Integrated Verification Approach during ADL-driven Processor Design

Anupam Chattopadhyay[1], Arnab Sinha[2], Diandian Zhang[1], Rainer Leupers[1], Gerd Ascheid[1], and Heinrich Meyr[1]

[1]Aachen University of Technology, Integrated Signal Processing Systems, 52056 Aachen, Germany,
Email - anupam@iss.rwth-aachen.de, Tel - 00-49-241-80-28272, Fax - 00-49-241-80-22195
[2]Department of Computer Science and Engineering, Indian Institute of Technology Kharagpur, India

Abstract

Nowadays, Architecture Description Languages (ADLs) are getting popular to achieve quick and optimal design convergence during the development of Application Specific Instruction-Set Processors (ASIPs). Verification, in various stages of such ASIP development, is a major bottleneck hindering widespread acceptance of ADL-based processor design approach. Traditional verification of processors are only applied at Register Transfer Level (RTL) or below. In the context of ADL-based ASIP design, this verification approach is often inconvenient and error-prone, since design and verification are done at different levels of abstraction. In this paper, this problem is addressed by presenting an integrated verification approach during ADL-driven processor design. Our verification flow includes the idea of automatic assertion generation during high-level synthesis and support for automatic test-generation utilizing the ADL-framework for ASIP design. We show the benefit of our approach by trapping errors in a pipelined SPARC-compliant processor architecture.

1. Introduction

The growing design complexity coupled with diverse, narrowing application domains 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. While optimal processor performance is achieved by flexible solutions like introducing special instructions into the Instruction-Set Architecture (ISA), the task of verification is simplified by starting from pre-verified IP blocks. As a result, the processor design community have emerged with distinct trends in the processor design approach. At one extreme, the processor design is performed on a high level of abstraction, thereby allowing fast design space exploration and ample scope of optimization for target-specific architectures. Architecture Description Languages (ADLs) offer one promising avenue for efficient design of an highly application-specific processor. ADLs [1] [2] [3] are employed to model the Application Specific Instruction-Set Processor (ASIP) at a higher level of abstraction. These *ADL-based approaches*, require to pass through several synthesis steps and thereby, come with a costly verification effort. On the other extreme, the processor design is simplified by limiting the choice of design within a range of pre-verified IPs or allowing fine-tuning of an existing template processor core [4] [5]. This *template-based approach* allows quick verification with limited choice of application-specific optimizations. Since, the major part of the processor is available as pre-verified IP, the verification concern is focussed on the possible designer-defined configurations or the extensions of the basic template. In order to reach the high flexibility like ADLs, more and more configurations need to be supported, eventually increasing the verification effort. Clearly, there is a trade-off between optimization and verification effort.

Independent of the two separate approaches, verification has received continuous attention from EDA researchers. With the traditional processor design approach, the verification is primarily done in the Register Transfer Level (RTL) or below. *Simulation-based* verification is a commonly known verification technique. In this domain, test patterns targeted to detect faults are fed into the design-under-test. However, the number of exhaustive test patterns being huge, automated test pattern generation approaches are adopted. On the other hand, with the advent of hardware verification languages like *Vera*, *e*, designed for automatic verification environment, has tried to blend simulation based approach with *formal verification* approaches. Among the semi-formal methodologies, *assertion-based verification* achieved prominence. Assertions are properties, which expresses the designers' intent in a purely mathematical language. The properties, expressed as assertions, help the verification engineer in a large way because, the latter can identify the designers' intent in a clear logical manner.

In essence, varied verification approaches are making their way into commercial and academic acceptance. Contemporarily, in order to handle the demanding design complexity, the designers are seeking their way to express the design in a high level of abstraction, which sometimes come at the penalty of longer verification cycle. With traditional verification approach, the ADL has to be synthesised to generate the RTL and then RTL-based verification methodologies need to be applied. In this approach, the ADL-based information about the architecture is often lost in the synthesis process thereby, making the RTL-driven verification process imprecise and computationally complex. An integrated verification flow merging the concepts of ADL and different genres of verification is of high interest for fast and error-free processor design. We address this issue in this paper. Our work focuses on the *ADL-based approach*, for which there is a strong need for seamless and robust verification methodology. We chose the ADL LISA for our work. We have integrated a mix of semi-formal and simulation-based verification methodology in our ADL-based verification approach. In summary, the contribution of this paper is to present:

- An ADL-driven assertion-based verification methodology.
- A semi-automatic user-constrained test generation methodology for ADL-driven ASIP design.

