate-level or transistor-level netlists either by CAD tools or by hand. Most signals appearing in controller descriptions are unstructured binary signals. Controllers are essentially sets of interacting finite-state machines. Datapaths, on the other hand, process structured data words and so can be represented at a higher level than the gate level, using high-level, multibit modules and buses. This high-level representation drastically reduces the size of the design representation.

From a verification point of view, it is also important to distinguish machine state that is visible to the specification, typically an ISA model, from machine state that is specific to the implementation. In pipelined microprocessors the implementation-specific machine state consists of the pipeline registers. Much of the complexity of these processors results from the interaction between multiple instructions in the pipeline. If instructions were to interact only through the ISA-visible part of the machine state, they could be treated independently for verification test generation. However, there is also interaction through the implementation-specific machine state, and this is intimately related to pipeline hazards. Hennessy and Patterson [13] define three standard techniques for dealing with pipeline hazards: *stalling*, *squashing* and *bypassing*. The signals that control these mechanisms are of interest because they reveal the essence of instruction interaction in the pipeline. They provide a means to characterize the control state of the pipeline in a much more compact way than by considering all the instructions in the pipeline simultaneously.

We are developing the model for pipelined processors, shown in Figure 1, that exposes high-level knowledge that can be used during test generation. The datapath and controller both exhibit pipeline structure and interact via status and control signals. The signals at each stage are classified as:

- *primary*: interfacing with the environment
- *secondary*: interfacing with the stage's pipeline registers
- *tertiary*: interfacing with another pipeline stage

The tertiary signals introduced here are precisely the signals needed to describe essential instruction interaction. Typical examples of tertiary signals in the controller are *squash* and *stall* signals; typical examples of tertiary signals in the datapath are bypasses. Imposing the model requires no more than the appropriate labeling of control signals, status signals, and pipe registers, along with appropriate high-level modeling of the datapath.



Dxx: data signal
Cxx: control signal
xPI (PO): primary input (output)
xSI (SO): secondary input (output)
xTI (TO): tertiary input (output)
STS: status signal
CTRL: control signal
DPR: data pipe register
CPR: control pipe register

Figure 1. Pipelined microprocessor model.

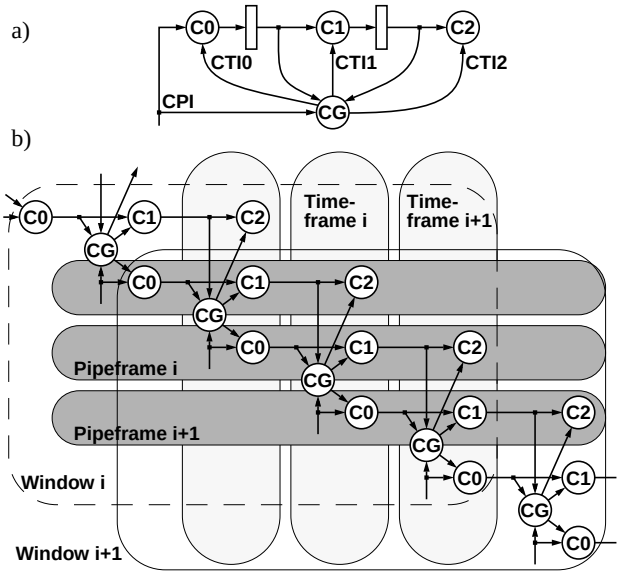


Figure 2. a) Pipelined controller; b) iterative array

4. Pipeframe Model

Conventional fault-oriented test generation algorithms for sequential circuits use the ILA model and iteratively apply test generation techniques for combinational circuits in one timeframe. In this section we describe a different organizational model specific to pipelined processors. This *pipeframe* organizational model exploits high-level knowledge about pipeline structure that is captured with the processor model. The advantages of this approach are a reduction of the search space and the elimination of many conflicts.

Consider the application of a conventional test generation algorithm to a pipelined controller circuit without a datapath. Figure 2a shows a three-stage pipelined circuit. *C*0, *C*1 and *C*2 are combinational logic corresponding to the three pipe stages. The global combinational logic *CG* sources all CPI's and all CSI's. In order not to clutter the figure, the CPI's sourced by *Ci* and the CPO's produced by *Ci* have been omitted. The ILA model for this circuit is shown in Figure 2b. If PODEM [2] is used as the combinational test generation algorithm, the decision variables are the CPI's and the CSI's in each timeframe. The state space to be searched during each iteration is that of the CSI's and CPI's. For the controller of pipelined microprocessors, the number of CSI's (state bits) is typically much larger than the number of CPI's. This is because the primary function of the controller is to decode the incoming instructions.

