Challenges in the Formal Verification of Complete State-of-the-Art Processors

Nathaniel Ayewah, Nikhil Kikkeri and Peter-Michael Seidel
Southern Methodist University, Dallas, TX 75205
Computer Science and Engineering
{ayewah,nikhil,seidel}@engr.smu.edu

Sven Beyer*
OneSpin Solutions GmbH, Munich
Sven.Beyer@onespin-solutions.com

Abstract— **Research on formal hardware verification has made steady progress in developing methodologies and tools that try to cope with the growing complexities of digital systems. Despite of case studies that demonstrate the applicability of formal methods to selected contemporary processor design strategies and aspects of industrial design efforts, the current state in formal hardware verification is far from being considered practical for digital systems of the complexity of complete contemporary processor designs.**

It is our goal to improve the practicality of current formal verification methods for complete state-of-the-art processor designs. The recent success in the complete formal verification of the VAMP (Verified Architecture MicroProcessor) can be considered pioneering for reaching design complexities close to this range. We dissect the VAMP verification effort in detail with the goal to identify the main technical and organizational challenges and the major productivity bottlenecks of the verification process. This is done in particular to search for opportunities of increased levels of automation. As part of our efforts we are developing the VAMPExplorer, a tool that provides an intuitive interface to the specification, the implementation and the verification of the VAMP. The VAMPExplorer visualizes the general implementation and verification structure and simplifies quick location and comparison of code fragments for improved accessibility to expert and non-expert users.

I. Introduction

Most current digital systems are far too complex to guarantee their correct design just by non-formal verification techniques such as exhaustive simulation. Formal verification techniques can, in principle, offer better scalability of the verification effort with the circuit size and it seems that recent advances in formal verification methodologies and tools have enabled the applicability to more practical design options. Various projects have been successful in showing the correctness of aspects of processor designs using formal techniques (e.g. [11], [12], [16], [19], [34], [45], [52]). However, most studies focus on high-level descriptions or on selected subcomponents of a system and make idealistic assumptions about some parts of the design. Details of the implementation are usually ignored and hardly any project design is synthesized or fully implemented. Formal verification efforts are also increasingly finding their way into commercial design efforts (e.g. [1], [15], [26], [37], [39], [43]). But even if in these cases the corresponding designs have an implementation, it is not usually the implementation that is formally verified, but it is some high-level description or only parts of the design. A full consideration of all system levels from specification to a synthesizable implementation description is necessary to consider it the complete verification of a processor. The authors of [32] have been generalizing the limitations of current verification techniques by stating that the complete functional verification of microprocessor designs could not even be achieved. Although this statement does not hold in this generality as shown by the recent success in the complete verification effort of the VAMP it is generally assumed that complete formal verification is still far from being considered practical for designs of the complexity of contemporary microprocessors.

It is our goal to improve the practicality of current formal verification methods for complete state-of-the-art processor designs. The complete formal verification of the VAMP can be considered pioneering for reaching design complexities close to this range. Therefore, we dissect the VAMP verification effort in detail with the goal to identify the main technical and organizational challenges and the major productivity bottlenecks of the verification process. The VAMP has been verified using PVS [40] (Prototype Verification System). This theorem proving based process has involved a high degree of interactive intervention. A premier motivation in our analysis and presentation of the VAMP verification effort is the search for opportunities of increased levels of automation and the increased awareness and communication about verified open source reference designs and challenges in complete processor verification. As part of this effort we are developing the VAMPExplorer, a tool that provides an intuitive interface to the specification, the implementation and the verification of the VAMP. The VAMPExplorer visualizes the general implementation and verification structure and simplifies quick location and comparison of code fragments for improved accessibility to expert and non-expert users.

