Generating an Efficient Instruction Set Simulator from a Complete Property Suite

Ulrich Kühne
Institute of Computer Science
University of Bremen
28359 Bremen, Germany
ulrichk@informatik.uni-bremen.de

Sven Beyer Christian Pichler
OneSpin Solutions GmbH
Theresienhöhe 12
80339 Munich, Germany
{Sven.Beyer, Christian.Pichler}@onespin-solutions.com

Abstract

Instruction set simulators can be used for the early development and testing of software for a processor before it is manufactured. While gate-level simulation offers cycle-accurate results, performance of the simulation is typically not sufficient for in-depth software testing. In addition, such a gate-level simulation cannot be carried out in the early phases of the design process when only the instruction set architecture (ISA) is present and the design is not yet complete. Therefore, more abstract simulators are based on the ISA; these simulators can achieve a performance of several million instructions per second. However, by introducing a simulator separate from the design, the ISA has to be re-implemented for the simulator. Therefore, there is a risk that the instruction set simulator is not in sync with the design or the ISA. We present an approach to automatically generate an instruction set simulator from a complete property suite, which can be used for the formal verification of the processor. In this way, we obtain a provably correct simulator with relatively small effort. We show the feasibility of the approach for an industrial design; the performance of the resulting simulator is shown to be comparable to custom state-of-the-art simulators.

1. Introduction

In today's processor and system design flows, instruction set simulators (ISS) play an important role. One major field of application of ISS is pre-silicon software development, enabling the simulation of software before the target system is manufactured or even the design is finished.

With increasing system complexity, simulation performance in terms of executed instructions per second has become an important factor. Thus, rather than simulating program code on a gate level or cycle accurate model of the design, ISS are based on the instruction set architecture (ISA) and implemented in high level languages like C++. For such an ISS, the ISA has to be reimplemented manually. There are several tools that provide dedicated languages for the description of instruction set architectures (see e.g. [1], [2]). However, since both design and ISS are derived from the ISA with a certain degree of independence, there is a risk that the actual design behaves differently from the ISS in some cases. Such a discrepancy between ISS and design may lead to erroneous software: while the software behaves as expected in the ISS, it does not work properly on chip.

In order to avoid the manual effort of developing an ISS, it is also possible to automatically derive an ISS from a high level or register transfer level (RTL) description of a processor [3]. Note that for optimized or pipelined RTL designs, such a higher level description is quite different from the actual design. Therefore, the automatic extraction of the ISS from the actual design is not feasible for these cases.

In order to achieve an ISS that really corresponds to the design, the ISS needs to be derived from the ISA that is actually used in verification. What is more, the verification should be carried out formally because only formal verification offers the chance of eliminating all discrepancies between ISA and the design. Today, formal hardware verification is already used in industry. Especially for safety critical systems involving medium size processor designs and embedded systems, formal verification can offer high quality solutions [4], [5]. One successful technique is *Interval Property Checking* (IPC) [6], a technique similar to Bounded Model Checking [7]. IPC is used in order to check if a design satisfies a set of properties which is written in a dedicated verification language. In contrast to simulation based methods which are not able to exhaustively cover all possible inputs for large designs, formal methods allow for a gap-free, i.e., *complete* verification. Here, complete means that the properties capture the behavior of the design in a unique way for each possible combination of states and inputs. Having finished the formal verification phase, the set of properties forms a functionally equivalent model of the verified design. This methodology generally offers the highest quality of verification.

In this paper, we show how a complete property suite resulting from the formal verification of a processor can be reused to automatically generate an ISS. In this way, the simulator is guaranteed to comply with the ISA that has been used for the verification of the processor; by construction, the ISS also complies to the design. This is