Taking into account that the circuit is pipelined and performs several concurrent, and to a large extent independent, decodes, a

different organization of the search, one that is directly in terms of the CPI's, is desirable. When the global control logic *CG* is absent, it is easy to see how this can be accomplished. In this case, the ILA consists of unconnected (horizontal) slices spanning a number of timeframes equal to the number of pipe stages. These horizontal slices will be referred to as *pipeframes*. It can be seen that the size of the circuit to be considered is exactly the same as that in the conventional time-frame based search, although the depth is greater. However, in the new approach conflicts due to unreachable states cannot arise as decisions are made only on the CPI's.

In general, there is interaction between pipestages through the global combinational logic *CG*. To organize the search by pipeframe, the tertiary signals *CTIi*, $i = 0, \ldots, 2$, need to be included as decision variables. The ILA is partitioned into pipeframes by cutting the tertiary signals, also shown in Figure 2b. A complication is that a pipeframe directly interacts with a number of other pipeframes via shared primary inputs and via the tertiary signals feeding the pipeframe. In the conventional organization, each timeframe depends directly only on the previous timeframe. To cope with this complication, multiple pipeframes need to be considered simultaneously during the search. The set of pipeframes directly relevant to pipeframe *i* is indicated by *window i* in the figure.

Consider an *p*-stage pipelined controller with a total of n_1 CPI's, n_2 CSI's per pipestage, and n_3 CTI's per pipestage. In the usual timeframe organization, there are $n_1 + p.n_2$ decision variables per timeframe, $p.n_2$ of which need justification. In our pipeframe approach, there are $n_1 + p.n_3$ decision variables per pipeframe, $p.n_3$ of which need justification. Our approach is targeted at the circuits with $n_3 << n_2$. For such circuits the following can be observed:

- The size of the search space in the pipeframe organization is significantly smaller than that in the usual timeframe organization.
- The size of the circuit to be dealt with in the pipeframe organization is comparable to that in the conventional organization, although its depth is greater.

For some pipelined controllers the pipeframe approach does not reduce the search space. This is the case when CSO*i* depends on CSI*i*+1 (referring to Figure 1) for every pipestage. For such circuits, all CSI's are also CTI's, the pipeframe approach reduces to the usual timeframe approach.

5. Test Generation Algorithm

Our high-level test generation algorithm is targeted at localized errors, such as those described in [24], in the datapath. It follows the iterative organization described above and decomposes the test generation problem into three subproblems: 1) path selection in the datapath, 2) value selection in the datapath, and 3) justification of control signals (controller). The subproblems are addressed by *DPTRACE*, *DPRELAX*, and *CTRLJUST*, respectively.

DPTRACE. *DPTRACE* computes a set of justification and propagation paths in the datapath to activate the error and expose the error effect at a primary output of the datapath. *DPTRACE* does not consider the values that need to be justified and propagated.

From the high-level description of the datapath, a controllability/observability graph (COG) is derived. Its nodes correspond to datapath modules, nets with multiple fanout, primary inputs and outputs, tertiary inputs and outputs, and control and status signals; its edges correspond to pairs of connected ports (module terminals) in the datapth. We distinguish three classes of basic datapath modules: ADD, AND, MUX.

Modules in the ADD class have one data output, and one or more data inputs. They have the property that the output can be justified (to an arbitrary value) by controlling only a single input, i.e., regardless of the values of the other inputs, the controlled input can be assigned a value that will justify the output. Also, if the output is observable than every input is observable as well. Modules in the AND class have the property that in order to justify the output (to an arbitrary value) all inputs need to be controlled. To observe an input, the output needs to be observable and all side inputs need to be controlled. Modules in the MUX class have one data output, one or more data inputs, and one or more control inputs. The control inputs are driven by CTRL signals, and determine which data input is selected. In order to justify the output, the control inputs need to be assigned and the selected data input needs to be controlled; the other data inputs are free. In order to observe a data input, the output needs to be observable, and the control inputs need to be assigned such that the requested data input is selected. Multiple fanout nodes have the property that only one fanout can be justified by controlling the stem.