Our work complements the existing ADL-based processor development flow by adding the verification flow. The components of the verification framework are completely derived from the ADL description, thereby minimizing the chances of design inconsistency. As a case study, we chose a 7-stage, SPARC v8 compatible processor and showed the strength of our approach by uncovering hidden and injected errors.

The rest of the paper is organized as follows: section 2 introduces the previous work in this domain. Section 3 discusses the basic features of LISA. Section 4 describes the complete verification flow in detail. Section 5 elaborates and analyzes our case study. We conclude with the summary and outlook.

2. Related Work

Processor verification had always remained a major issue for the research community from the inception of processor design. Evolving processor design concepts e.g. pipelining and VLIW made the task of verification even more demanding, thereby rendering exhaustive manual testing to be an impractical and inefficient solution. Consequently, numerous verification approaches and techniques emerged over the past decades. Here, we limit our discussion among ADL-based verification approaches.

ADL-based Verification Techniques has received serious research interest in recent years. Mishra et al. [6] proposed a graph-based functional test-case generation. They transformed the processor description, written in EXPRESSION ADL [7], into a graph consisting of nodes (representing *functional units* and *storages*), and edges (representing *pipeline* operations and *data-transfer* operations). By the help of that graph, they defined a graph-based coverage metric, which in turn, led to a set of test-patterns. In the domain of formal verification, Mishra et al. [8] successfully transformed an ADL representation into an equivalent SMV representation and used SMV-based model checking tools to generate test-patterns. In another context, Luethje [9] approached the problem from a different perspective. For the ADL LISA [10], he analysed the *behavior* section of the LISA model to achieve 100% ADL-code coverage. In another recent advancement, Luethje demonstrated the usage of the Genesys test generator [11] for processors described in LISA [12]. The retargetable test program generator from the ADL nML [13] comes close to our work, where an automatic selective and/or random test-pattern generation is proposed.

In summary, there are existing ADL-based verification approaches. However, none of the above-mentioned ADLs support *automatic generation of assertions from the ADL* to the best of our knowledge. ADL EXPRESSION used assertions during their top-down validation flow. In their approach, the ADL description automatically generates an RTL description. A symbolic simulator automatically generates property/assertion from the generated RTL description [14] (figure 2). The drawback with this approach is that the generation of property/assertion is done from the automatically generated RTL and not from the ADL. Therefore, this approach heavily relies on the correct generation of the RTL from the ADL. Understandably, in their approach the automatically generated RTL is considered as golden reference, which is used to verify manually implemented RTL design. In our approach, we take it a step higher and try to verify the correctness of the ADL description itself. Therefore, it is imperative to automatically generate the assertions from the ADL, as proposed in this paper.

In the domain of ADL-based automatic test pattern generation, there are existing methodologies [6] [15]. We propose an alternative ADL-based test-generation methodology, where the user can strongly influence the test-pattern generation. The existing test-generation methodologies focus on the statement coverage [9] or functional coverage [15]. These approaches, though introduces novel test pattern generation techniques, allows little user interaction in the process. Our approach is semi-automatic in nature, where user deals with the *instruction grammar* to influence the test patterns. This is closer to the approach of IBM Genesys [11] but, our approach does not require separate modelling of architecture or test template.

3. Brief Overview of LISA

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.

3.1 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.

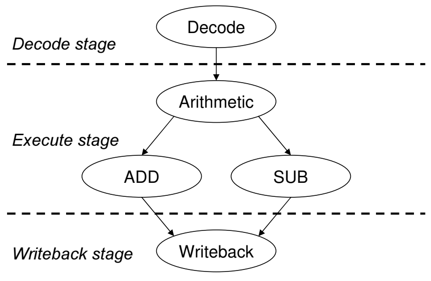


Figure 1. LISA Operation DAG Example

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 either *Behavior Calls* or *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. A chain of operations, forming a complete branch of the LISA operation DAG represents an instruction in the modelled processor.

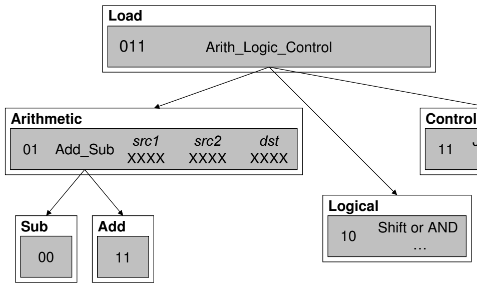


Figure 2. LISA Coding Tree Example

A LISA Operation contains different subsections to capture the entire processor behavior. The ones relevant for RTL synthesis are discussed below.

