High-Level Test Generation for Design Verification of Pipelined Microprocessors[1]

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

Advanced Computer Architecture Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan, Ann Arbor, MI 48109-2122
E-mail: {davidvc, tnm, jhayes}@eecs.umich.edu

ABSTRACT

This paper addresses test generation for design verification of pipelined microprocessors. We describe a high-level model for testing pipelined microprocessors, which exposes high-level knowledge that is useful for verification test generation. We present a three-part test generation algorithm that uses this knowledge: The core part of the algorithm conducts a branch-and-bound search in a transformed state space of the controller. The decision variables of the search represent the essential interaction between concurrent instructions in the pipeline. The size of this transformed search space can be significantly smaller than the original state space of the controller. The second part of the algorithm selects justification and propagation paths in the datapath, which guide the search in the control space. The third part uses discrete relaxation to determine appropriate data values. We have implemented the proposed algorithm and used it to generate verification tests for design errors in the datapath of a representative pipelined microprocessor.

I. 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. Manufacturers of modern microprocessors still rely heavily on simulation-based methods to verify their products.

Traditionally hand-written test cases have been used as a first line of defense against bugs, focusing on basic functionality and important rarely-occurring corner cases. To increase test productivity, sophisticated test generation systems have been developed that are biased towards corner cases [3, 9]. Advances in simulation and emulation technology have enabled the use of other sources of test stimuli such as existing application and system software [20].

These test generation techniques are used in conjunction with coverage metrics to quantify the effectiveness of a verification test suite. The metrics include code coverage metrics from software testing [5], finite-state machine coverage [15], architectural event coverage [27], and observability-based metrics [10]. 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, 18, 28]. 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. Both the implementation and specification of the target design are simulated for that test set. A discrepancy in the simulation outcome indicates an error in the design implementation or in the specification. 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 first present a novel structured high-level model for pipelined processors that applies to a broad class of microprocessors. This model exposes high-level knowledge about microprocessor structure that can be used during test generation. We then present a novel high-level test generation algorithm for pipelined microprocessors that uses the information provided by our model. This algorithm has three parts: Its core part conducts a branch-and-bound search in a transformed state space of the controller. The branching decision variables derive from the essential interaction between concurrent instructions in the pipeline. The transformed search space can be much smaller than the controller's original state space. The second part of the

1. This research is supported by DARPA under Contract No. DABT63-96-C-0074. The results presented herein do not necessarily reflect the position or the policy of the U.S. Government.

algorithm selects appropriate justification and propagation paths in the datapath, which guides the search in the control space. The third part uses discrete relaxation to determine appropriate data values.

We review relevant previous work in Section II. Our high-level model for pipelined processors is presented in Section III. The iterative organization of the proposed high-level test generation algorithm is described in Section IV. The three parts of the algorithm are described in Section V. We present experimental results in Section VI, and give some concluding remarks in Section VII.

II. Related Work

A. Test generation for gate-level sequential circuits

Typical test generators for sequential circuits [2, 26] iteratively apply a test generation algorithm for combinational circuits by using a gate-level iterative logic array (ILA) model of the circuit. Kelsey et al. [19] describe a test generation algorithm for sequential circuits that does not follow the iterative structure of the ILA. For a given fault, an estimate of the test sequence length is computed, and the circuit is unrolled over that many cycles. The PODEM algorithm is applied to the resultant circuit, which is treated as a single combinational circuit. Because this approach only makes decisions on primary inputs and only propagates information forwards, it can result in a more efficient search. On the other hand, the search process is performed on a much larger and deeper circuit than in conventional approaches, hence its efficiency depends critically on the backtracing heuristics used.

Ghosh, Devadas and Newton [12] decompose the test generation problem into three subproblems: combinational test generation, fault-free state justification, and fault-free state differentiation. By performing state justification and differentiation in the fault-free machine their algorithm can re-use a significant amount of computation.