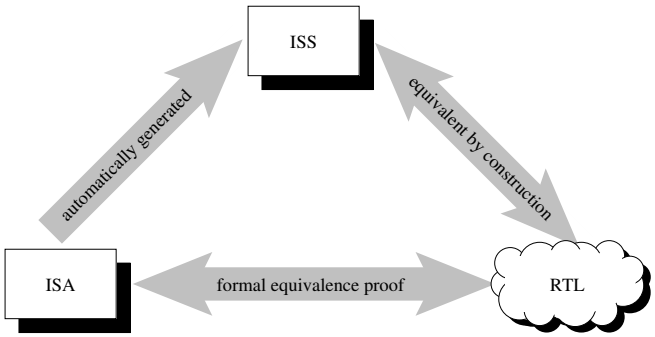

Figure 1. Generating a Provably Correct ISS

illustrated in Figure 1. Furthermore, by making use of an existing set of properties, the overhead for the creation of the ISS is relatively small. Nevertheless, the generated ISS offers a simulation performance comparable to state-of-the-art techniques. Note that the ISS can already be generated very early in the design and verification process, namely as soon as the ISA has been captured formally. With the completion of the verification, it is later on ensured that the generated ISS actually corresponds to the design.

It turns out that even in a late phase of the design flow—mostly as a consequence of the formal verification—changes in the design or in the specification are likely to occur [5]. In this case, one can obtain an adapted ISS from the revised formal property suite with virtually no additional effort, using our approach. Using the property suite as a single source for the specification ensures the consistency of software simulation with the verified design. This is a major contribution of this work. As the property suite offers a rigorous formalization of the specification, the simulator will reflect all sophisticated effects of the design that might be difficult to model using a high-level description, including e.g. exceptions and asynchronous interrupts.

The paper is structured as follows: related work is discussed in Section 2. In Section 3, the formal verification techniques used in this work are reviewed. The generation of the ISS is described in Section 4. In Section 5, experimental results are discussed, followed by the conclusions in Section 6.

2. Related Work

Another approach for a tight interaction between high-level simulation and verification is presented in [8]. The method is complementary to this work in that it uses an architectural description as starting point. From this description, an implementation is generated, as well as assertions to ensure the correctness of the design. However, the approach relies on simulation-based and semi-formal verification techniques, that do not allow for a complete verification. Furthermore, it is limited to those domains where a high-level synthesis is sufficient. Highly optimized RTL designs cannot be handled.

The idea of generating an executable model from a property suite is also used in [9]. In this work, it is even possible to generate models from incomplete specifications, resulting in partially nondeterministic behavior. However, the emphasis of [9] is on the generation of verification-friendly hardware designs rather than an efficient simulation model.

The automatic generation of instruction set simulators is examined in many publications. A common way to avoid the manual coding of high-performance simulators is the use of architecture description languages (ADL), which can then be compiled into an ISS. Examples for such ADL are *Facile* [1] or *LISA* [2]. However, these approaches still require the reimplementation of the processor semantics in the ADL. Thus, the functional equivalence of the ISS and the design remains to be shown.

A different approach is presented in [10]. There, the starting point is a structural description of all the components on a processor's data path. From this description, an instruction set is extracted automatically. The information can be used to generate an ISS as well as an RTL implementation. But, as the description can get quite complex, this does not replace the verification of the design. Furthermore, generated RTL code is typically not suited for highly efficient designs.

3. Background

3.1. Instruction Set Simulation

As for the technical aspects of instruction set simulators, there are mainly three different paradigms: interpretive simulation, compiled simulation and just-in-time compiled simulation. They differ in flexibility and performance. Interpretive simulators decode the instructions to be executed one by one. In this way, they offer the highest flexibility concerning run-time modifiable programs. The bottleneck in interpretive simulation is the instruction decoding. Compiled simulators carry out the decoding and in some cases even static scheduling at compile time. However, this technique is not applicable for run-time modifiable code and for dynamic scheduling. Therefore, just-in-time compiled simulation (JIT-CS) tries to combine the best of both worlds. In JIT-CS, information on previously decoded instructions is stored in a cache and can be reused when the instruction is executed again. In this way, a simulation performance comparable to compiled simulation can be achieved without losing the flexibility of the interpretive approach [2].

3.2. Formal Verification

Within the last two decades, there has been a great amount of research in formal verification techniques. Methods based on Boolean satisfiability (SAT) have proven to be a robust solution. One prominent technique is SAT based *Bounded Model Checking* (BMC), that has first been described in

[7]. Successive improvements in performance have made BMC a suitable method for the formal verification of larger scale designs. For the work at hand, we use the techniques described in [6], referred to as *interval property checking* (IPC). In the following, this verification methodology will be briefly outlined.