Instruction Coding Description: The instruction encoding of a LISA operation is described as a sequence of *coding fields*. Each coding field is either a terminal bit sequence with "0", "1", "don't care"(X) bits or a nonterminal bit sequence referring to the coding field of another child LISA operation. An example of a coding tree is given in figure 2. In this example, the *Add* and *Sub* operations have only terminal codings whereas *Load*, *Arithmetic*, *Logical* and *Control* consist of both terminal and nonterminal coding fields.

Activations: A LISA operation can *activate* other operations in the same or a later pipeline stage. In either case, the child operation may be activated *directly* or via a *group*. A *group* collects several mutually exclusive LISA operations.

Behavior Description: The behavior description of a LISA operation corresponds to the datapath of the processor. Inside the behavior description 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. The behavior section of every *LISA operation* is transformed into a *functional block* in the RTL datapath.

4. Integrated Verification Environment

In this section, we elaborate the flow and the major components of the proposed verification scheme in detail. We start with the dynamic flow of the scheme, followed by the detailed discussion on automatic generation of test-cases and assertions, to support the multiple stages of verification.

4.1 The Verification Flow

The flow of our verification scheme is as shown in figure 3. At the beginning, the processor description is written in LISA. The RTL description (along with *Assertions*), the *Instruction-Set Simulator* and the *Instruction-Grammar* are automatically generated from the LISA model. We also built up a library of *assertion-modules* containing the assertions, which is written in OVA and SVA. The assertion library contain basic static and temporal assertions allowing e.g. comparison of signals. The assertion modules from this library are used as templates to instantiate actual assertions inside the RTL description. The *instruction-grammar* is fed

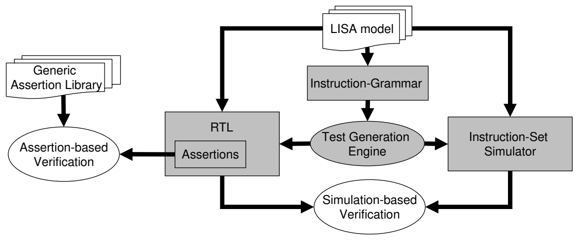

LISA model
Generic Assertion Library
Instruction-Grammar
RTL
Assertions
Test Generation Engine
Instruction-Set Simulator
Assertion-based Verification
Simulation-based Verification

Figure 3. Complete Verification Flow

to the *Test Generation Engine (TGE)* to churn out test-cases automatically. In this context, we define the instruction-grammar.

Instruction-Grammar: The instruction grammar represents the valid instructions in Backus-Naur Form (BNF) grammar. In other automatic test generation approaches, the *instruction-set hierarchy* is either manually developed [16], or an instruction-library needs to be set up [17]. In Genesys-Pro [11] each instruction needs to be manually modelled in C++, introducing more complex semantics into it. Here, we automatically generate the instruction-grammar from the ADL operation-graph (see section 3.1). 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`. The test generation engine loads this instruction grammar in an internal data-structure. Note that, this instruction grammar is essentially an instruction coding tree (like figure 2) represented in another form.

insn	: add dst_reg src_reg src_reg
	‖ sub dst_reg src_reg src_reg
	‖ mac dst_reg src_reg src_reg src_reg
	‖ nop
add	: 0000 0001
sub	: 0000 0010
mac	: 0000 0011
src_reg	: 0000 xxxx
dst_reg	: 0000 xxxx
nop	: 0000 0000 0000 0000 0000 0000 0000 0000

Table 1. Exemplary Instruction Grammar

Formally, the instruction grammar can be defined as following.

- Terminal (T): It is either '0' or '1' or 'x' (don't care).
- Non-Terminal (NT): Syntactic variables that denote the sets of strings. (containing terminals and/or non-terminals).
- Production (S) : $P : \alpha_1\alpha_2...\alpha_n$, where $\alpha_i \in NT \setminus \{P\}$ or $\alpha_i \in T$, $P \in NT$, $\forall i \in N$, where N is set of positive integers. In this context, we define P as the *Producer*.
- Rule: A set of productions (S) having the same producer P. Formally, S = $\{S_i \mid P_i = P, \forall i \in N\}$,

  where $S_i \rightarrow P_i : \alpha_1\alpha_2...\alpha_n$

First phase of verification: In the first phase, we run the test-cases to simulate the HDL code. We propose a definitive set of *abstract fault models*, targeted to cover typical processor errors. On the basis of that, assertions are generated automatically. In the HDL simulation, these assertions help to dynamically identify RTL bugs. The RTL simulation contributes in the coverage analysis, which in turn, may be taken as a feedback for the test generation scheme.

