

ADL-driven Test Pattern Generation for Functional Verification of Embedded Processors

Abstract

With increasing complexity of modern microprocessors, the task of verification is becoming an imposing challenge to tackle. To cope with the complexity, verification in early design phase of the processor is being proposed in literature [1]. Architecture Description Languages (ADLs) offer a convenient high-level modelling platform for advanced embedded processors. From an ADL description, the software tools e.g. simulator, C-compiler, assembler as well as the processor implementation in Register Transfer Level (RTL) can be automatically generated. Clearly, ADL-driven verification offers an approach to integrate verification in early phase of microprocessor design. A major requirement in such verification approaches is a compact test-pattern yielding high coverage. In this paper, this problem is addressed. We have proposed a novel backtracking algorithm for cycle-accurate pipelined processors to obtain the test-pattern, which explicitly targets the coverage of each conditional block of the processor description. This results in a high-coverage test pattern. The effectiveness of the approach is demonstrated with a case study of two simple scalar pipelined RISC processors.

1. Introduction

The growing design complexity coupled with the short time-to-market have increased the importance of verification in the design cycle of today's embedded processors considerably. The designer needs to deliver the optimum performance in a short time without compromising the verification issues. To manage the imposing design complexity, nowadays the processor design is performed on a high level of abstraction, thereby allowing fast design space exploration for target-specific architectures. Architecture Description Languages (ADLs), allowing efficient high-level processor design, is getting prominence in industry [2][3] and academia [4]. From an ADL description, the software tools e.g. simulator, HLL compiler, assembler as well as the processor implementation in RTL can be automatically generated.

With the increasing design abstraction, verification is receiving continuous attention from the Electronic System Level (ESL) design community. Several research initiatives are taken in order to integrate the verification flow with the high-level design flow. Notable endeavours are high-level simulation [5], sequential logic equivalence checking [6], assertion-based semi-formal verification [7], property-driven verification [8] etc. For two of the major verification approaches, namely assertion-based verification and simulation-based verification, a key component of the verification flow is a test-pattern to drive the verification. Here we focus on functional test pattern i.e. test patterns to trigger the functional errors in the design.

In the perspective of processor design, the functional test pat-

terns are a set of processor instructions. In general, the test-pattern needs to be small (to reduce simulation time) as well as sufficient to cover the complete design. The term coverage plays an important role to justify the quality of the test-pattern. Different coverage metrics and corresponding test pattern generation to achieve a high coverage have been proposed in the literature. In the following paragraphs existing approaches of automatic test pattern generation from high-level specification is studied. The corresponding coverage metrics are mentioned, too.

Functional test generation from high-level processor descriptions is a well-studied field. Several attempts to automate the test generation from Register-Transfer-Level (RTL) or higher-abstraction of a processor have been made. These approaches can be broadly classified as *coverage-driven* or *analytical* approach.

Coverage-driven approaches strongly rely on the feedback of the coverage to the test-generation mechanism. Exemplarily, Corno et al [9] proposed a genetic algorithm-based framework to build up the test pattern on the basis of coverage feedback. This approach demonstrated 100% RTL statement coverage for a sparc-compatible processor. However, the genetic algorithm required simulation of up to 7.3 million instructions to obtain the compact instruction-set showing 100% RTL statement coverage. Clearly, the runtime of the test generation is a big drawback here. Fast convergence of the algorithm is dependent on the choice of parameters like mutation, cross-over and effective fitness criteria.

In analytical approaches, the instructions which excite the processor resources and/or operators are chosen systematically [3]. The operands are selected to reflect different scenarios like data hazards or control hazards in a pseudo-random manner. Often, manual intervention or good heuristics play an important role in determining the quality of the test-patterns. IBM Genesys Test Generation framework [10] is one such example. In this test generation process, the verification engineer guides the generation process via a Graphical-User-Interface. IBM Genesys framework is shown to be quite useful to reveal corner case bugs. Full coverage of the RTL implementation of the processor is not explicitly targeted. The analytical approach presented by Luethje [11] is one prominent approach to achieve complete statement coverage from an high-level description. Luethje performed an abstract execution of the conditional blocks of an ADL description and showed full coverage of the ADL statements. However, his algorithm works for instruction-accurate processor models only i.e. the algorithm is not scalable to pipelining effects. Another notable effort to automate test generation from high-level description is presented by Mishra et al [12]. Mishra et al proposed a new functional coverage metric on the basis of a graph model of the processor. The functional coverage metric include scenarios like pipeline execution, register read/write, operation execution etc. A drawback of this approach is that, it considers the operations *atomically* as nodes of the graph and the input/output operands are loaded/stored by specific processor instructions, which are known in advance. If an operation contains deeply nested data-

flow within it, this approach may not cover the execution of all conditions within it. To the best of our knowledge, there is no analytical approach to automatically obtain the coverage of all conditional blocks for a cycle-accurate processor implementation. The biggest hindrance to such an attempt is the complexity of the processor itself. In [13], a HDL-satisfiability checker algorithm is developed to determine stimuli for exciting selected paths in an HDL model. This is close to the solution approach presented in this paper. In our case, the problem is attacked with a backtracking algorithm aided with some heuristics. Due to the overall architectural knowledge available from ADL description, the backtracking algorithm turned out to be efficient in runtime.