B. High-level test generation

Lee and Patel describe a high-level test generation algorithm for microprocessors [22]. They assume a high-level model of the datapath, and represent the control unit by the set of control behaviors (a sequence of control values presented to the datapath), corresponding to the instructions in the instruction set of the processor. Their algorithm generates instruction and data sequences that apply precomputed test sets to modules under test in the datapath. The key features of this approach are that test generation is split into path selection and value selection phases, and that value selection is performed by discrete relaxation. The modeling of the control unit limits the approach of [22] to non-pipelined processors with a relatively small number of control behaviors.

Iwashita et al. [17] describe a technique for generating instruction sequences to excite given “test cases”, such as hazards, in pipelined processors. Test cases are mapped onto states of a reduced FSM model of the processor. The technique performs implicit enumeration of the reachable states to synthesize the desired test sequences. Some limitations are that the reduced FSM model is derived manually and that no details are given on the effect of the abstraction on the types of test cases that can be handled.

Chandra et al. [9] present a sophisticated code generator for architectural validation of microprocessors. The user provides symbolic instruction graphs together with a set of constraints; these compactly describe a set of instruction sequences that have certain properties. The system expands these templates into test sequences using constraint solvers, an architectural simulator, and biasing techniques. A similar work is discussed in [16]. As these techniques operate on the microarchitectural specification of the design only, they are not suitable for generating tests for structural errors in the implementation.

C. Formal verification

Bhagwati and Devadas [6] describe an automated method to verify pipelined processors with respect to their ISA specification. A mapping between input and output sequences of the implementation and the specification is given. The method assumes that the implementation can be approximated by a k-definite[1] FSM. The equivalence of the two machines is checked by symbolic simulation. The assumptions made about the implementation and the lack of abstraction limit the applicability of this approach.

Burch and Dill [8] propose a method for microprocessor verification based on symbolic simulation and the use of a quantifier-free first-order logic with uninterpreted functions. The method requires manually generated abstract models of both the implementation and the specification in terms of uninterpreted functions. Symbolic simulation of the models is used to construct the next-state functions. The verification problem is turned into checking the equivalence of the next-state functions of implementation and specification.

Levitt and Olukotun [23] develop a methodology for verifying the control logic of pipelined microprocessors. The datapath is modeled using uninterpreted functions. Verification is performed by iteratively merging the two deepest stages of the pipeline. After each step a check is made to see whether the newly obtained pipeline is still equivalent to the previous one. The equivalence is proven automatically using induction on the number of execution cycles. To achieve the high degree of automation, the approach of [23] uses high-level knowledge about the design, such as the design intent of a bypass.

D. Hybrid verification techniques

A class of hybrid verification techniques [11, 13, 15, 24, 25] 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 coverage on the reduced FSM

1. A k-definite FSM is one that can only remember the last k inputs.

model; a typical coverage metric is state transition coverage. 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.

III. Pipelined Processor Model

An important element of microprocessor structure is the distinction between data and control. The merits of treating datapath and control differently have been recognized in many other domains such as high-level synthesis, formal verification, etc. Because it possesses unstructured binary signals, the controller is normally represented at the gate level. The datapath, on the other hand, processes structured data words and so can be represented at a higher level, using high-level, multibit modules and buses. This high-level representations 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 (instruction set architecture) model, from machine state which 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 [14] 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, which 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 are precisely the signals needed to describe essential instruction interaction. Typical examples of tertiary signals in the controller are squash and stall; 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.