Second phase of verification: In the second phase, we separately run the instruction-set simulator and the RTL simulator, with the same set of test-cases, and compare the processor states in each cycle, as obtained in them. If the equivalence in the simulation-based verification phase is established, it is likely that the processor is working perfectly, otherwise, the design calls for further re-modelling and verification cycles.

In the following subsections, we discuss, in detail, the features of the TGE, the basis and complexities of automatic generation of assertions and the simulation-based verification flow.

4.2 Test Generation Engine (TGE)

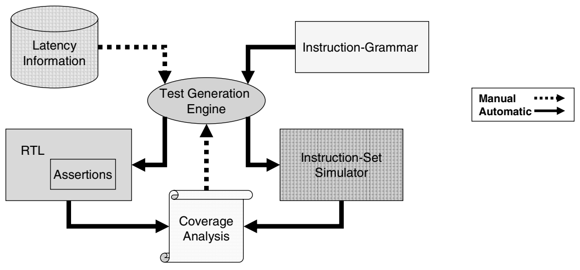


Figure 4. Test Generation Engine

This TGE works in the front-end of the verification-platform. Figure 4 explains the environment of the TGE. It receives the *instruction-latency information* alongwith the instruction-grammar to produce test-cases for the RTL as well as instruction-set simulators. Although, the generated test-cases are not meaningful applications, with careful directives, as we will show later, the TGE mimics the realistic application-like scenario quite effectively. In this regard, the TGE can effectively be used by a designer to prepare *synthetic testbenches* for any given processor. The ADL code coverage (obtained from the instruction-set simulation), the RTL statement coverage (obtained from the RTL simulation) and the latency information are manually fed to influence the nature of generated test patterns. The TGE is equipped with different constraint modes. By constraint mode, we mean the degree of restriction imposed on the process of generation of the test-cases. It can be anywhere between a fully directed and a fully randomized one.

Features of the 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 features are discussed.

Fully exhaustive test-case: The engine can generate fully exhaustive test case for an instruction grammar, i.e. all possible valid instructions of an ASIP, which is always a finite number.

Size Constraint: The designer might wish to test the design with an appropriate number of instructions where the *nature*, *context*, *occurrence-frequency* of instructions can be *precisely* controlled. The TGE supports such verification intent, by allowing the user to determine the number of instructions.

`insn`	: `mac bypass_reg src_reg src_reg`
	‖ `add dst_reg src_reg bypass_reg`
`bypass_reg`	: `0000 0101`

Table 2. Overriding rules

Storage Access Biasing: Data-forwarding is a feature often found in ASIPs. To verify data-hazard handling capability in the processor, we need to create an environment where same storage element (register/memory unit) is accessed in consecutive instructions. The TGE helps the designer to create those situations, in the following manner. Suppose, the following assembly language program is desired,

mac r5 $\langle$*src1_reg*$\rangle$ $\langle$*src2_reg*$\rangle$
add $\langle$*dst_reg*$\rangle$ $\langle$*src1_reg*$\rangle$ r5

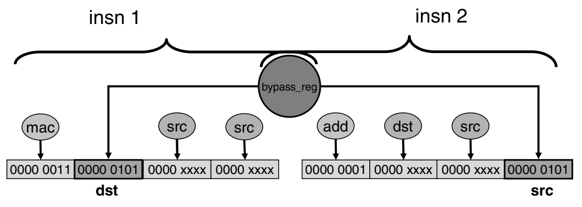


Figure 5. Grammar-Rule Overriding

By using the technique of *instruction-grammar-rule overriding* shown in figure 5, one can modify/add the following rules, as shown in table 2 to restrict the test-case productions. The last rule which is added to the rule database elegantly restricts the TGE to generate instructions having `r5` as one of its source/destination register.