In the work presented in this paper, we start from an existing test generation tool capable of generating constrained random test-cases. This framework, with able guidance, is shown to achieve high RTL statement coverage for pipelined processors [14]. Similar constraint-driven test-generation frameworks are used heavily in industry and academia. In this paper, our contribution is to propose an analytical backtracking-based algorithm for extracting constraints from an ADL description. These constraints, when fed to the existing test generation tool, can generate test-case achieving high RTL statement coverage. The complete tool is based on a high-level ADL-driven processor design framework thereby aiding early verification.

The contribution of this paper is to present:

- An ADL-based automatic tool-flow to achieve high RTL statement coverage.
- A novel backtracking algorithm for functional test generation of pipelined processors.

This paper is organized as follows: section 2 discusses the basic features of ADL necessary for understanding the rest of this paper. The information necessary for the test generation is extracted from the ADL, which is elaborated in this section. Section 3 provides a short overview of the ADL-driven constrained random test generation. Section 4 and section 5 describes the complete verification flow in detail, including the constraint extraction algorithm. Section 6 elaborates and analyzes our case study. We conclude with the summary and outlook.

2. ADL : An Overview

In this section, a brief overview of the architecture description language LISA is provided. Only those language elements, which are relevant for this paper are covered here.

2.1 LISA Operation Graph

In LISA, an *operation* is the central element to describe the timing and the behavior of a processor instruction. The instruction may be split among several LISA operations. The resources (registers, memories, pins etc.) are declared globally in the *resource section*, which can be accessed from any LISA operation.

The LISA description is based on the principle that a specific common behavior or common instruction encoding is described in a single operation whereas the specialized behavior or encoding is implemented in its *child* operations. With this principle, LISA operations are basically organized as an n -ary tree. However, specialized operations may be referred to by more than one *parent* operation. The complete structure is a Directed Acyclic Graph (DAG) $\mathcal{D} = \langle V, E \rangle$. V represents the set of LISA operations, E the graph edges as set of child-parent relations. These relations represent *Activations*, which refer to the execution of another LISA operation. Figure 1 gives an example of a LISA operation DAG. As shown, the operations can be distributed over several pipeline stages.

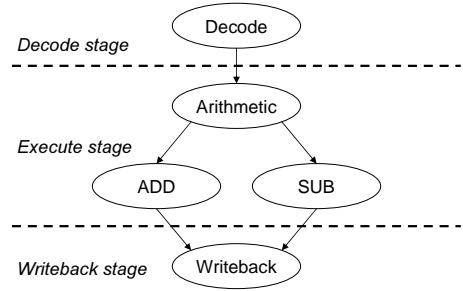


Figure 1. LISA Operation DAG Example

2.2 Instruction Coding Description

The instruction encoding of a LISA operation is described as a sequence of several *coding fields*. Each coding field is either a terminal bit sequence with “0”, “1”, “don’t care”(X) bits or a non-terminal bit sequence referring to the coding field of a child LISA operation.

2.3 Activations

A LISA operation can *activate* other operations in the same or a latter pipeline stage. In either case, the child operation may be activated *directly* or via a *group*. A *group* collects several LISA operations, with the elements being mutually exclusive. The elements are distinguished by a distinct binary coding.

2.4 Behavior Description

The behavior description of a LISA operation corresponds to the datapath of the ASIP. The behavior description is a non-formalized element of the LISA language (contrary to formalized elements like *coding*, *activation* etc.), where plain C code can be used. Resources such as registers, memories, signals and pins as well as coding elements can be accessed in the same way as ordinary variables.

2.5 Data Flow Graph (DFG) Representation

The behavior section of a LISA operation is converted into a pure, directed Data Flow Graph (DFG). The graph vertices of $G_{DFG} = \langle V_{op}, E_{ic} \rangle$ are the basic *operators* for data manipulation e.g. additions while edges represent the flow of unchanged data in form of *interconnections* of inputs and outputs.

Operators: The following list summarizes the basic classes of operators represented by graph vertices.