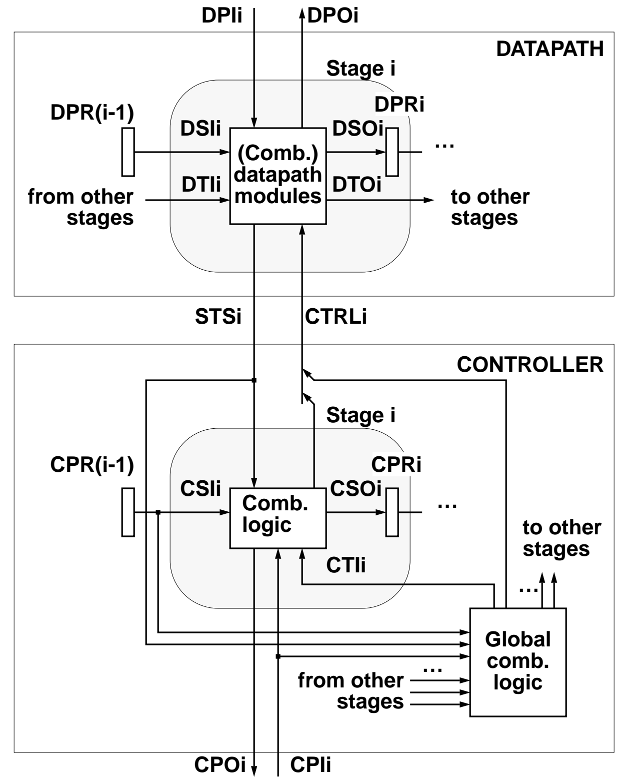

Figure 1. Pipelined microprocessor model.

IV. Pipeframe Model

Conventional 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. $C0$, $C1$ and $C2$ are combinational logic corresponding to the three pipe stages. The global combinational logic CG sources all CPIs and all CSIs. In order not to clutter the figure, the CPI sourced by Ci and the CPOs produced by Ci have been omitted. The iterative logic array model for this circuit is shown in Figure 2b. If PODEM is used as the

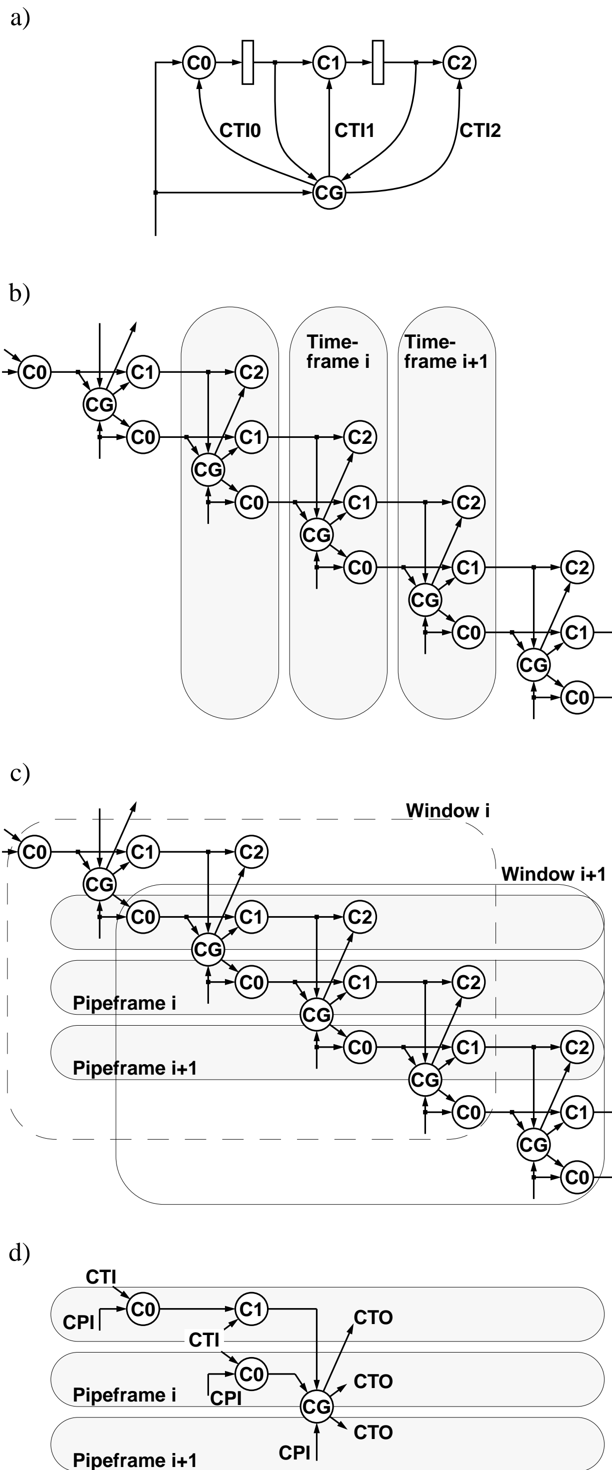

Figure 2. a) Pipelined controller; b) iterative array: conventional organization; c) iterative array: alternative organization; d) composite circuit dealt with in c).