Instruction-Group Occurrence Frequency Biasing: Within the scope of this feature, we can define a group of instructions, which are functionally similar in nature. Typical examples can be load-store operations, ALU operations, etc. It is very likely that few instruction-groups occur more frequently than some other instruction-groups. To mimic those situations, one can identify them and form instruction-groups, by the technique of overriding grammar-rules, and appropriately bias them, to occur in the desired frequency.

Instruction Occurrence Frequency Biasing: In some cases, we need to control the occurrence frequency of some specific instructions and not an instruction-group. Using the TGE, it is possible to bias the occurrence frequency of *add* or *sub* in the generated test-cases. The input of the TGE simply require the frequency biases of the chosen instructions, based on which it determines the suitable test-case. Moreover, the TGE supports the inquisitive designer to bias the instructions at a finer level of granularity. By this statement, it is meant that, the TGE can influence occurence frequency of `JMP` $\langle$`imm`$\rangle$ over `JMP` $\langle$`reg`$\rangle$.

Selection of instruction: A test-sequence containing specific instruction(s) can be generated. This is useful in order to test the processor behavior for that particular instruction.

Latency Information Insertion: If one instruction blocks a resource then, the following instruction waits for a specific number of machine-cycles, so that the execution of preceding operation terminates. The number of cycles, for which the latter instruction should wait, is known as the latency period of an instruction. The TGE supports such kind of test-case generation injecting the *nop* operation automatically if, the latency information is provided.

Automatic Test-Pattern Generation: 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. In case of an overwritten grammar rule, an entire sub-branch of the data-structure is replaced with the DAG of the new rule. The supports provided by the TGE necessitates traversal of the instruction grammar which is in essence a DAG. Thus the worst-case time-complexity is same as that of performing DFS in a graph i.e. $\mathcal{O}(n \times e)$, where n and e are the cardinalities of set of vertices and edges in the graph, respectively.

4.3 Assertions

Assertions are logical, sentential forms which are useful in expressing design properties explicitly. The assertions help the designer to express his verification intent from a higher level of abstraction. For elaboration, an assertion can be the following. At any instant of time, a unique address of the memory, can be written by only one hardware component. The major benefit of using such assertions is the formal rigor embedded in it. Assertions can be broadly classified into two groups, namely, *static* and *temporal* assertions. *Static* assertions do not include time, while, *temporal* assertions are useful in expressing properties which depend on time, i.e. the clock. In our methodology, we have used both static, as well as, temporal assertions, to detect the following abstract fault models.

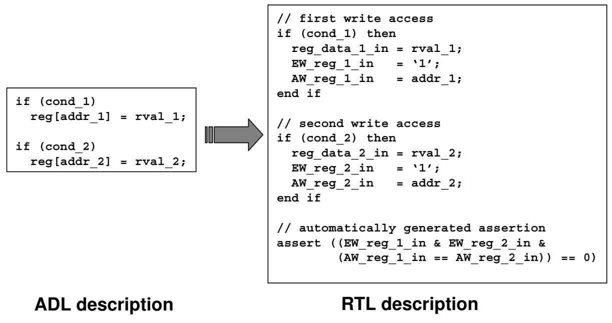
Abstract Fault Models: In this part, we discuss the various fault models which are covered by our methodology. These abstract fault models are the potential faults likely to be discovered in an ASIP. *Similar fault models have been used to derive test-patterns in* [15]. The contribution of these fault models in this paper is that we derive the assertions and not the test programs. Therefore, our approach is complementary to the test-pattern generation approach developed in [15].

Structural Fault Models: Structural fault models are those which are embedded in the micro-architecture. These faults are not always easy to be trapped at the ADL level. Usually, these bugs are detected during the RTL simulation phase since, the control flow of the ASIP needs to be monitored. From this perspective, we present the following fault models.

- Fault Model for Register Transfer Operations: The registers properly load a value after it is written to and provides the correct value after it is read from. Otherwise, the behavior is faulty. In order to detect whether more than one signal is trying to write simultaneously to a given register, the dynamic exclusiveness of available enable-write signals has to be verified.
- Fault Model for Register Write Operations: The behavior is faulty, if two or more instructions try to write the same register during the same cycle. In order to detect whether more than one signal is trying to write simultaneously to a given register, the dynamic exclusiveness of available enable-write signals has to be verified.
- Fault Model for Pipeline Control Operations: During stall/flush the pipeline behaves in a special manner. While the processor is *stall*-ed, the content of the pipeline registers do not change. Again, during *flush*, the contents of affected pipeline registers get replaced by zero. This behavior, if not satisfied, leads to a fault.
- Fault Model for Single Branch Execution: Due to partial or complete similarity in the code contribution of two instructions, the decoder might activate more than one execution sequence simultaneously (in a single-threaded processor). The model is declared faulty under such circumstances. To ensure execution of a single, valid chain of units, the exclusiveness of activated signals, as well as, non-existence of hanging activated signal should be ensured.
- Fault Model for Memory Accessing: A memory unit having multiple write-ports cannot be written in the same address at any instant of time. Moreover, the program memory should contain meaningful data. This can be verified by comparing the logical conjunction between the equivalence of enable-write signals and that of the addresses with *false*. Moreover, in order to ensure that the program memory contains meaningful data, the instruction-register contents can be checked to contain a non-*nop* instruction, at least once, during the whole span of the simulation.