In Section II we provide a detailed description of the functionality and the implementation of the VAMP. In Section III we overview alternative processor verification efforts and discuss their restrictions. In section IV we dissect the challenges in the VAMP verification effort by identifying the difficulties, outlining opportunities for automation while spotlighting selected challenges, and rating the difficulties of

*Work funded by the German Federal Ministry of Education, Science, Research and Technology (BMBF) in the framework of the Verisoft [50] project under grant 01 IS C38. The work reported here was done while the author was with Saarland University, Saarbrücken, Germany

generalizations of the VAMP verification effort. In section V we give a short overview of our efforts in developing the VAMPExplorer, before we finally conclude in section VI.

II. Detailed Description of the VAMP

A. Functionality - Instruction Set Architecture

The VAMP is a 32-bit RISC CPU with full DLX instruction set [38] including extensions for single and double precision IEEE 754 [22] floating-point operations, resulting in about 100 instructions. Nested precise interrupts are supported; a subset of these interrupts is maskable. The VAMP features a delayed PC [28] architecture, i.e., any update to the program counter does not take effect until *one* instruction after the modification. In other words, a queue of 2 PCs is part of the programmers view of the VAMP. Memory accesses are register relative, i.e., the effective memory address is computed as the sum of some register value and an immediate constant, and access widths $d \in \{1, 2, 4, 8\}$ bytes are supported. Load operations from memory can be signed or unsigned. The VAMP contains three register files, one for 32 general purpose registers, one for floating point data with 32 single precision register which can also be accessed as 16 double precision registers, and one register file for special interrupt or floating point registers, e.g., interrupt mask bits *SR* or the rounding mode RM. The `gcc` and `glibc` have been ported to the VAMP [36].

Floating point interrupts are supported in compliance with the IEEE FP standard 754 [22], which requires their accumulation in 5 bits of a special purpose register *IEEEf* (IEEE flag). Since new exception bits have to be OR-ed to the *IEEEf* register for every FPU instruction, this means that *IEEEf* is formally both source *and* destination of any floating point instruction. The IEEE standard also requires the rounding mode RM and the interrupt mask bits *SR* to be source operands of floating point instructions. Hence, with two additional double-precision operands, there is a total of seven 32-bit source operands for a single instruction in the VAMP. In addition, there are up to three results given by a double precision result and the *IEEEf* register. Note that each instruction also computes some exception flags called ECA and exception data, $EData$, which determine interrupt behavior. In case of memory instructions, e.g., exception data equals the effective memory address. The values of ECA and $EData$ of an instruction are considered to be additional results; this makes up for a total of five results per instruction.

B. Microarchitecture Overview

The VAMP [6], [8] data paths are illustrated in figure 1. Stages IF and ID—instruction fetch and decode—realize a pipelined implementation of delayed PC [38]. The remaining pipeline stages implement a Tomasulo scheduler [51] with five execution units and a reorder buffer for precise interrupts. The memory unit *MU* [7] handles instruction fetch and load/store operations, the fixed point unit *XPU* allows for the standard ALU and shift operations. Finally, there are three specialized floating point units [2], [3], [23]–[25], one for addition and subtraction, one for multiplication and division, and the last one handles the remaining test and convert operations.