combinational test generation algorithm, the decision variables are the CPIs and the CSIs in each timeframe. The state space to be searched during each iteration is that of the CSIs and CPIs. For the controller of pipelined microprocessors, the number of CSIs (state bits) is typically much larger than the number of CPIs. 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 CPIs, is desirable. When the global control logic *CG* is absent, it is easy to see how this can be accomplished. In this case, the iterative array model 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 invalid (unreachable) states cannot arise as decisions are made only on the CPIs.

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 iterative array is partitioned into pipeframes by cutting the tertiary signals, as shown in Figure 2c. 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. The linking of pipeframes via the tertiary signals is shown in Figure 2d. It can be seen that the tertiary signal $CTI0$ to pipeframe $(i + 2)$ depends on PIs and CTIs to pipeframes i, $(i + 1)$ and $(i + 2)$. (In order not to clutter the figure the indices were omitted.)

Consider an p-stage pipelined controller with a total of n_1 CPIs, n_2 CSIs per pipestage, and n_3 CTIs 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. This can be seen in Figure 2d.

For some pipelined controllers the pipeframe approach does not reduce the search space. This is the case when

TG

```
1.  status = UNDETERMINED
2.  DPTRACE /* derive initial path objectives */
3.  while ( status == UNDETERMINED )
4.      /* imply */
5.      ctrlStatus = CTRLJUST:imply
6.      pathStatus = DPTRACE
7.      valueStatus = DPRELAX
8.      status = Status(ctrlStatus, pathStatus, valueStatus)
9.      if (status == CONFLICT)
10.         UndoImplications(currentDecision)
11.         status = UNDETERMINED
12.         while (NoUntriedValuesLeft(CurrentDecision))
13.             Undo(currentDecision)
14.             if (DecisionStackEmpty)
15.                 status = FAILURE
16.                 break /* out of inner while */
17.             else
18.                 currentDecision = Pop(decisionStack)
19.         if (status == UNDETERMINED)
20.             SelectNextUntried( currentDecision )
21.     else /* status == UNDETERMINED */
22.         if (reset state reached and all objectives satisfied)
23.             status = SUCCESS
24.         else
25.             currentObjective = SelectObjective()
26.             currentDecision = BackTrace( currentObjective)
27.             Push(currentDecision, decisionStack)
```

Figure 3. Overall test generation algorithm.

CSO*i* depends on CSI*i*+1 (referring to Figure 1) for every pipestage. For such circuits, all CSIs are also CTIs, the pipeframe approach reduces to the usual timeframe approach.

V. Test Generation Algorithm

In this section we describe our high-level test generation algorithm for design verification of pipelined microprocessors. It is targeted at errors in the datapath and of the type described in [28]. The algorithm follows the iterative pipeframe organization described in the previous section. The algorithm decomposes the test generation problem into three subproblems:

- *P*1: path selection in the datapath,
- *P*2: value selection in the datapath, and
- *P*3: justification of control signals (controller).

Pseudo-code for the overall algorithm *TG* is shown in Figure 3; the interaction of the three subproblems is shown in Figure 4. Let *DPTRACE*, *DPRELAX*, and *CTRLJUST* be the procedures that solve *P*1, *P*2, and *P*3, respectively. *TG* is mounted on the branch-and-bound search (*CTRLJUST*) for solving *P*3. *DPTRACE* selects justification and propagation paths in the datapath for activating and exposing the error. Part of the solution produced by *DPTRACE* is a set of objectives (s, v), where s is a CTRL signal and $v \in \{0, 1\}$. These objectives are used to guide the search performed by *TG*. *DPRELAX* uses discrete relaxation to determine appropriate data values.