Behavioral Fault Models: Behavioral fault models are those, which are usually difficult to track down in the micro-architecture because of the involved data-flow analysis required. For elaboration, there can be a particular signal set to high resulting in some other signal to be set high for next 5 clock-cycles, otherwise there is a fault. Intuitively, trapping of such design-violations requires involved semantic behavior analysis of each instruction in the ADL level, as they are not generic in nature. We provide the designer with the required platform,(i.e. the assertion library) with the help of which one can manually insert assertions in the generated HDL.

Automatic Generation of Assertions: In order to generate the assertions, we resort to static ADL analysis techniques [18]. By static analysis of a LISA model, we derive a **global conflict graph**, where a conflict edge between two LISA operations indicates that those are not mutually exclusive.



```
if (cond_1)
  reg[addr_1] = rval_1;

if (cond_2)
  reg[addr_2] = rval_2;
```

```
// first write access
if (cond_1) then
  reg_data_1_in = rval_1;
  EW_reg_1_in   = '1';
  AW_reg_1_in   = addr_1;
end if

// second write access
if (cond_2) then
  reg_data_2_in = rval_2;
  EW_reg_2_in   = '1';
  AW_reg_2_in   = addr_2;
end if

// automatically generated assertion
assert ((EW_reg_1_in & EW_reg_2_in &
        (AW_reg_1_in == AW_reg_2_in)) == 0)
```

Figure 6. Automatic Assertion Generation from ADL

One processor instruction is distributed in the LISA description over a chain of LISA operations. For a single-threaded processor, it is not possible to execute more than one instruction inside one pipeline stage. The afore-mentioned *fault model for single branch execution* is covered by a structural assertion, which checks if two or more exclusive LISA operations are activated at the same instance. The *fault model for memory access* is covered by another assertion, which runs through all possible combinations of memory access from the processor and simply asserts if more than one `enable_write_memory` signal is high at the same instance alongwith a conflicting memory address location. The *Fault Model for Register Transfer Operations* is detected by structural assertions similar in nature. In order to detect whether more than one signal is trying to write simultaneously to a given register, the dynamic exclusiveness of the available enable-write signals is checked.

One example of automatic assertion generation is shown in figure 6. Here, the the ADL description of register write access and corresponding RTL description is written in pseudo-code. The prefix *EW* indicates enable write signal, whereas *AW* stands for address of the write access. For a non-faulty behavior, the two register-write operations cannot be activated at the same cycle given, the addresses are equal. Understandably, such errors are difficult to track from static analysis. An automatically generated assertion eases the verification by checking all such combinations across the complete processor.

4.4 Simulation-based Verification

This is the second and the last phase of our complete verification platform. In this phase, we separately run the automatically generated instruction-set simulator (cycle-accurate) and the RTL simulator for the same test-case. The processor states are compared for each cycle between two simulations. We consider the cycle-accurate instruction-set simulator as the golden reference. Hence, a mismatch of the states indicates a design implementation error. The complete simulation-based verification is guided by a shell script.

5. Case Study

In order to implement and test the strength of our methodology, we required a considerably complex ASIP. LEON3 [19] is a 7-stage pipelined processor, compliant with the SPARC V8 architecture. Due to its SPARC compatibility (which is a well-studied architecture) and availability of its full source code under the GNU GPL license, LEON3 finds high relevance in research and academic pursuits. For our case study, we implemented the integer pipeline of LEON3 processor with integrated multiplication and division unit using LISA. In the following sub-sections, we discuss the results of our verification approach.

5.1 Assertion-based Verification

In our case study, we have detected design errors of two kinds. Firstly, there are *resident errors*, which were unknowingly introduced during the design-phase by the designer. Secondly, there are *injected errors*, which were intentionally introduced during the verification-phase.