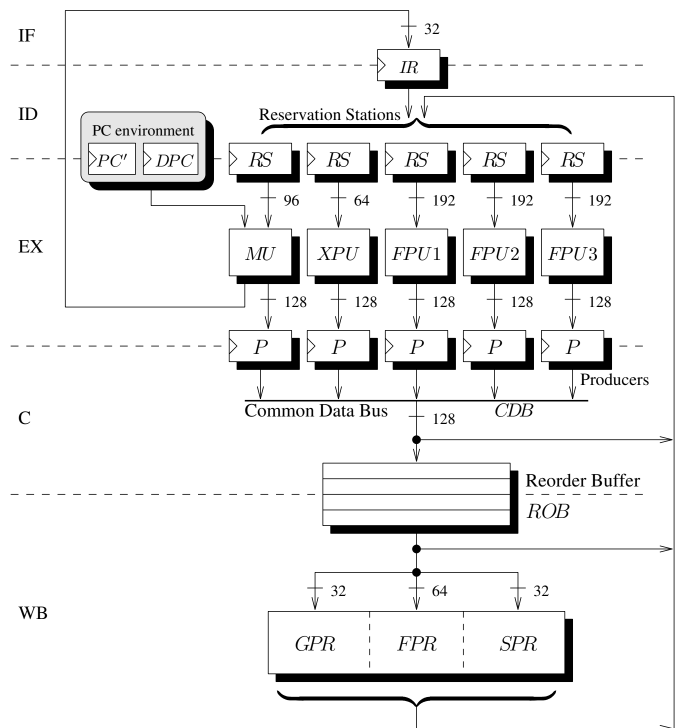

Fig. 1. VAMP data paths

As introduced above, register *IEEEf* is both source and destination register of any floating point instruction. Hence, only one floating point instruction could be in all the three floating point units together at any time if *IEEEf* would be handled by the standard Tomasulo scheduler. Therefore, a different implementation for register *IEEEf* has been chosen which achieves far better performance; this also decreases the number of source operands in the VAMP Tomasulo scheduler to six and the number of results to four. The improved *IEEEf* implementation works as follows:

- Floating point instructions just compute their four “normal” results with the Tomasulo scheduler without requiring *IEEEf* as source operand.
- When a floating point instruction is about to write its results back to the register file and leave the pipeline, it just computes a logical OR of the current value of *IEEEf* and the five corresponding bits in the exception cause ECA which is part of the reorder buffer since it is one of the results of an instruction.
- Any move instruction that copies the content of *IEEEf* to the general purpose register file for further processing is stalled in decode until no further instruction is alive in the VAMP. This is crucial since the special treatment of *IEEEf* breaks the standard Tomasulo forwarding.

With this implementation, the size of reservation stations, producer, and reorder buffer is not affected by *IEEEf*; in addition, any floating point instruction completes just a fast as without *IEEEf* since *IEEEf* is not a source operand and thus, there is no additional stalling. Only the special moves

from *IEEEf* to the general purpose register file, which occur much more rarely, have to pay the high penalty of stalling instruction fetch until all previous instructions have left the pipeline. However, even this fact is exploited by using one of these special moves with source *IEEEf* as `sync` instruction for self-modifying code [8].

The support of multiple results per instructions in the VAMP Tomasulo scheduler is mostly straightforward. However, consider the case where an instruction in a reservation station snoops for a single precision floating point source operand that is produced as part of a double precision result. In this case, both 32-bit data words of the result carry the *same* tag; however, the reservation station needs a way to actually select the correct part of the result as its source operand. This essentially means that operands in reservation stations in the presence of mixed single and double precision instructions need an additional bit in order to perform snooping correctly; this bit may be considered to be an extension of the existing tag such that any part of an instruction's result carries a separate tag. Note that only one bit is needed since the two additional results ECA and $EData$ are never forwarded.

C. The VAMP as Reference Design for Processor Verification

We think that the VAMP project [9] can be considered a reference project for processor verification for several reasons:

1) With the help of the theorem-prover PVS [13], [40], the VAMP is formally verified on the gate level, i.e., the verified PVS design is automatically translated [5] to Verilog HDL for synthesis on an FPGA [31].
2) The complete sources of both design and proofs are available online at the project website [9].
3) The VAMP offers a full DLX instruction set with about 100 instructions including floating point extensions; memory access widths of 1, 2, 4, or 8 bytes are supported.
4) The VAMP offers a state-of-the art Tomasulo scheduler with reorder buffer supporting multiple results, direct issuing into the reorder buffer, and reordering of instructions inside execution units [29]. In addition, a specially optimized *IEEEf* extension is implemented and verified [7]; these items are crucial features when effectively dealing with IEEE-compliant floating point units.
5) Design and verification are structured hierarchically resulting in many verified modules. The nice property of these modules is that they can be instantiated in order to gain the overall correctness of the complete VAMP which is synthesizable in hardware. In other words, the correctness criteria of the modules are not only what one thinks they should be, but they *formally* are what is needed for the complete verification of the VAMP.