- Commutative n -ary Operator (ALU_OP), $n \geq 2$
- Noncommutative n -ary Operator (ALU_OP), $n \geq 1$
- Read/Write Access to Registers (RESOURCE, PIPE_RESOURCE) and Memories
- Decoding and Control Signal (DECODE, FLUSH, STALL)
- Multiplexer

Note that, unary operators are treated as a special case of Non-commutative n -ary operator.

Interconnections: Interconnections represent the data flow on symbol-level representations. The information about the data type transferred is given by an annotation to the interconnection.

3. Constrained Random Test Pattern Generation from ADL

The ADL-driven constrained random test generation engine accepts an instruction-grammar (which is generated automatically

from the ADL) and generates test-cases on the basis of user constraints in an GUI-based environment. In this context, we present the instruction-grammar (as defined in [?]) for a detailed understanding.

Instruction-Grammar: The instruction grammar represents the valid instructions in Backus-Naur Form (BNF) grammar. Table 1 shows an exemplary *instruction grammar*. For this example, the instruction word width is 32 bit and there are 16 available registers indexed by `src_reg` and `dst_reg`.

<code>insn</code>	:	<code>add dst_reg src_reg src_reg</code>
		<code> sub dst_reg src_reg src_reg nop</code>
<code>add</code>	:	<code>0000 0001</code>
<code>sub</code>	:	<code>0000 0010</code>
<code>src_reg</code>	:	<code>0000 xxxx</code>
<code>dst_reg</code>	:	<code>0000 xxxx</code>
<code>nop</code>	:	<code>0000 0000 0000 0000 0000 0000 0000 0000</code>

Table 1. Exemplary Instruction Grammar

3.1 Test Generation Engine (TGE)

There are various features available in the TGE, for fine-tuning the test-cases so that the processor properties can be conclusively tested. In the following, the relevant features are listed.

- *Instruction Register Biasing:* Using the TGE, it is possible to bias the instruction register with some definite immediate values.
- *(Un)Selection of Node:* A test-sequence containing specific node(s) of the *instruction grammar* can be generated or prevented from being generated. This node may represent a LISA operation, a register or an immediate.
- *Branch/Jump Address Biasing:* In order to prevent infinite loops or jumping outside program location, specific constraints can be provided to the TGE to bias the jump/branch address.

For automatic generation of the test-pattern, the instruction grammar is loaded into an internal DAG. Test patterns are generated by traversal of this DAG. The nodes and edges of the data-structure is appropriately tagged with the user-defined constraints. For example, to have an instruction with high occurrence frequency, the edge leading to the instruction is traversed with higher probability than the other edges.

4. The Tool Flow

In order to attain full statement-coverage in the RTL, it is a prudent idea to target the coverage of the conditional blocks in it. These conditional blocks in the RTL is represented as the muxes in the corresponding data-flow graph of the ADL behavior section. So our problem reduces to generating test-patterns which verifies individual muxes, and verification of a mux is possible if it is excited with all the different control-sequences in the muxes. In the following subsections we first clearly state the inputs to our developed tool and outputs we target to deliver.

4.1 Inputs to the Coverage Constraint Generator (CCG)

The inputs to the CCG are the following (refer figure 2).

1. The **DFG**(G_{DFG}) described in section 2.5.
 2. The **LISA Operation DAG**(\mathcal{D}) described in section 2.1.
- The output of the CCG is the *constraint* which is a set of LISA operations and/or instruction-register biases with the desired immediate value to cover a particular mux in DFG.

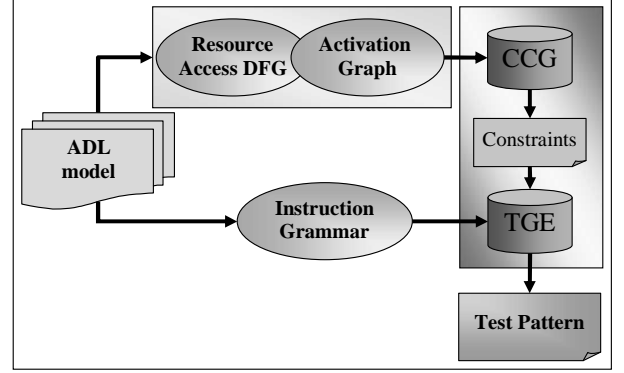


Figure 2. The Tool Flow

4.2 Inputs to the TGE

The next part of the tool flow has the TGE which generates the actual binary test-patterns for the RTL simulation. For this task it takes the *instruction-grammar* and the *constraints* generated by the CCG. In particular the *Instruction Register Biasing* and the *(Un)Selection of Node* feature of TGE are used.