DPTRACE computes an initial path selection and corresponding set of path objectives (step 2). *CTRLJUST* makes decisions on CPI, CTI and STS signals (step 27), guided by the objectives (step 26). These decisions are implied on three fronts. First, they are implied in the controller (step 5) where they affect the CPO, CTO and CTRL signals. Second, *DPTRACE* checks whether the updated CTRL signals are consistent with the current set of justification and propagation paths in the datapath (step 6). If there is consistency, no further action is required. Otherwise *DPTRACE* computes a new set of justification and propagation paths, taking into account the current state of the CTRL lines. The objectives on the CTRL lines are updated accordingly. Only if *DPTRACE* fails to derive a set of justification and propagation paths will *DPTRACE* cause *TG* to backtrack. The third aspect of implication involves invoking *DPRELAX* (step 7) to compute data values. Failure to converge will cause *TG* to backtrack. Steps 9-20 are the usual actions for backtracking. Step 22 checks

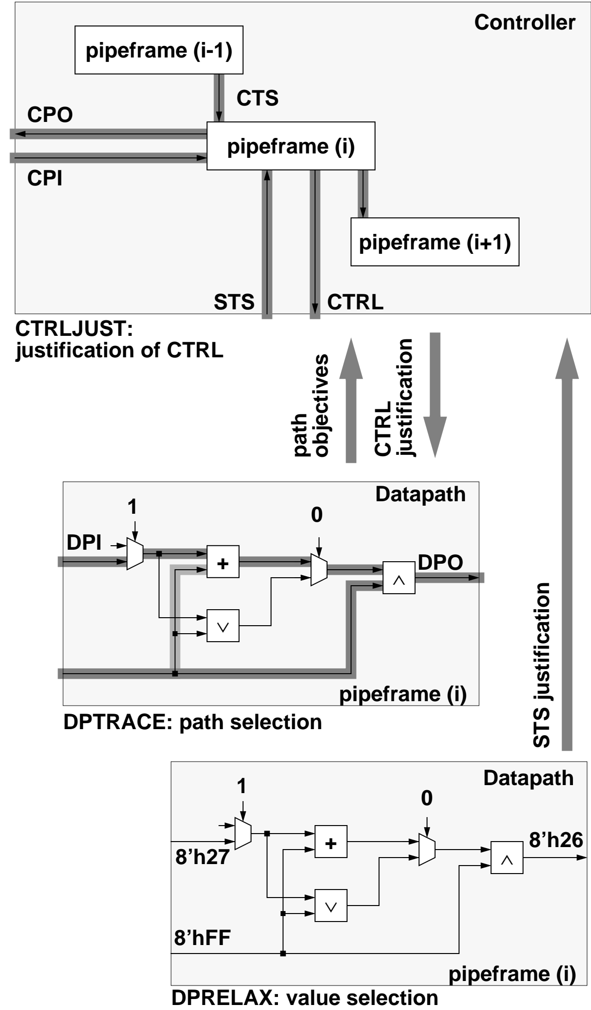

Figure 4. The three parts of the test generation algorithm and their interactions.

whether the reset state has been reached. If so, and if all objectives are satisfied, we return successfully with a test. The three subalgorithms are described in the remainder of this section.

A. DPTRACE: path selection in datapath

The task of the path selection algorithm DPTRACE is to determine 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. Value selection is delegated to DPRELAX. This divide-and-conquer approach reduces the problem size significantly, but may fail to find a solution even if the problem is feasible.

First we define a circuit representation suitable for path selection. Datapath modules are classified into the following three categories:

- ADD class
- AND class
- MUX class

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 this class include the adder, the subtractor, and the X(N)OR word gate. Predicate modules, which take two *n*-bit inputs *A* and *B* and produce a single bit output *Y* = *A* <*op*> *B*, where <*op*> ∈ {=, ≠, <, ≤, >, ≥, ADDOVF, SUBOVF}, are also placed in the ADD class. The latter two modules compute overflow for signed addition and subtraction.

Modules in the AND class have one data output, and one or more data inputs. 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 this class are word gates such as (N)AND, (N)OR, and shift modules.

Modules in the MUX class have one data output, one or more data inputs, and one or more control inputs. The control inputs 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. This class contains modules such as multiplexers and tristate buffers.