Design Resident Errors

Same write-port of the simulated memory was being accessed, simultaneously, by multiple units. It got detected, as the assertion responsible for checking exclusiveness among the write-enable signals, got triggered.

The designer wanted to assign the following in the write-back stage, `Reg[reg_num] = a; Reg[reg_num | 1] = b;` Unfortunately, in a corner case, `reg_num` was *odd*, and hence the same register got accessed twice in the same cycle. It got trapped by the same class of assertion as mentioned in the previous example. This kind of faults can only be found by *dynamic execution* of instructions.

The execution of the *div* instruction was initially carried out in the EX stage of the pipeline, alongwith other arithmetic, logic instructions, where all of those were activating *write_regarith* operation of WB stage. The designer wanted to decrease the critical path of the processor, which was running along the *div* operation. The execution of *div* instruction was distributed over two pipeline stages - *EX* and *ME*. Due to unmindful copying, still then, both the parts were erroneously activating the *write_regarith* operation, and got itself trapped into the single branch execution fault model.

Trapping of Injected Bugs

A temporary register was inserted inside the LEON3 model which was written by two instructions, from different pipeline stages, simultaneously. It was caught by the fault model for register transfer operations.

Two new instructions were inserted, which can be represented by the following table 3. Here `x[29]` stands for 29 consecutive *don't care* bits in the instruction-word. It is easy to appreciate that, `testA` and `testB` have *partial similar coding contribution* which made the decoder activate two execution branches simultaneously. This design violation was covered by our fault model for single branch execution.

`test_insn`	`: testA ‖ testB`
`testA`	`: 110 x[29]`
`testB`	`: 1 testC x[28]`
`testC`	`: 100`

Table 3. Instructions with Coding-Overlap

5.2 Simulation-based Verification

In the second phase, we tried to catch some of the behavioral bugs hidden in the design. In order to trap them, we ran the HDL simulation as well as the instruction-set simulation with the same test-cases, and compared their results in a bitwise fashion, to reveal any mismatch in the processor states as obtained from both the simulations. The test-case used for this purpose is automatically generated from the TGE, which attains a high ADL behavior coverage. To our satisfaction, the major design resident errors got trapped in first phase of verification itself. This proves the strength of our assertion-based verification methodology.

5.3 Coverage Analysis

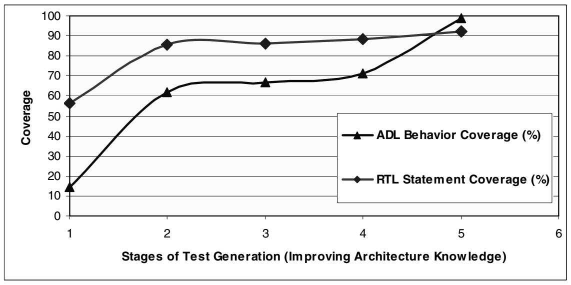


Figure 7. ADL vs RTL Coverage

Here, we present a comparative study of the effect of incorporating design-knowledge with our proposed verification support, on coverage results. With increasing architecture knowledge, we found steeper rise in the ADL coverage, compared to that of RTL coverage (refer figure 7). The ADL coverage represents the line coverage in the ADL *behavior* description and the RTL coverage represents the line coverage in the RTL description. The increasing design expertise is simply an abstract notation, where the higher value indicates greater usage of the architecture knowledge in test-generation.

The comparative result that we obtained regarding the ADL coverage vs RTL coverage, has demonstrated some interesting features in it. We utilized the power of TGE for the generation of required test-benches, moving from pure random to constricted test-cases gradually. As we used more and more directed test-scenarios the initial coverage-gap of about 41.9% converged at a point when both the coverages were approximately 90%. With the aim to achieve more coverage, we restricted test-cases further, to reach the final position where ADL coverage is **98.88%**, leaving RTL coverage at **92%**. This high coverage is obtained by using only **532** instructions. 100% ADL coverage was not achievable since, we restrained from testing the LEON3 FPU unit, which is under development in LISA. It is interesting to note that the ADL coverage is initially much lower than the RTL coverage, whereas with high design knowledge the ADL coverage is higher than the RTL coverage. The reasoning behind this observation is the following. In an ADL, the hardware details are implicitly mentioned. For elaboration, an operation, has got the resource-usage declaration, coding, and the behavior, in the ADL level. The corresponding RTL forms are registers, decoders and data-path respectively. Hence, a given behavior gets expanded in RTL code. So, RTL coverage is initially bulkier, compared to ADL behavior coverage. With the increase in number of instructions, ADL coverage rises, while RTL coverage curve assumes a nearly flat, slow-rising gradient, as the major bulk is already covered. The two coverage curves converge at a point (in our case, when both attain 90% coverage). Beyond that point, ADL coverage curve steeply rises, as more and more corner cases get revealed. These cases have more share in the ADL-code than their RTL counterpart have in the complete RTL description. Therefore the percentage of code which is covered in ADL, is more than the percentage of code, covered in RTL.