In contrast to the original BMC, only safety properties are verified using IPC. As digital circuits always have a finite response time, this is not a serious restriction in practice. It is rather natural to describe the intended behavior of a design in terms of safety properties in order to formalize the specification. Furthermore, this restriction leads to bounded properties that can be checked efficiently using a SAT solver.

The main idea of IPC is to use an arbitrary starting state instead of the initial state used in BMC. Any property that holds starting from an arbitrary state then also holds from any reachable state, i.e., it is exhaustively verified. Conversely, false negatives can occur in IPC, i.e. counterexamples for properties starting in unreachable states may be produced. These false negatives need to be removed by adding invariants in order to restrict the starting state. For more details on the idea of IPC and the following formalization, refer to [6].

A synchronous circuit is modeled as a finite state machine (FSM) $M = (I, S, S_0, \Delta, \Lambda, O)$ with input alphabet $I \subseteq \mathbb{B}^n$, output alphabet $O \subseteq \mathbb{B}^w$, a finite set of states $S \subseteq \mathbb{B}^m$, output function Λ and next state function Δ. The set $S_0 \subseteq S$ denotes the initial states. With next state function $\Delta : \mathbb{B}^n \times \mathbb{B}^m \rightarrow \mathbb{B}^m$, the transition relation of the circuit is given by

$$T(s, s') \;=\; \exists x \in \mathbb{B}^n : s' \equiv \Delta(x, s) \qquad (1)$$

A safety property $f = \mathsf{AG}(\varphi)$ can be translated to a Boolean function $[[f]]_t$, checking the validity of the formula φ at time point t. Here, the translation is done such that a satisfying assignment of $[[f]]_t$ corresponds to a counterexample of φ. The resulting function depends on the inputs, outputs and states within a bounded time interval $[0, c]$. IPC searches for counterexamples by solving the SAT instance

$$\bigwedge_{i=0}^{c} T(s^{t+i}, s^{t+i+1}) \wedge [[f]]_t \qquad (2)$$

The transition relation is unrolled within the time interval $[0, c]$ and it is connected to the single instantiation of $[[f]]_t$.

3.3. Completeness

IPC is a powerful verification technique, enabling the formalization of a specification in terms of safety properties and its verification against the implementation. However, to be sure that no bugs have been missed, the verification engineer needs to reason about the *completeness* of the written property suite. A technique to formally check whether a set of properties forms a complete specification is described in [11]. Applications of the method on industrial processor designs can be found in [5]. Automatic completeness analysis integrated within an IPC verification environment is commercially available in [12].

Completeness analysis determines whether every possible input scenario—corresponding to a transaction sequence of the design—can be covered by a chain of properties that predicts the value of states and outputs at every point in time. In other words, any two designs fulfilling all the properties of a complete property suite are formally equivalent. The completeness analysis basically boils down to check in the end state of each property whether (1) there is always a successor property with matching assumptions, (2) the successor property is uniquely determined and (3) each property describes the outputs and states of the DUV in a unique way. For a detailed description of the methodology please refer to [5], [11].

3.4. Verification Language

The properties presented here are written in the ITL language [4]. In ITL, temporal logic expressions are used to describe the behavior of a synchronous sequential system. The discrete time steps correspond to the clock cycles of the described system. Figure 2 shows an example of a simple ITL property. Usually, the properties have an implication structure: if the expressions in the *assume* part evaluate to true, then the expressions in the *prove* part must hold as well. In the *freeze* section, expressions can be assigned to variables that are fixed to a certain time point, i.e. the freeze variable `c1` refers to the value of signal `c` at time point $t + 1$, no matter in which temporal context it is used. In the temporal expressions, the standard operators of the respective HDL language (VHDL or Verilog) can be used, as well as the operators `next` and `prev`, which shift the enclosed expression relatively one cycle into the future or the past, respectively.

Thus, the first line in the prove part of Figure 2 states that at time point $t + 1$, the value of signal `c` must have increased by one compared to the previous time step. The next line shows an equivalent expression, making use of the freeze variable `c1`.

ITL also supports macro functions, which are a powerful mechanism to achieve abstraction and write high-level properties. Furthermore, all data types of the respective HDL are supported, including arrays as well as user defined types and nested record data types in VHDL.