5. The Algorithm

In this section, we describe the algorithm used in the CCG. The algorithm is backtracking in nature. Apparently, a SAT-solver could have replaced the backtracking algorithm. But the reasons for not using it are as follows,

1. The different heuristics that we have devised add to the efficiency of the methodology. The run-time of our algorithm and the coverage results obtained (in section 6) show that the heuristics are appropriate for the automatic test generation from the ADL-description of the embedded processors.
2. The mapping of the behavioral constructs of an ADL in a SAT-solver is also a non-trivial task.

In the backtracking algorithm, since we are targeting the conditional block coverage, the nodes with label MUX are targeted. Here the term *pathway* is defined. A *pathway* (Π), is a sequence of vertices (v_0, v_1, \dots, v_{n-1}), where v_0 and v_{n-1} are the start-node and last-node respectively ($v_i \in V(G_{DFG}), \forall i$).

$$\Pi : v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_i \rightarrow \dots \rightarrow v_{n-1}$$

A pathway with respect to a particular vertex (v_i) is Π_{v_i} , i.e. $\Pi_{v_i} : v_0 \rightarrow \dots \rightarrow v_i$. Moreover, the labels of start-node and last-node are as follows.

$$label(v_0) = \text{MUX}; \quad label(v_{n-1}) = \{\text{DECODE}, \text{CONST}\}$$

The algorithm has three interweaved parts in it. Before describing it formally the algorithm is explained intuitively referring to figure 3.

5.1 The Intuitive Idea

1. Backtracking: In the first part we backtrack in the G_{DFG} . Referring to figure 3, we start the algorithm by initializing the bottom-most MUX node with its control value '1'. This '1' in the MUX control demands '1' from the AND-labeled vertex. Here we define the term *demand-value* (say $demand(v_i)$) and *node-value* (say $nodeval(v_i)$). Demand-value is the set of expected values for a node to satisfy the current pathway upto vertex v_i (Π_{v_i}), and node-value denotes the temporary value which is set in a given node for

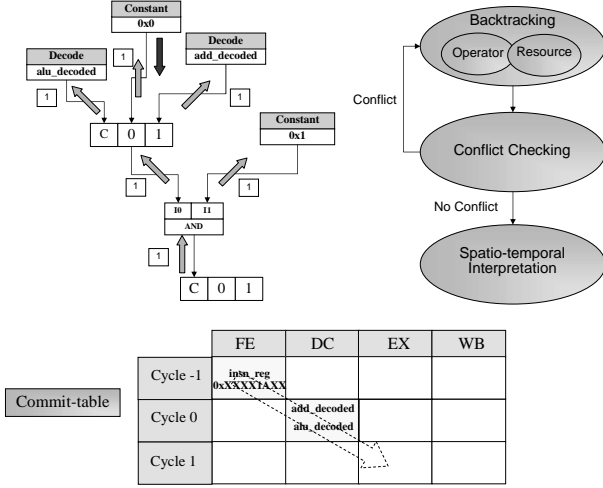


Figure 3. The Overall Algorithm

the path-satisfaction. Formally,

$$demand(v_i) = \{d_{v_i} : d_{v_i} \models nodeval(\Pi_{v_i-1})\}$$

where, $d_{v_i} \models nodeval(\Pi_{v_i-1})$ represents that d_{v_i} is consistent with the node-values assigned to the sequence of vertices in Π_{v_i-1} . In the fig 3, $demand(AND\text{-labeled vertex}) = 1$ (since '1' in the output of AND-gate is necessary for the control logic value of the MUX being '1'). This node temporarily assumes '1' in its output (hence, $nodeval = 1$) and in its turn propagated '1' to both the operands of this node. In this way, the required value is back-propagated upwards until some node with label DECODE or CONST is reached e.g. here the back-propagation stopped after reaching the following nodes namely, *alu_decoded*(DECODE), *add_decoded*(DECODE) and the CONST-labeled nodes.

2. Conflict Checking: It is not always the case that the back-propagation will be smooth throughout. For example in figure 3, the demand-value to the CONST node (with constant value '0') is '1', which is not feasible. Sometimes a node is reached through other path earlier and assigned a value which is conflicting with the value assignment in the current path. We denote such cases as *node-value conflicts*. Formally, node-value conflict occurs, whenever in a particular vertex, either of the following is true,

- $nodeval(v_i) \notin demand(v_i)$. This happens when the node-value is already assigned by some other path Π'_{v_i} .
- There exists no d_{v_i} ($d_{v_i} \in demand(v_i)$), such that d_{v_i} is a feasible node-value in vertex v_i . This situation arrived in CONST node of the figure 3.