6. Conclusion and Future Work

With the growth in demand of complex, optimized, fault-critical systems, ADLs integrated with verification support will become inevitable for the designers. Traditional verification approaches, performed using test automation tools or assertions begin at RTL level and therefore key design points might be missed due to the complexity of the system. In this paper, we presented a verification methodology, in order to assist the ADL users with a completely self-sufficient, consistent design-and-test environment. The novelty of this approach lies in having stronger control over the complete verification flow, without doing away with the comfort of high abstraction level. Moreover, experiments showed that the integration of TGE and assertion-based verification approach would not burden the tool-chain and micro-architecture generation process, because of its manageable light kernel.

Our future work will include the semantic analysis of each instruction behavior for the automated generation of the behavioral assertions and guiding of test pattern generation.

7. REFERENCES

[1] A. Hoffmann, H. Meyr, and R. Leupers. *Architecture Exploration for Embedded Processors with LISA*. Kluwer Academic Publishers, 2002.

[2] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt and A. Nicolau. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, Mar. 1999.

[3] A. Fauth et al. Describing Instruction Set Processors Using nML. In *Proc. of the European Design and Test Conference (ED&TC)*, 1995.

[4] MIPS Technologies. *http://www.mips.com*.

[5] Tensilica. *http://www.tensilica.com*.

[6] P. Mishra and N. Dutt. Graph-based functional test program generation for pipelined processors. In *Design Automation and Test in Europe (DATE), Paris, France*, pages 182–187, February 16-20, 2004.

[7] P. Grun, A. Halambi, A. Khare, V. Ganesh, N. Dutt and A. Nicolau. EXPRESSION: An ADL for System Level Design Exploration. Technical Report 98-29, Department of Information and Computer Science, University of California, Irvine, Sep. 1998.

[8] P. Mishra and N. Dutt. Automatic functional test program generation for pipelined processors using model checking. In *Seventh IEEE International Workshop on High Level Design Validation and Test (HLDVT)*, 2002.

[9] 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*, October 18-19, 2004.

[10] A. Hoffmann, O. Schliebusch, A. Nohl, G. Braun, O. Wahlen and H. Meyr. A Methodology for the Design of Application Specific Instruction-Set Processors Using the Machine Description Language LISA. In *Proc. of the Int. Conf. on Computer Aided Design (ICCAD)*, Nov. 2001.

[11] A. Adir et al. In *Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification*. Copublished by the IEEE CS and the IEEE CASS, March-April 2004.

[12] O. Luethje et al. Designing and Modeling MPSoC Processors and Communication Architectures. In G. Matthias and K. Kuetzer, editors, *Building ASIPs: The Mescal Methodology*. Springer, 2005.

[13] Target Compiler Technologies. *http://www.retarget.com*.

[14] P. Mishra, N. Dutt, N. Krishnamurthy and M. S. Abadir. A Top-Down Methodology for Validation of Microprocessors. In *IEEE Design and Test of Computers (Design and Test)*, 2004.

[15] 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.

[16] K. U. Bhaskar, M. Prasanth, G. Chandramouli, V. Kamakoti. A universal random test generator for functional verification of microprocessors and system-on-chip. In *Proceedings of the 18th International Conference on VLSI Design (VLSID'05)*.

[17] F. Corno et al. Automatic test program generation: A case study. In *IEEE Design and Test of Computers*, 2004.

[18] O. Schliebusch, A. Chattopadhyay, E. M. Witte, D. Kammler, G. Ascheid, R. Leupers and H. Meyr. Optimization Techniques for ADL-driven RTL Processor Synthesis. In *IEEE Workshop on Rapid System Prototyping (RSP)*, 2005.

[19] Gaisler Research. *http://www.gaisler.com/*.