4. Generating the Instruction Set Simulator

With the techniques presented above, it is possible to perform a complete verification even for industrial designs [4], [5]. If the verification is completed successfully, the property suite forms a model of the verified design, i.e. the properties describe the transitions and the output behavior of the design in a unique way. This fact and the use of abstraction in the verification can be exploited to obtain an

```
property simple;
freeze:
c1 = c@t+1;

assume:
at t: reset = '0';

prove:
at t+1: c  = prev(c) + 1;
at t:   c1 = c + 1;
end property;
```

Figure 2. Simple ITL Property

executable model that captures the entire behavior of the design: a simulator.

In this section the generation of the ISS is described. First we discuss the techniques applied during the verification, which are used to define an abstract state of the design, corresponding to the architectural state of a processor. Then, in Section 4.2 it is shown how the property suite can be formulated to allow for an automatic translation into an equivalent C++ program. The translation itself is presented in Section 4.3.

4.1. Abstraction

As stated in section 3.2, after having completed the verification, the properties form a model of the verified design. However, the equivalence between the properties and the design under verification (DUV) is not achieved by simply reimplementing the complex logic of the circuit in the verification language ITL. Instead, the properties are formulated in a compact and readable form by making use of abstraction techniques.

The view of the DUV that is expressed by the properties is a *high-level operation view*. For a processor, an operation naturally corresponds to the execution of a single instruction. Thus, each property describes the change of the internal state and the behavior of the output signals, when the processor executes an instruction. The state of the DUV is described in terms of a high-level or *architectural* state, which corresponds to a programmer's view on the visible registers of the design. This abstraction is achieved by the use of mapping functions.

As an example, consider the register file of a pipelined processor. The behavior of the implementation registers in the design may depend on several instructions that are currently processed by the pipeline. Consequently, the mapping function that links the architectural state of the register file to the implementation captures the forwarding logic of the pipeline. Note that these mapping functions are still much more compact than the implementation (see also [5]).

By using these mapping functions as representatives of the state of the design, the operation properties strongly resemble a high level specification. As an example, Figure 3 shows the ITL property for an `ADD` instruction of a simple

```
property instrADD;
freeze:
opcode = instr(15 downto 11)@t, // decoding of
regA   = instr(10 downto  8)@t, // instruction
regB   = instr( 7 downto  5)@t, // word
regD   = instr( 4 downto  2)@t;

assume:
at t: opcode = ADD_op; // processor ready
at t: not stall;       // to execute
at t: not interrupt;   // instruction ADD

prove:
at t: write_reg(regD, vreg(regA) + vreg(regB));
[...]
end property;

macro write_reg(i, res: unsigned): boolean :=
  foreach k in 0..7:
    if (k = i) then
      next(vreg(k)) = res(15 downto 0);
    else
      next(vreg(k)) = vreg(k);
    end if;
  end foreach;
end macro;
```

Figure 3. Operation Property

pipelined processor. In the *freeze* section of the property, the instruction word is decomposed according to the specification into the opcode and the addresses of source and target registers. The assume part states that there is actually an `ADD` instruction, and that the processor is ready to execute it. Under these constraints it is proved that one time step later the register file is updated with the correct value. The forwarding logic of the pipeline is hidden in the mapping function *vreg* which represents the architectural register file. Due to completeness requirements, the property also needs to claim that the remaining registers will not change their value—this is included in the `write_reg` macro—and it must specify the output behavior of the processor. The latter statements are omitted in the figure.

4.2. Architectural Style Properties

The basis for an ISS is an architecture description; such a description mainly consists of an architectural state and a next state function describing the effect of each of the instructions and interrupts on this state. For a set of operation properties as illustrated in Figure 3, the architecture description is rather implicit; hence, a generic and fully automatic extraction from the property would be quite hard. We therefore focus on reformulating the properties in a so-called architectural style which allows for a straightforward generation of the ISS; also supporting operation properties would be a possible extension of this work after showing the feasibility of the approach.

Note that the reformulated property is checked against the RTL and the automatic gap-detection is executed as well.

Hence, any possible discrepancy between the architectural and operation properties are identified automatically and, in particular, the reformulated property set is still equivalent to the design. An architectural style verification is characterized by:

1) Explicit modeling of the architectural state and the interfaces to memories or ports
2) Explicit definition of a macro *next_state* capturing the effects of instructions and interrupts on the architectural state
3) Explicit definition of macros that capture the behavior of the interfaces to memories and ports
4) Explicit definition of the architectural reset state

Note that if the verification has been carried out in architectural style to begin with, the ISS can be generated from the verification without any manual steps in between.