In case of conflict in the current pathway, other possible options are explored by backtracking. It is obvious that such backtracking might prove to be costly in time-complexity. To speed up the runtime of the algorithm some efficient heuristics have been devised.

Another possible conflict is *mutual exclusion conflict* among the operations. Consider the LISA Operation DAG in figure 1. There, the two operations ADD and SUB belong to the same LISA group, signifying that those are mutually exclusive. Formally,

- any two operations which have same parent in the activation graph are mutually exclusive. i.e. $(parent(op_i) == parent(op_j)) \Rightarrow mutex(op_i, op_j)$
- any two operations whose parents are mutually exclusive are also mutually exclusive. i.e. $mutex(parent(op_i), parent(op_j)) \Rightarrow mutex(op_i, op_j)$

During backtracking, it must be ensured that no two such exclusive operations are required to be executed at the same cycle.

3. Spatio-temporal Interpretation: Once there is no conflict in the pathway, i.e. the control value of a particular mux becomes achievable through identification of some constraints or by setting certain resource biases, we call this pathway as '*conflictless-pathway*'. This '*conflictless-pathway*' is a set of DECODE and CONST vertices in some particular stage and cycle. It is important to interpret a '*conflictless-pathway*' in a suitable form so that the final test-patterns can be easily generated using the TGE. We maintain a dedicated data-structure - '*commit-table*' (in figure 3) for this interpretation. This commit-table supplies the required constraints for the TGE as shown in the figure 2. The commit-table consists of rows (*cycles*) and columns (*stages*). Each cell in the commit-table maintains a list of DECODE, CONST nodes and instruction-register biases. The DECODE nodes are converted to constraints for generating appropriate LISA operations and the node-value of CONST node is assumed to be instruction-register bias.

The location of the vertices in the commit-table is decided by the *spatio-temporality* of the vertices. In the graph G_{DFG} , each vertex is associated with a *spatio-temporality* in the current pathway. By spatio-temporality, we mean the desired cycle and stage of occurrence (i.e. the position in the commit-table) of a given node.

Each vertex has a definite stage which determines the stage in the commit-table too. The cycle of initial MUX-node is 0. For pipeline registers, the cycle is decided such that the data is available through propagation without being overwritten in the pipeline. In case of the non-pipeline registers the overwriting can be easily avoided by inserting 'nop' instructions once the desired data is written on the register. For other kinds of nodes, the cycle is same as the the cycle of the previous node in the current pathway. The algorithm is formally stated in algorithm 5.3.

In our example, the commit-table suggests the following constraint *alu_decoded*, *add_decoded* (stage: DC, cycle: 0) and an instruction register bias (stage: FE, cycle: -1). The complete set represents one single instruction, as indicated by the diagonal arrow in the commit-table (in figure 3).

5.2 Some Special Cases

In order to pace up the back-propagation algorithm, some heuristics are employed. In the following subsections we have described some of those heuristics through examples.

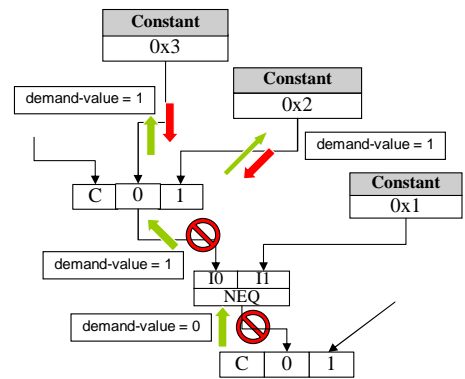


Figure 4. The block-path heuristic

Block Path Heuristic: While executing the algorithm described in section 5, it is important to minimize the back-tracking. Sometimes, it becomes evident from the scenario that no further back-tracking can lead to a feasible $nodeval(v_i)$ which satisfies Π_{v_i} . A typical scenario is shown in the following example. In figure 4 it

can be observed that the demand-value ('0b1') propagated from the NEQ node to the MUX cannot be achieved since both the operands of the MUX is of type CONST having other values ('0b3' and '0b2'). Unless, we block the current pathway, the NEQ-node will continue sending other infeasible demand-values. So the path is blocked (or 'flagged') so that no more unnecessary iteration takes place in this pathway. Formally, the condition for blocking a path at node v_i is following,

$$(label(v_i) = \text{CONST}) \wedge (nodeval(v_i) \text{ does not satisfy } \Pi_{v_i})$$

Constant Lookahead Heuristic: Sometimes, it becomes totally unnecessary to iterate over a pathway with all the possibilities. In the G_{DFG} , the ALU operator nodes often have constant nodes as one of the operands. In this heuristic, the constant is *lookahead*-ed in order to determine a suitable demand-value. For example, a NEQ operator has the demand-value '0b0' and it has one constant operand with value '0b0'. Hence, it is clear that '0b0' needs to be sent in the pathway of the other operand.