The formal composition of the verified modules to an overall correctness proof is highly complex engineering work; however, only by actually carrying out this step can one be really sure that everything indeed fits in reality. Correctness criteria for modules that only fit together at first glance—but not formally—are of no big use. Just consider the difference between i) an aeroplane with yourself on board crashing due to some interplay of its modules that was not foreseen by its engineers because it was not modeled formally and ii) you safely landing at your desired holiday resort. Consequently, the engineering work of integrating module correctness to overall correctness is of high scientific interest. This item in particular allows for replacing any verified module by an improved implementation as long as it fulfills the same correctness criterion as the original one. On the other hand, one may simply improve the verification of the existing module by applying automated methods in order to gain the desired proof with less interaction than in the original proof.

The VAMP is obviously not intended as a prototype that demonstrates feasibility of some new verification technique or some new tool. On the contrary, the VAMP project "just" used slight refinements of existing techniques and tools and applied them in an engineering process to the complex real-life problem of formally verifying a complete microprocessor without any abstractions. We therefore suggest using real-life examples like modules of the VAMP as benchmark suites for new verification methodologies instead of some purely academic examples that often only marginally impact on real life. New techniques need not only to be able to solve the benchmarks of the VAMP theoretically, but they have to be able to cope with the concrete benchmark in its full state space down to the last bit of its definition.

However, the sheer amount of code in the VAMP project makes it really hard even for experienced engineers to actually quickly find specification, correctness criteria, and proofs of specific modules in order to use it as a benchmark. Hence, we develop the VAMPExplorer and make the VAMP benchmark suite available online such that any tool engineer can get a quick look at what real-life problems look like.

D. Design and Verification Effort

The generated Verilog sources of the VAMP contain 866 modules in more than 100000 lines of code. The PVS sources of the design alone have roughly 10000 lines of code. The VAMP contains more than 8800 register bits. The Xilinx software reports an equivalent gate count of about 1.5 million for the design that is synthesized from the verified PVS implementation description. Note that this number also covers equivalent gate counts for the instruction and data cache, i.e., 8 KB of data memory for a 2-way set-associative instruction cache and 16 KB of memory for the 4-way set-associative data cache. In addition, there is a small memory for the history information of each cache and the register files. With a cost of 4 gates per memory bit, this results in about 0.8 million gates for memories, i.e., slightly more than half the overall gate count. As a comparison, an Intel Pentium of the first generation has a gate count of about 3.1 million including 16 KB of caches, i.e., 0.5 million memory gates.

In the VAMP project, almost 90000 interactive proof steps were carried out for about 2500 lemmas; the overall design and verification effort was about eight man years.

E. *Proof structure*

The overall correctness proof of the VAMP is structured hierarchically. At the lowest level, there is the gate-level implementation of the VAMP, i.e., one argues about single gates in circuits like adders, equal testers, or multipliers, which may employ recursion. While this bit level consideration is needed in order to ensure that the implementation is correct, interactive formal arguments of this level tend to be far from efficient. With the help of a basic circuits library [4], the level of abstraction is therefore carried to some term-level. Hence, instead of arguing on the bit level about, e.g., recursive implementations of adders, one mostly argues about numbers, e.g., addition modulo the respective power of 2. This allows for much more efficient reasoning on the term-level. About 5% of the overall proof effort were spent on this step.

The abstraction and the verification for the above term level concerns purely combinational circuits. As a next step, complete execution units are verified and abstracted from with uninterpreted functions. The floating point units, e.g., are represented as a simple function allowing reordering of instructions and returning the correct data of instructions with respect to the IEEE standard. A formal description of this abstraction function for execution units can be found in [3], [24]. In order to actually verify that the complex pipeline implementation of the floating point units satisfy this function, additional modules and levels of hierarchy are identified and integrated into an overall FPU proof in [23], [24]. Similarly, the layering for the memory unit and its verification is reported in [7], [8]. Carrying out this module verification and replacing the execution units by their specification function leads to an abstract implementation of the overall processor with uninterpreted functions as execution units. This abstract implementation is then sufficiently compact to allow for efficient formal arguments on the overall correctness. More than 60% of the proof effort was spent for the complex execution units.