The reformulation of operation properties does not require any new detailed consideration of the design's behavior. The identification of the components of the architectural state and the precise description of the semantics of the instructions is carried out in the verification phase, anyway. In fact, one of the most sophisticated parts of the verification is to find the appropriate mapping functions for the architectural state; these mapping functions are not needed at all in order to automatically generate the ISS.

We only describe the main technical aspects of the formulation in an architectural style in the following. The architectural state is established in terms of a user defined VHDL record data type. This record combines all parts of the architectural state. Typically, this includes a register file as well as status flags and a program counter of the processor. In the same way, new data types describing the interfaces to memories and ports are defined.

One essential aspect for the modeling of the ISS in ITL is the newly introduced keyword *update*, which allows for the explicit definition of a write access to an array or record data structure. As an example, Figure 4 shows part of the *next_state* function for a simple processor. Here, the variable `state` of type *state_t* is the record holding all the elements of the architectural state. In another record `iw` of type *instruction_t*, the decoded fields of the current instruction word are kept. Using this information, the repeated decoding of the same instruction can be avoided in the simulator (see Section 4.3). The return value of the macro *next_state* is the architectural state modified by the execution of the current instruction. The execution itself is modeled in a case block, stating that when the opcode refers to an `ADD` instruction, the register file of the current architectural state will be updated by the sum of the source operands. Hence, the *next_state* functions forms the core of the ISA.

The intuition of the equivalence proof between RTL and ISA is shown in Figure 5. The dashed arrows represent the abstraction function $vstate$ that maps the implementation state of the CPU to the state of the ISA. The aim is to prove that applying the mapping $vstate$ to the implementation state

```
macro next_state(state: state_t;
                 iw: instruction_t): state_t :=
  case iw.opcode is
  when ADD_op =>
      update( state.register(iw.regD),
                ( state.register(iw.regA) +
                  state.register(iw.regB) ) );
  [...]
  end case;
end macro;
```

Figure 4. Next State Function

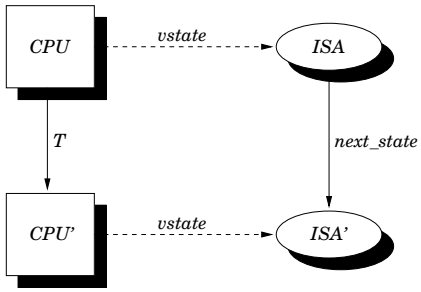

Figure 5. Structure of Equivalence Proof

and then mapping the resulting architectural state ISA to the new architectural state ISA' using $next\_state$ corresponds to applying the transition relation T on the CPU and then mapping the resulting implementation state CPU' to the architectural state. Furthermore it has to be proved that the interface signals of the design actually behave as captured by the interface macros, and that the implementation reset state of the design complies with the defined architectural reset state.

4.3. Generation of the ISS

The outcome of the reformulation in architectural style is a single property capturing all of the behavior of the verified design by making use of the *next_state* function (see Figure 6). In other words, one obtains a formally checkable ISA description. This is the starting point for the generation of the C++ instruction set simulator. In this section we discuss how the ISS is automatically generated by translating the property.

Note that for the generation of the ISS, it is *not* required to carry out the full equivalence proof between ISA and RTL upfront or even to identify the mapping functions between the architectural state and the implementation state. Instead, it is possible to develop the ISA at the beginning of the design process in ITL and use it for an early generation of the ISS. This ISS only needs to be generated again in case the ISA is updated, for example because a bug is found in the ISA during the formal verification. Therefore, a generated ISS is already available when the verification starts; the full confidence that the ISS complies to the design is achieved additionally at the end of the verification.

```
property isa;
freeze:
instr       = decode(instruction) @ t,
isa_state   = vstate @ t,
isa_state_p = vstate @ t+1,
nstate      = next_state(isa_state, instr) @ t;

assume:
at t: not stall; // ready to start instruction

prove:
at t+1: isa_state_p = nstate;
[...]
end property;
```