Edges in the COG are attributed with symbolic values that encode controllability information. The attribute, the *C*-state, assumes values from the set {*C*1, *C*2, *C*3, *C*4}. The interpretation of these values is as follows:

- *C*1: it is unknown whether the edge can be controlled
- *C*2: the edge cannot be controlled, but there are still open decisions in the transitive fanin of the edge
- *C*3: the edge cannot be controlled and there are no more open decisions in the transitive fanin of the edge
- *C*4: the edge is controlled

Similarly, information about an edge's observability is represented by the *O*-state, which assumes values from the set {*O*1, *O*2, *O*3}. The interpretation of these values is as follows:

- *O*1: it is unknown whether the edge can be observed
- *O*2: the edge is not observable
- *O*3: the edge is observable

The path selection problem is that of finding a valid assignment to the *C*-state and *O*-state of all edges, and to the CTRL signals to the pipeframe such that the edged associated with the error both controllable (*C*4) and observable (*O*3). The path selection problem can be solved with a PODEM-like directed search with decision variables the CTRL lines and variables that are associated with nets with multiple fanouts.

DPRELAX. *DPRELAX* determines values for DPI's that expose the error effect and justify any STS signals assigned by *CTRLJUST*. We use a discrete relaxation algorithm, similar to that proposed in [17], to solve the problem.

CTRLJUST. *CTRLJUST* derives an input sequence that, when applied to the circuit, and starting from its reset state, makes the controller produce the desired values on the CTRL lines as requested by *DPTRACE*. *CTRLJUST* uses a PODEM-like search with the CPI, CTI, and STS signals as decision variables. Decisions on CTI signals need further justification. Decisions on STS signals need to be justified by *DPRELAX*.

6. Experiments

We have built a prototype implementation of the proposed test generation algorithm. We are using a version of the DLX microprocessor [13] as a test vehicle. This design implements 44 instructions, has a five-stage pipeline and branch prediction logic, and consists of 1552 lines of structural Verilog code. The controller has 95 bits of state; the number of tertiary signals in the controller is 43. The pipeframe approach reduces the number of decision variables that need justification from 95 to 43 compared to the conventional timeframe approach. The datapath has 512 bits of state, not includ-

Table 1. Verification test generation for bus SSL and standard SSL errors in datapath of DLX implementation.

Parameter	High-level: our method with bus SSL errors	Gate-level: HITEC with standard SSL errors
Total no. of errors	316	385
No. of errors detected	274	278
No. of errors aborted	42	93
No. of undetectable errors	N/A	14
Efficiency [%]	86	75
CPU time [minutes]	17	46

ing those in the register file. The high-level model of the datapath consists of 100 combinational modules.

We targeted our test generation system at all bus single stuck line (bus SSL) errors [7] in the decode, execute, memory and write-back stages of the datapath. Although our test generation algorithm can be used in conjunction with other error models proposed in [24], the bus SSL model was chosen for these initial experiments because it defines a number of error instances linear in the size of the circuit. The results are summarized in Table 1. A total of 316 errors were targeted; test generation succeeded for 86% of these errors. The overall algorithm performed only 50 backtracks for the successful errors. Analysis of the 44 aborted errors revealed that 8 of them are undetectable, 2 failed because *DPTRACE* exceeded the maximum number of backtracks, and 14 errors require a non-sequential instruction stream (branches). The remaining 18 errors require error propagation through STS signals, which is not yet supported. The current implementation does not use error simulation, and much re-use of work in the algorithm has yet to be exploited. Therefore, we can expect that addressing these issues will significantly improve run times.

We also synthesized a gate-level model of the design, consisting of 10,757 gates and flipflops. We used HITEC [22] to generate tests for standard SSL errors. For each bus SSL error in the high-level design, we selected SSL errors corresponding to bits 0, 15, 16 and 31. For some of the signals in the high-level design we were not able to identify corresponding signals in the gate-level design. Also, the number of SSL errors in Table 1 is after error collapsing, which is not the case for the number of bus SSL errors reported. As can be seen from the table, our method compares favorably with HITEC. Analysis of the gate-level test generation results revealed that HITEC has great difficulty generating tests for errors that require a sequence of instructions with register dependencies. Gate-level test generation succeeded for only one of a total of 14 forwarding paths.

7. Discussion

In the experiments described above, the instruction stream has been modeled by primary inputs. This model is sufficient for generating test sequences that do not involve branches. However, some errors can only be exposed through the address used to fetch an instruction (program counter). A model for the instruction stream needs to be developed that takes into account the dependency between an instruction and its address. A related problem is that of modeling memory arrays that are not part of the ISA, such as branch target buffers. To handle these structures efficiently, the model needs to hide the individual memory cells underlying the structure, while retaining the association of data with addresses.