More complex modules such as ALUs are represented as a composition of the modules listed above. For nets with multiple fanout, only one fanout can be justified by controlling the stem. This reflects that the values to be justified on two fanouts of the same stem may be different, but during path selection this information is unavailable.

Formulation

Controllability information is represented by a symbolic value attributed to each port (terminal of a module) in the circuit. The attribute, the *C*-state of a port, assumes values from the set {*C*1, *C*2, *C*3, *C*4}. The interpretation of these values are as follows:

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

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

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

Controllability and observability information is propagated forwards and backwards, respectively, according to propagation tables. Propagation tables for a representative of each module class are given in Figure 5. The first table expresses the *C*-state of the output port *y* of a two-input ADD module in terms of the *C*-state of its input ports x_1 and x_2. The second table expresses the *O*-state of x_1 in

ADD2

***C*(*y*)**		***C*(*x*2)**			
		*C*1	*C*2	*C*3	*C*4
	*C*1	*C*1	*C*1	*C*1	*C*1
	*C*2	*C*1	*C*2	*C*2	*C*1
***C*(*x*1)**	*C*3	*C*1	*C*2	*C*3	*C*4
	*C*4	*C*1	*C*1	*C*4	*C*4

ADD2

***O*(*x*1)**		***O*(*y*)**		
		*O*1	*O*2	*O*3
	*C*1	*O*1	*O*2	*O*1
	*C*2	*O*1	*O*2	*O*1
***C*(*x*2)**	*C*3	*O*1	*O*2	*O*3
	*C*4	*O*1	*O*2	*O*3

AND2

***C*(*y*)**		***C*(*x*2)**			
		*C*1	*C*2	*C*3	*C*4
	*C*1	*C*1	*C*2	*C*2	*C*1
	*C*2	*C*2	*C*2	*C*2	*C*2
***C*(*x*1)**	*C*3	*C*2	*C*2	*C*3	*C*3
	*C*4	*C*1	*C*2	*C*3	*C*4

AND2

***O*(*x*1)**		***O*(*y*)**		
		*O*1	*O*2	*O*3
	*C*1	*O*1	*O*2	*O*1
	*C*2	*O*2	*O*2	*O*2
***C*(*x*2)**	*C*3	*O*2	*O*2	*O*2
	*C*4	*O*1	*O*2	*O*3

MUX2 (*s*=1 selects *x*2)

***C*(*y*)**		
s	*u*	*C*2 if *C*(*x*1),*C*(*x*2) ∈ {*C*2,*C*3} *C*1 otherwise
	0	*C*(*x*1)
	1	C(*x*2)

MUX2 (*s*=1 selects *x*2)

***O*(*x*1)**		***O*(*y*)**		
		*O*1	*O*2	*O*3
	u	*O*1	*O*2	*O*1
s	0	*O*1	*O*2	*O*3
	1	*O*2	*O*2	*O*2

Figure 5. *C*- and *O*-propagation tables.

terms of the *O*-state of y and the *C*-state of x_2. The figure also shows similar tables for a two-input AND word gate and a two-input multiplexer.

We formulate the path selection problem as a search problem with two types of decision variables:

- CTRLs: control variables
- FOs: fanout-select variables

CTRL variables are associated with the CTRL signals to the datapath and assume three values {0, 1, u} (u: unassigned). FO variables are associated with nets that have multiple fanout; they assume values from {1,…, n, u}, where n is the number of fanouts. An FO variable indicates which fanout uses the stem for its justification; the other fanouts cannot be controlled.

The path selection problem is as follows: Given is a J-frontier = $\{(p_i, cp_i) \mid i=1\ldots n\}$ where p_i is a data-port, and $cp_i \in \{C3, C4\}$; given is an E-frontier[1] = $\{p_i \mid i=1\ldots m\}$; given is a partial assignment to the CTRL variables; determine a partial assignment to the decision variables such that the *C*-state of every line in the J-frontier is justified to the specified value, and that at the *O*-state of at least one line in the E-frontier is justified to *O*3.