The verification of a parameterized Tomasulo algorithm constitutes the next module. For a meaningful correctness criterion with respect to a sequential execution model, the concept of *scheduling functions* [7], [8], [38] is employed in order to map pipeline stages in hardware cycles to instructions in the sequential execution model. Tomasulo algorithm correctness is based on the formal proof presented in [29]; however, its interface needs to *exactly* match the definitions of the abstract execution units and to be able to cope with the *IEEEf* extension in order to allow for instantiation for the VAMP. Details on these extensions of the proof are reported in [7, Chap. 4]. For the right version of the Tomasulo algorithm, about 15% of the verification effort was spent.

In the next step, the actual data management of the pipeline is verified in the absence of interrupts, i.e., the abstract pipeline implementation is mapped to a programmer's model without interrupts which carries out one instruction per step. In other words, the Tomasulo correctness proof [29] is instantiated with the given abstract execution units. In addition, the data management concerns the computation of the next value of the program counter and the correctness of the decoding mechanism which is mostly straightforward. As a last step, the correctness of instruction fetch needs to be established. Instantiating the Tomasulo algorithm proof correctly with the complete instruction set and taking care of instruction fetch and the program counter computation was just as complex as the proof of the Tomasulo algorithm itself.

In the final step, interrupt support is added. Therefore, the interrupt verification techniques reported in [7], [8] are applied to the abstract pipeline which is already correct with respect to a programmer's model without interrupts as outlined above. This final step amounted to about 3% of the proofs.

III. Other Processor Verification Projects

Most of the features of the VAMPhave also been covered by other verification projects; however, most of these projects only focus on single modules, make heavy restrictions, or use strong abstractions without giving a gate-level implementation together with a proof that it fulfills the abstraction. Work on the formal verification of processors so far has concentrated mainly on the following aspects of architectures:

1) Microprocessors with in-order scheduling, one or several pipelines including result forwarding, stalling, and interrupt mechanisms [12], [27], [54]. The verification of the very simple, non-pipelined FM9001 processor is reported in [10], [11]. Using the flushing method from [12] and uninterpreted functions for modeling functional units, superscalar processors with multicycle execution units, exceptions and branch prediction [54] have been verified by automatic BDD based methods. Also, one can transform specification machines into simple pipelines (with forwarding and stalling mechanism) by an automatic transformation, and automatically generate formal correctness proofs for this transformation [30].
2) Tomasulo schedulers with reorder buffers for the support of precise interrupts [16], [20], [44]. Exploiting symmetries, McMillan [34] has shown the correctness of a powerful Tomasulo scheduler with a remarkable degree of automation. Using theorem proving, Sawada and Hunt [21], [44], [45] show the correctness of an entire out-of-order processor, precise interrupts, and a store buffer for the memory unit. They also consider self-modifying code (by means of a *sync* instruction).
3) Floating point units. The correctness of an important collection of floating point algorithms is shown in [42], [43] using the theorem proving system ACL2. Using a combination of theorem proving and model checking techniques, correctness proofs for the floating point units of Pentium processors are reported in [14], [39]. Based on the constructions and on the paper and pencil proofs in [38] a fully IEEE compatible floating point unit has been verified [3], [24], [25] (using mostly but not exclusively theorem proving). In [48] and [26] the verification of fused-multiply-add FPUs is reported.
4) Caches. Multiple cache coherence protocols have been formally verified, e.g., [17], [35], [47], [49]. Paper and

pencil proofs are error prone, and hence the generation of proofs for interactive theorem proving systems is slow. The method of choice is model checking. The compositional techniques by McMillan [35] even allow for the verification of parameterized designs, i.e., cache coherence is shown for an arbitrary number of processors.