Figure 6. Architectural Style Property

The core of the ISS is a C++ class `Sim`. It contains all of the code for the execution of instructions and it holds the architectural state. Now that the ITL description is already in a form similar to a procedural language, the translation of the functionality is straightforward. It comprises the following steps:

1) Generate public functions for *next_state*, *decode* and the interface macros
2) Generate private functions for all remaining macros
3) Generate a member variable for the architectural state
4) Replace ITL/HDL data types and operations by C++ types and operations
5) Replace *update* expressions by a direct array/struct overwrite in C++

In order to achieve a simulation performance comparable to custom state-of-the-art simulators, several optimizations are applied during the generation of the C++ code. As long as the bit size of the ITL/HDL data types is less than or equal to the bit width of the hosting system, native data types like unsigned integers are used. Only the remaining large bit vectors are replaced by dedicated data structures. Similarly, all basic operations, like integer arithmetic and simple logic operations, are mapped to the corresponding native C++ operations. For more complex operations like bit slicing or bit rotation, as well as for operations on large bit vectors, a library with optimized functions that implement the ITL/HDL operators is used.

Furthermore, an analysis of the data dependencies in the macro functions is performed in order to identify shared expressions. With this information, additional member variables can be inserted in the simulator class in order to hold temporary results and to cache intermediate results of the computations, speeding up the simulation.

Finally, a technique similar to the just-in-time compiled simulation (see Section 3.1) is used. As described above, there is a *decode* macro which decomposes the instruction word into several bit fields according to the specification. These bit fields are stored in a record data type. By caching the results of the decode function, it is possible to avoid the repeated instruction decoding during the simulation. Due to the locality of most software—caused e.g. by loop constructs—this is a very efficient technique to decrease simulation run-time.

The generated C++ class forms the core functionality of the ISS. Besides this, the user has to provide a suitable wrapper, which calls the generated public functions of the simulator class to trigger the execution of the single instructions. Moreover, the wrapper is used to connect the simulation core to peripheral components like external memories or buses. It is also possible to integrate the simulation core with commercial simulation and debugging tools, as has been done for the experiments that are discussed in the next section.

5. Experimental Results

For a first evaluation of the proposed method, an ISS was generated for a small pipelined processor. The CPU contains a total of 8 registers with 16 bit and a special register for an interrupt return vector. It is implemented with a 5 stage pipeline. The data memory is connected to the CPU via a simple interface. The instruction set is made up of 7 instructions, including logic, arithmetic, memory access and jump instructions.

For comparison, an ISS for the processor was built using a commercial tool. This tool also provides a wrapper that supplies the simulation kernel with the instructions, as well as a graphical debugging interface. The ISS generated with our approach has also been integrated with this environment.

As for the commercial tool, the ISS showed an average performance of 0.22 million instructions per second (MIPS) using an interpretive approach, while a just-in-time compiled simulator achieved 14 MIPS. The ISS generated from the property suite showed a performance of 7 MIPS. This shows that our method clearly outperforms interpretive simulators—emphasizing the effectiveness of the optimization techniques described in Section 4.3—while it reaches about 50% of the performance of a state-of-the-art JIT-CS simulation tool.

As a second experiment, an ISS was generated for an industrial processor design. The CPU contains a total of 64 registers of 32 bit in multiple hardware contexts and offers a number of advanced processor features. It is implemented as a 7 stage pipeline and connects to a data memory and a bus interface. It is capable of executing 88 different instructions based on the DLX instruction set architecture [13]. The source code of the processor core adds up to about 10,000 lines of VHDL code, while the final reformulated property suite has a size of 2,000 lines of ITL. The property suite and its completeness were successfully checked against the processor design using [12] and thus can be considered a correct and complete specification.

Like for the first experiment, another ISS was implemented using the commercial tool suite. It showed an average performance of 2.5 MIPS (just-in-time compiled

Table 1. Performance of Different ISS

Design	Interpretive	JIT-CS	Generated
P1	0.22 MIPS	14.0 MIPS	7.0 MIPS
P2	-	2.5 MIPS	1.2 MIPS

simulation), while the ISS that was generated using our approach reached 1.2 MIPS. This confirms the results of the first experiment, i.e. that the generated ISS shows a performance comparable to modern custom made instruction set simulators. The results are summarized in Table 1, showing the simulation performance for the simple processor (P1) and the industrial design (P2) in terms of MIPS.