5.3 The Formal Algorithm

In this subsection, the algorithm is stated formally. The BACKTRACK algorithm given below takes current node($curr_v$), previous node ($prev_v$) the demand-value ($demand_value$) as the inputs. The term *spatio-temporality* of a node is used here. Spatio-temporality contains the information regarding the cycle of occurrence and the stage to which the node belongs to. This spatio-temporality needs to be computed for every node (appearing in Π_{v_i} for assigning its position in the commit-table. Moreover, to find the write access to a resource, it is necessary to find out the corresponding list of RESOURCE-nodes ($write_resource_list_{curr_v}$). This list needs to be ordered according to a priority criterion. This formal algorithm is presented in algorithm 5.2 below.

ALGO. 5.1. Backtrack

```

Backtrack( $curr\_v$ ,  $prev\_v$ ,  $demand\_value$ )
begin
1:  $curr\_spatio\_temporality \leftarrow \text{AdjustSpatioTemporality}(prev\_spatio\_temporality)$ .
2: CheckConflict( $curr\_v$ .node  $v$ value,  $demand\_v$ value)
   // required for checking node-value conflict
   2.1: if(conflict) return conflict.
3: If( $label(curr\_v) = \text{DECODE}$ )
   3.1: Check for mutual-exclusion conflict in diagonal cells containing the  $curr\_v$ .
   3.2: If(no conflict)
     3.2.1:  $nodeval(curr\_v) \leftarrow d_{curr\_v}$ , ( $d_{curr\_v} \in demand(curr\_v)$ )
     3.2.2: Commit in the commit-table.
   3.3: Else
     3.3.1: return conflict.
4: If( $label(curr\_v) = \text{CONST}$ )
   4.1:  $nodeval(curr\_v) \leftarrow d_{curr\_v}$ , ( $d_{curr\_v} \in demand(curr\_v)$ )
   4.2: return
5: If ( $label(curr\_v) = \text{RESOURCE}$  or  $\text{PIPE\_RESOURCE}$ )
   5.1: Check if the node enabling  $curr\_v$  can be satisfied.
   5.2: If(read-resource node)
     5.2.1:  $write\_resource\_list_{curr\_v} \leftarrow \text{SearchWriteResource}(curr\_operation)$ .
     5.2.2: If no write-resource exists, commit. // usually this is instruction memory
     5.2.3: Try all write-resource until pathway is satisfied.
     5.2.4: If conflict all the pathways return conflict.
   5.3: If(write-resource node)
     5.3.1: Back-Propagate the demand_value.
     5.3.2: If conflict in all the paths return conflict.
6: If ( $label(curr\_v) = \text{MUX}$ )
   6.1: Check if there exists any satisfiable node-value in  $demand(curr\_v)$ 
   6.2: Appropriately try to set the control of the  $curr\_v$ 
     by back propagating the required control value.
   6.3: If conflict in all the paths return conflict.
7: If ( $label(curr\_v) = \text{ALU\_OP}$ )
   7.1: Apply constant lookahead if one of its operands is CONST.
   7.2: If there exists no feasible  $d_{curr\_v}$  ( $d_{curr\_v} \in demand(curr\_v)$ )
     apply block-path heuristic and return conflict.
   7.3: If conflict in all possible variations, return conflict.

```

end

Next we describe the *SearchWriteResource* algorithm. We define, $v_j \implies v_i$ when there is a path from node (say v_j) denoting $operation_j$ to node (say v_i) representing $operation_i$ in the LISA Operation DAG(\mathcal{D}). Thus, *parent* and *ancestor* operations in the \mathcal{D} are defined as follows.

$$parent(operation_i) = \{operation_j \mid \text{s. t. } e(v_j, v_i) \in E(\mathcal{D})\} \quad (1)$$

$$ancestor(operation_i) = \{operation_j \mid \text{s. t. } v_j \implies v_i \in \mathcal{D}\} \quad (2)$$

ALGO. 5.2. SearchWriteResource

```

SearchWriteResource( $curr\_operation$ )
begin
1: If(label = PIPE_RESOURCE)
   1.1: First priority to nodes with operation  $\in$ 
      $parent(curr\_operation)$  and in immediately previous stage.
   1.2: Second priority is given to write-resource nodes in other previous stages
     and operation  $\in ancestor(curr\_operation)$ .
2: If(label = RESOURCE)
   2.1: Priority to nodes in previous stages.
   2.2: Second priority is given to nodes in current and future stages.
end

```

Next, we present the algorithm for placement of the vertices in the commit-table for the spatio-temporal interpretation of the pathway.(refer 5). As mentioned earlier, the determination of the stage is straight-forward since that is determined by the stage information embedded within each vertex. Hence, only the cycle-selection needs to be elaborated. It takes the current vertex ($curr_v$) and the previous vertex ($prev_v$) (whose cycle and stage are given by $prev_stage$ and $prev_cycle$) as the inputs.