Except for the work on floating point units, the cache coherence protocol in [17], and the FM9001 processor [11], *none* of the papers quoted above states that the verified design has been implemented. *All* results cited above except [3], [11], [17], [24], [25] use several simplifications and abstractions:

1) The realized instruction set is restricted: always included are the six instructions considered in [12]: load word, store word, jump, branch equal zero, ALU register operations, ALU immediate operations. Five typical extra instructions are trap, return from exception, move to and from special registers, and sync [44]. The branch equal zero instruction is generalized in [54] by an uninterpreted test function. Most notably the verification of machines with load/store operations on half words and bytes has apparently not been reported. In [53] the authors report an attempt to handle these instructions by automatic methods which was unsuccessful due to memory overflow.
2) Delayed branch is replaced by non-deterministic speculation (speculating branch taken/not taken).
3) Sometimes, non-implementable constructs are used in the processors: e.g., Hosabettu et.al. [20] use tags from an infinite set. Obviously, this is not directly synthesizable.
4) The verification of Intel's and AMD's FPUs does neither cover the handling of denormal numbers nor of exception flags. The verification of a dual precision FPU has not been reported (though, obviously, Intel's and AMD's FPUs are capable of dual precision). IBM's FPU verification [26] does handle denormal numbers, exception flags, and dual precision, but does not cover the verification of the multiplier array.
5) No verification of a memory unit with caches has been reported. Eiriksson [17] only reports the verification of a bit-level implementation of a cache coherence protocol without data consistency.
6) The highly automated Tomasulo proof of McMillan [34] neither covers multiple results sharing the same tag, specialized execution units or reservation stations, nor an *IEEEf* extension, direct issuing into the reorder buffer, nor a special treatment for register $R0$ which always contains 0 in the VAMP.
7) The verification of pipelines or Tomasulo schedulers with *instantiated* floating point units and memory units with caches and main memory bus protocol has not been reported apart from the VAMP project [7]. Indeed, in [53] the authors state: "An area of future work will be to prove that the correctness of an abstract term-level model implies the correctness of the original bit-level design."

IV. Challenges in the Verification of the VAMP

A. Difficulties

The aim of the VAMP project was the design and verification of a complete state-of-the-art microprocessor without any abstractions or restrictions. Most of the difficulties faced in the course of the project stemmed from this fact.

From the beginning, intense cooperation between the project members was needed in oder to define interfaces between different modules that not only looked good, but would actually allow for their formal integration into the overall correctness proof later on. In addition, the interfaces were chosen with design efficiency in mind, e.g., the number and width of operands and results was parameterized and instruction reordering in execution units was allowed. This required early proof sketches on paper and pencil. Different modules were then designed and verified in parallel, e.g. the floating point units, the memory unit, and the Tomasulo algorithm. In order to increase the efficiency, the published designs from [38] were improved upon, e.g., by adding a Tomasulo scheduler, stalling instructions only when absolutely necessary and allowing instructions to overtake each other, and using write-back policy for the data cache. Since synthesis of the design had been one of the initial project objectives, all these modules had to be verified down to the gate level.

When all the different modules were integrated to the VAMP proof, the full DLX instruction set with about 100 instructions was realized. This made the instantiation with a corresponding decoding module for all instructions a non-trivial process as suggested by the effort it actually took. In addition, the Tomasulo proof was extended by its special *IEEEf* treatment at this point since the trivial implementation cannot be considered to be state-of-the-art by any means.

Finally, the VAMP project also carried PVS to its limits concerning complexity. Type checking of the overall proof alone took two hours of CPU time in 2003; actually rerunning all the proofs several days.