We are developing a system for automatically generating targeted test sequences for design verification of pipelined microprocessors. To handle the complexity of these designs, our algorithm integrates high-level treatment of the datapath with low-level treatment of the controller and exploits high-level knowledge about pipeline structure. We have developed a model that captures high-level knowledge about pipeline structure. As the analysis here shows, the pipeframe approach can significantly reduce the search space. Our high-level test generation method compares favorably with gate-level test generation in the experiments.

Acknowledgment

This research is supported by DARPA under Contract No. DABT63-96-C-0074.

References

[1] M. S. Abadir, J. Ferguson, and T. E. Kirkland. “Logic design verification via test generation.” *IEEE TCAD*, vol. 7, no. 1, pp. 138–148, Jan. 1988.

[2] M. Abramovici. *Digital systems testing and testable design*. Computer Science Press, New York, 1990.

[3] A. Aharon et al. “Verification of the IBM RISC System/6000 by dynamic biased pseudo-random test program generator.” *IBM Systems Journal*, pp. 527–538, 1991.

[4] H. Al-Asaad and J. P. Hayes. “Design verification via simulation and automatic test pattern generation.” In *Proc. ICCAD*, 1995, pp. 174–180.

[5] B. Beizer. *Software testing techniques*. Van Nostrand Reinhold, New York, 2nd edition, 1990.

[6] V. Bhagwati and S. Devadas. “Automatic verification of pipelined microprocessors.” In *Proc. DAC*, 1994, pp. 603–608.

[7] D. Bhattacharya and J. P. Hayes. “High-level test generation using bus faults.” In *Dig. FTCS*, 1985, pp. 65–70.

[8] J. Burch and D. L. Dill. “Automatic verification of pipelined microprocessor control.” In *Proc. CAV*, June 1994, pp. 68–80.

[9] A. K. Chandra et al. “AVPGEN - a test generator for architecture verification.” *IEEE Trans. on VLSI*, pp. 188–200, 1995.

[10] K.-T. Cheng. “Gate-level test generation for sequential circuits.” *ACM TODAES*, vol. 1, no. 4, pp. 405–442, 1996.

[11] F. Fallah, S. Devadas, and K. Keutzer. “OCCOM: Efficient computation of observability-based code coverage metric for functional simulation.” In *Proc. DAC*, 1998, pp. 152–157.

[12] A. Gupta, S. Malik, and P. Ashar. “Toward formalizing a validation methodology using simulation coverage.” In *Proc. DAC*, 1997, pp. 740–745.

[13] J. Hennessy and D. Patterson. *Computer Architecture: A quantitative Approach*. Morgan Kaufman Publishers, San Mateo, Calif., 1990.

[14] R. C. Ho and M. A. Horowitz. “Validation coverage analysis for complex digital designs.” In *Proc. ICCAD*, 1996, pp. 146–151.

[15] A. Hosseini, D. Mavroidis, and P. Konas. “Code generation and analysis for the functional verification of microprocessors.” In *Proc. DAC*, 1996, pp. 305–310.

[16] H. Iwashita et al. “Automatic test program generation for pipelined processors.” In *Proc. ICCAD*, 1994, pp. 580–583.

[17] J. Lee and J. H. Patel. “An architectural level test generator based on nonlinear equation solving.” *Journal of Electronic Testing: Theory and Applications*, vol. 4, no. 2, pp. 137–150, 1993.

[18] J. Lee and J. H. Patel. “Architectural level test generation for microprocessors.” *IEEE TCAD*, pp. 1288–1300, 1994.

[19] J. Levitt and K. Olukotun. “Verifying correct pipeline implementation for microprocessors.” In *Proc. ICCAD*, 1997, pp. 162–169.

[20] D. Lewin, D. Lorenz, and S. Ur. “A methodology for processor implementation verification.” In *Proc. FMCAD*, 1996, pp. 126–142.

[21] D. Moundanos, J. A. Abraham, and Y. V. Hoskote. “Abstraction techniques for validation coverage analysis and test generation.” *IEEE Trans. Computers*, vol. 47, no. 1, pp. 2–14, Jan. 1998.

[22] T. Niermann and J. H. Patel. “HITEC: A test generation packaged for sequential circuits.” In *Proc. EDAC*, 1991, pp. 214–218.

[23] S. Taylor et al. “Functional verification of a multiple-issue, out-of-order, superscalar Alpha processor - the DEC Alpha 21264 microprocessor.” In *Proc. DAC*, 1998, pp. 638–643.

[24] D. Van Campenhout et al. “High-level design verification of microprocessors via error modeling.” *ACM TODAES*, vol. 3, no. 4, pp. 581–599, 1998.