ALGO. 5.3. DetermineCycleInCommitTable

```

DetermineCycleInCommitTable( $curr\_v$ ,  $prev\_v$ )
begin
1: If (label = PIPE_RESOURCE)
   1.1: if ( $curr\_stage$  precedes  $prev\_stage$ )
     1.1.1: if ( $curr\_v = ancestor(prev\_v)$ ) // place it in the diagonal
       1.1.1.1:  $curr\_cycle \leftarrow prev\_cycle - (prev\_stage - curr\_stage)$ 
     1.1.2: else // it is not feasible since the pipeline will be overwritten
       1.1.2.1: return error
   1.2: else if ( $curr\_stage$  succeeds  $prev\_stage$ )
     1.2.1:  $curr\_cycle \leftarrow prev\_cycle$ 
2: If (label = RESOURCE)
   2.1:  $available\_cycle \leftarrow$  cycle of the topmost cell in the column  $curr\_stage - 1$ 
   2.2: if ( $prev\_stage$  follows  $curr\_stage$ )
     2.2.1: if ( $curr\_v = ancestor(prev\_v)$ )
       2.2.1.1:  $curr\_cycle \leftarrow prev\_cycle - 1$  // fill the diagonal
     2.2.2: else
       2.2.2.1:  $desired\_cycle \leftarrow prev\_cycle - 2$ 
       2.2.2.2: if( $available\_cycle < desired\_cycle$ )
         2.2.2.2.1:  $curr\_cycle \leftarrow available\_cycle$ 
       2.2.2.3: else
         2.2.2.3.1:  $curr\_cycle \leftarrow desired\_cycle$ 
     2.3: else
       2.3.1:  $desired\_cycle \leftarrow prev\_cycle - 1$ 
       2.3.2: if ( $available\_cycle < prev\_cycle$ )
         2.3.2.1:  $curr\_cycle \leftarrow available\_cycle$ 
       2.3.3: else
         2.3.3.1:  $curr\_cycle \leftarrow desired\_cycle$ 
end

```

6. Case Study

The automatic test pattern generation approach, presented in this paper, are tested with two different pipelined RISC processors. The first one, LTRISC, is a 32-bit 4-stage pipelined RISC

processor with basic support for arithmetic, load-store and branch operations. LTRISC contains mechanism for automated detection of pipeline data-hazards. All the instructions of LTRISC are conditional instructions, preventing straight-forward high-coverage test-case generation. The ICORE [15] architecture is dedicated for Terrestrial Digital Video Broadcast (DVB-T) decoding. It is based on a pipelined Harvard architecture implementing a set of general purpose arithmetic instructions as well as specialized trigonometric operations. Other notable features in ICORE, presenting hindrance to a smooth test-pattern generation, includes zero-overhead-loop, extensive branch instructions, complex instructions involving nested data-flow. A brief summary of the processors is shown in table 2.

	ICORE	LTRISC
Basis Architecture	21-bit RISC	32-bit RISC
Pipeline Stages	4	4
Lines of LISA Code	2200	1838
Lines of Verilog Code	25200	9501

Table 2. Benchmark Processors

The test program for both the architectures are automatically generated using the flow described in this paper. The run-time of the backtracking algorithm is observed to be extremely low, due to the heuristics applied in the process. For both the architectures, the complete test program is generated in less than a second on a AMD Athlon XP 2600+ Processor (1916 MHz, 512 MB RAM) running SuSE Linux 9.2 operating system. The total number of instructions generated for ICORE is 727, whereas for LTRISC, 132 instructions are generated. In order to avoid unnecessary large loops or jumping outside program location, the regular features from TGE e.g. branch address biasing are extensively used during coverage constraint generation. Actually, it gave the complete test-automation tool an immense advantage to have a constrained random test pattern generator at the back-end.

The test programs are then fed to the HDL descriptions and the coverage metrics are measured using Synopsys VCS tool flow [16]. The overall results of the block coverage and statement coverage are presented in the following table 3.