We have adapted PODEM for the path selection problem. We have adapted gate-level controllability and observability measures [2] for our problem.

B. DPRELAX: value selection in datapath

The task of the value selection algorithm is to determine values for DPI that expose the error effect and justify any STS signals assigned by the *CTRLJUST.* As in path selection, the problem is solved on a per pipeframe basis.

More precisely, the value selection problem for a single pipeframe is as follows: Given a partial assignment to the CTRL, DPI, and DTI signals, and a set of (s, v) pairs that need to be justified, where s is a STS or DTO signal, and v is an integer value, determine a partial assignment to the DPI and DTI signals that exposes the error effect at a DPO or DTO and justifies every given (s, v).

This problem can be formulated as that of finding a solution to a system of non-linear equations [21]. For special cases, such as that of datapaths containing only linear modules, efficient deterministic methods can be devised to solve the system. However such techniques are not applicable to the non-linear systems that result in most practical cases. Lee and Patel [21] suggested the use of discrete relaxation in the context of high-level test generation for physical fault testing. The main advantages of this technique are its ability to deal with any type of combinational datapath modules and its simplicity. A disadvantage is that it is not a complete method: it cannot prove that the system has no solutions, and may fail to find a solution even if there is one. A key observation is that during path selection, appropriate justification and propagation paths are selected so that the system to be solved during value selection is likely to be underdetermined, in which case discrete relaxation is likely to converge quickly.

In our discrete relaxation algorithm, each net in the datapath is characterized by two pairs of variables, one corresponding to the error-free circuit, the other to the erroneous circuit. Each pair consists of an integer in the range specified by the bit-width of the net, and a type which is in the range {*unassigned*, *determined*, *fixed*}. The algorithm iteratively re-evaluates the modules in the circuit until a consistent assignment is obtained or until a maximum iteration count is exceeded. The mechanism is event-driven. An event is associated with each terminal of a net when the value of that net is changed. An event is processed by re-evaluating the module to which the triggering terminal belongs. If the current values of the nets connected to the module are consistent with the module's functionality, no further action is required. Otherwise the values of one or more nets connected to the module are changed in order to make them consistent. New events are generated at all terminals (except those belonging to the module that is being processed) of every net whose value has been changed.

The choice of which net to update and what value to assign can, in principal, be random, but it strongly influences convergence. We implemented a number of heuristics whose goal is to try to exercise all possible modes of event propagation and to aid convergence.

C. CTRLJUST: justification of CTRL signals in controller

Given is a set of objectives (c_i, v_i) where c_i is a CTRL signal and $v_i \in \{0.1\}$, and a J-frontier = $\{(t_j,v_j)\}$ where t_j is a CTI signal. The justification problem is that of determining an input sequence (to be applied to CPI, STS) that starts from the controller's reset state, and satisfies the given objectives and justifies the J-frontier.

CTRLJUST is a PODEM-based algorithm with decision variables the CPI, CTI and STS signals. The search is guided by path objectives on the CTRL lines produced by DPTRACE. The objectives are backtraced to generate decisions. Decisions on CTI signals need to be justified and are therefore added to the J-frontier. If a decision concerns a STS signal, that STS signal needs to be justified by the datapath, and is therefore added to the set of signals to be justified by DPTRACE. After a decision is made its implications are determined. As explained above, the overall test generation algorithm *TG* is mounted on *CTRLJUST.* In fact what remains of *TG* (Figure 3) after deleting step 2, 6 and 7, is *CTRLJUST.*

VI. Experiments

We have built a prototype implementation of the proposed test generation algorithm. It encompasses 22K lines of C-code, excluding the Verilog parser and the BDD package. We are using a version of the DLX microprocessor [14] as a preliminary test vehicle. This design implements 44 instructions, has a five-stage pipeline and branch prediction logic, and consists of 1552 lines of structural

1. In design verification we use the term *error* to differentiate from the term *fault* used in physical fault testing; hence *E*-frontier instead of F-frontier.