We presume that the custom JIT-CS simulators are still faster than our solution due to technical issues. The commercial tools that were used to build the JIT-CS simulator are subject to a great amount of optimizations, while we provided here the basic methodology of ISS generation as a proof of concept. Furthermore, the properties that the ISS is generated from reflect all hardware and pipeline effects that may not be included in a high level ISA description and that decrease the simulation performance.

6. Conclusions

In this paper, we have presented an approach to use a gap-free, i.e., complete property suite, formulated in an architectural style, to generate a C++ instruction set simulator. The generation makes use of the fact that after the successful formal verification, the property suite forms an architectural model of the verified design and hence, the ISS is equivalent to the design by construction. By applying a number of optimizations during the generation of the C++ code, the performance of the resulting simulator is comparable to state-of-the-art commercial tools.

As formal verification is increasingly used for industrial designs, the presented method is a practical way to obtain a proven correct and efficient instruction set simulator. It allows for an automatic adaptation of the simulator, when parts of the design or the specification need to be changed in a late phase of the design process. Using the formal property suite as a single source for the specification ensures correct results for early software development based on instruction set simulation.

Acknowledgment

This research work was supported by the German Federal Ministry of Education and Research (BMBF) in the Project HERKULES under the contract number 01M3082.

References

[1] E. Schnarr, M. Hill, and J. Larus, "Facile: a language and compiler for high-performance processor simulators," in *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation (PLDI)*, 2001, pp. 321–331.

[2] G. Braun, A. Nohl, A. Hoffmann, O. Schliebusch, R. Leupers, and H. Meyr, "A universal technique for fast and flexible instruction-set architecture simulation," *IEEE Trans. on CAD*, vol. 23, no. 12, pp. 1625–1639, 2004.

[3] R. Leupers, J. Elste, and B. Landwehr, "Generation of interpretive and compiled instruction set simulators," in *Proceedings of the ASP-DAC*, 1999.

[4] K. Winkelmann, H.-J. Trylus, D. Stoffel, and G. Fey, "Cost-efficient block verification for a UMTS up-link chip-rate coprocessor," in *DATE*, vol. 1, 2004, pp. 162–167.

[5] J. Bormann, S. Beyer, A. Maggiore, M. Siegel, S. Skalberg, T. Blackmore, and F. Bruno, "Complete formal verification of TriCore2 and other processors," in *Design and Verification Conference (DVCon)*, 2007.

[6] M. Nguyen, M. Thalmaier, M. Wedler, J. Bormann, D. Stoffel, and W. Kunz, "Unbounded protocol compliance verification using interval property checking with invariants," *IEEE Trans. on CAD*, vol. 27, no. 11, pp. 2068–2082, Nov. 2008.

[7] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 1579. Springer Verlag, 1999, pp. 193–207.

[8] A. Chattopadhyay, A. Sinha, D. Zhang, R. Leupers, G. Ascheid, and H. Meyr, "Integrated verification approach during ADL-driven processor design," in *IEEE International Workshop on Rapid System Prototyping (RSP)*, 2006, pp. 110–118.

[9] M. Schickel, V. Nimbler, M. Braun, and H. Eveking, "An efficient synthesis method for property-based design in formal verification: On consistency and completeness of property-sets," in *Advances in Design and Specification Languages for Embedded Systems*, S. Huss, Ed. Kluwer Acad. Publishers, 2006, pp. 163–182.

[10] S. Weber, M. W. Moskewicz, M. Gries, C. Sauer, and K. Keutzer, "Fast cycle-accurate simulation and instruction set generation for constraint-based descriptions of programmable architectures," in *International Conference on Hardware/Software Codesign (CODES)*, September 2004, pp. 18–23. [Online]. Available: http://www.gigascale.org/pubs/566.html

[11] J. Bormann and H. Busch, "Method for determining the quality of a set of properties, applicable for the verification and specification of circuits," Patent, 2007, european Patent Application, publication number EP1764715.

[12] OneSpin Solutions GmbH, Munich, Germany, *OneSpin Verification Solutions*, http://www.onespin-solutions.com, 2009.

[13] P. M. Sailer, P. M. Sailer, and D. R. Kaeli, *The DLX Instruction Set Architecture Handbook*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996.