	ICORE	LTRISC
RTL Block Coverage	96.87%	97.71%
RTL Statement Coverage	98.42%	98.52%

Table 3. Results of Coverage Measurement

By checking the individual sections of the two processors, it turned out that the instruction behaviors are completely covered. Actually, those are the conditional blocks explicitly targeted. However, the register file contained several general purpose registers, which remain uncovered. For example, the following piece of assembly code is generated to cover pipeline data-hazard in LTRISC.

```
r0 = 1600
r0 = (r2 ≤ 1685 )
r0 = (r0 ≤ 1529 )
```

Here, register r0 is used to model the data-hazard in comparison instructions of LTRISC. Similarly, the coverage of conditional blocks presented the test generator with varied choices of general purpose registers and one is randomly picked up. This assured full coverage of the instruction behaviors but, the coverage of the register files is still at the best constrained random. This resulted in less than 100% block coverage for LTRISC. However, the register file is regular in their structure and their coverage can be achieved by writing generic test-pattern generation subroutines. For ICORE, the overall block coverage and statement coverage is comparable

to LTRISC. However, ICORE contained few uncovered conditional blocks in the instruction behavior. Closer inspection revealed that those blocks are conditioned by I/O pins. Obviously those could not be controlled by the instruction grammar-based TGE.

It is interesting to compare the advantage of the automated test pattern generator with the manual constrained random test pattern generation approaches. By allowing an experienced processor designer to use the ADL-driven constrained random TGE (the back-end of the presented tool-flow), it took 2-3 days to achieve similar coverage results. A completely manual creation of the test pattern for achieving full instruction coverage, without the deep knowledge of the target processor, will certainly take much longer. Considering the test pattern is generated within few seconds - the importance of this work is obvious.

7. Conclusion and Future Work

This paper presents a fully automatic test generation framework for functional verification of modern embedded processors. The test generation is driven by ADL, a high level processor description formalism. This allows an early test generation mechanism, which ensures high coverage in lower level of abstraction e.g. RTL. The case study with two simple scalar RISC processors show the efficacy of this framework.

In future, this tool will be tested with more complex processors. The influence of this test generation over other coverage metrics like fsm coverage, toggle coverage will be studied, too.

8. REFERENCES

- [1] P. Mishra et al. A top-down methodology for validation of microprocessors. In *IEEE Design and Test of Computers (Design and Test)*, pages 122–131, 2004.
- [2] CoWare/LISATEk. <http://www.coware.com>.
- [3] Target Compiler Technologies. <http://www.retarget.com>.
- [4] D. Kammiller, E. M. Witte et al. ASIP Design and Synthesis for Non Linear Filtering in Image Processing. In *Design, Automation & Test in Europe (DATE)*, 2006.
- [5] T. Kempf, M. Dörper et al. A Modular Simulation Framework for Spatial and Temporal Task Mapping onto Multi-Processor SoC Platforms. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, 2005.
- [6] Calypto Design Systems. <http://www.calypto.com/>.
- [7] OpenVera. <http://www.open-vera.com/>.
- [8] H. Koo and P. Mishra. Functional Test Generation using Property Decompositions for Validation of Pipelined Processors. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, 2006.
- [9] F. Corno et al. Automatic test program generation: A case study. In *IEEE Design and Test of Computers*, 2004.
- [10] A. Adir, E. Almog et al. Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification. *IEEE Design and Test*, 2004.
- [11] O. Luethje. A methodology for automated test generation for lisa processor models. In *The Twelfth Workshop on Synthesis And System Integration of Mixed Information technologies, Kanazawa, Japan. Synthesis And System Integration of Mixed Information technologies (SASIMI 2004)*, October 18-19, 2004.
- [12] P. Mishra and N. Dutt. Functional coverage driven test generation for validation of pipelined processors. In *Design Automation and Test in Europe (DATE)*, pages 678–683, 2005.
- [13] F. Fallah, S. Devadas and K. Keutzer. Functional Vector Generation for HDL models using Linear Programming and 3-satisfiability. In *DAC '98: Proceedings of the 35th annual conference on Design automation*, 1998.
- [14] A. Chattopadhyay, A. Sinha, D. Zhang, R. Leupers, G. Ascheid, H. Meyr. Integrated Verification Approach during ADL-Driven Processor Design. In *Seventeenth IEEE International Workshop on Rapid System Prototyping*, 2006.
- [15] T. Gloekler and S. Bitterlich and H. Meyr. ICORE: A Low-Power Application Specific Instruction Set Processor for DVB-T Acquisition and Tracking. In *Proc. of the ASIC/SOC conference*, Sep. 2000.
- [16] Synopsys. VCS <http://www.synopsys.com/products/simulation/simulation.html>.