Table 1. Test generation for bus SSL errors in execute, memory and write-back stages of DLX

No. of errors	298
No. of errors detected	252
No. of errors aborted	46
Average test sequence length	6.2
No. of backtracks (detected errors only)	50
CPU time [minutes]	36

Verilog code, excluding the models for library modules such as adders and register-files. The datapath has 512 bits of state, not including those in the register file; the controller has 96 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 96 to 43 compared to the conventional timeframe approach. Solving data values by discrete relaxation allows us to avoid searching the huge data state space.

We targeted our test generation system at all bus single stuck line (bus SSL) errors [7] in the 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 [28], 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 298 errors were targeted; test generation succeeded for 85% of these errors. The average length of the test sequences is slightly more than 6 instructions. Typical sequences consist of a few non-trivial instructions followed by a sequence of NOP instructions. The overall algorithm performed only 50 backtracks for the successful errors. We are currently investigating why test generation failed for the aborted errors, and expect to achieve higher coverage in the near future. It should be noted that no error simulation was used in this preliminary implementation, and that much re-use of work in the algorithm has not yet been exploited. Therefore, we can expect that run times will significantly improve as these issues are addressed.

VII. Conclusions

We are developing a system for automatically generating 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. It exploits high-level knowledge about the operation of pipelines which is captured by our microprocessor model. As the analysis shows, the pipeframe approach can significantly reduce the search space. We have also formulated the path selection problem such that it can be solved by a variety of branch-and-bound algorithms. We are building software that implements our test generation algorithm and used it to generate verification tests for a pipelined microprocessor. Our preliminary experimental results demonstrate the feasibility scalability and of the approach.

References

[1] M. S. Abadir, J. Ferguson, and T. E. Kirkland. "Logic design verification via test generation." *IEEE Trans. CAD*, vol. 7, no. 1, pp. 138–148, 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. Int. Conf. CAD*, 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 *Computer-Aided Verification*, 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] 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.

[11] D. Geist et al. "Coverage-directed test generation using symbolic techniques." In *Proc. Int. Conf. FMCAD*, 1996, pp. 143–158.

[12] A. Ghosh, S. Devadas, and A. R. Newton. "Test generation and verification for higly sequential circuits." *IEEE Trans. CAD*, vol. 10, no. 5, pp. 652–667, 1991.

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

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

[15] R. C. Ho and M. A. Horowitz. "Validation coverage analysis for complex digital designs." In *Proc. Int. Conf. CAD*, 1996, pp. 146–151.

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

[17] H. Iwashita et al. "Automatic test program generation for pipelined processors." In *Proc. Int. Conf. CAD*, 1994, pp. 580–583.

[18] S. Kang and S. A. Szygenda. "The simulation automation system SAS; concepts, implementations, and results." *IEEE Trans. on VLSI*, pp. 89–99, 1994.

[19] T. P. Kelsey, K. K. Saluja, and S. Y. Lee. "An efficient algorithm for sequential circuit test generation." *IEEE Trans. Computers*, pp. 1361–1371, 1993.

[20] J. Kumar. "Prototyping the M68060 for concurrent verification." *IEEE Design & Test of Computers*, pp. 34–41, 1997.

[21] J. Lee and J. H. Patel. "A signal-driven discrete relaxation technique for architectural level test generation." In *Proc. Int. Conf. CAD*, 1991, pp. 458–461.

[22] J. Lee and J. H. Patel. "Architectural level test generation for microprocessors." *IEEE Trans. CAD*, pp. 1288–1300, 1994.

[23] J. Levitt and K. Olukotun. "Verifying correct pipeline implementation for microprocessors." In *Proc. Int. Conf. CAD*, 1997, pp. 162–169.

[24] D. Lewin, D. Lorenz, and S. Ur. "A methodology for processor implementation verification." In *Proc. Int. Conf. FMCAD*, 1996, pp. 126–142.

[25] 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, 1998.

[26] T. Niermann and J. H. Patel. "HITEC: A test generation packaged for sequential circuits." In *Proc. European DAC*, 1991, pp. 214–218.

[27] 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.

[28] D. Van Campenhout et al. "High-level design verification of microprocessors via error modeling." To appear in *ACM TODAES*, vol. 3, no. 4, 1998.