B. Automation

The VAMP project basically employed purely interactive proofs in PVS with little use of strategies. Although version 2.3 of PVS that was used in the project features a model-checker, it was only used three times to show liveness of the pipelined floating point units [23]. However, the overhead for these three commands was huge:

- Part of the design and its properties were translated to μ-calculus since this is the model-checker's input language.
- The lemmas that formally relate the μ-calculus results to their actual temporal version used in the interactive proof part were proved in PVS.
- Since the model-checker of PVS 2.3 does not even return a counter-example in case of failure, the verification was carried out in SMV [33] and only after establishing correctness there, the PVS model-checker was used.

In summary, only one automated tool, e.g., the built-in model-checker, is available in PVS 2.3, and even this automated tool

cannot really be used efficiently. Additionally, PVS 2.3 is a closed system and the integration of more powerful automated tools was therefore not an option in the VAMP project. Hence the almost exclusive interactive proof effort.

However, 90000 interactive proof steps are obviously too much for the VAMP project. We believe that with integration of the right automated tools, at least three quarters of the interactive proof steps will become obsolete since they seem trivial. Only with this gain in productivity, the verification of entire systems will become feasible, e.g., the combination of hardware and system software needed in order to provide virtual memory to user processes [18] which is a crucial step towards the verification of entire computer systems.

C. Selected Challenges for the Automated Verification Community

In this section we outline selected challenges that should be addressed with formal verification techniques at a high degree of automation. We consider solutions to these challenges as a key enabler for increased productivity in formal microprocessor verification of the complexity of the VAMP. In order to formulate these challenges we introduce some notation.

We denote configurations of the implementation in hardware cycle t by c_I^t and configurations of the specification before the execution of instruction i by c_S^i. Components of configurations are accessed like records, i.e., $c_I.IR$ denotes the instruction register in the implementation and $c_S.M$ the memory in the specification. Scheduling functions map pipeline stages in hardware cycles to instructions, i.e., the hardware stage k in cycle t contains instruction $i := sI(k,t)$. If an instruction in stage k' in cycle t moves to stage k, we define $sI(k,t+1) := sI(k',t)$. Further details on the recursive definition of scheduling functions are given in [7], [8]. In summary, scheduling functions are basically infinite tags that are clocked through the pipeline together with the corresponding instruction. Note, however, that these infinite tags are only employed in the proof and are not part of the fully synthesizable implementation. For a byte addressable memory M, we denote with $M_n[a]$ the n bytes $M[a+n-1],\ldots,M[a]$. We denote correctness of the VAMP without interrupts in cycle t by $corr?(t)$.

The easiest target for proof automation is the purely combinational step from bit-level to term-level, e.g., the correctness of adders, decoders, and multipliers in the contexts `basics` and `dlxalu` as well as the equality of the overall term-level implementation of the VAMP and its gate-level implementation from theory `tom_impl_spec_correct`. However, even with full automation, this covers only 5% of the overall proof.

The verification of the Tomasulo algorithm with *IEEEf* extensions from context `tomasulo` is carried out on a fairly abstract level with many uninterpreted functions. However, induction is used extensively in these proofs. Nevertheless, for the specific set of parameters that were actually instantiated in the VAMP, a high degree of automation also seems possible for this proof due to the level of abstraction.

As a next candidate for a high degree of automation, we consider the implementation of the Tomasulo algorithm from context `dlxtom`. Most lemmas in this module assume equivalence of the current VAMP and Tomasulo configuration and show that some signals are equal or part of that the equivalence still holds in the following cycle. After expanding both configurations of the following cycle, this also results in purely combinational properties that could be easily automated. In addition, however, there are some lemmas concerning the correctness of memory and instruction fetch that are not part of the Tomasulo proof. These lemmas often argue about an arbitrary number of hardware cycles and sometimes require induction proofs. As a case in point, consider the correctness of instruction fetch. Here, we have to argue that for code with `sync` instructions the VAMP really fetches the instruction that the sequential model is expected to execute next, i.e., there is no pending store to this address in the pipeline.

Lemma 1 *Let $corr?(t)$ hold, let no interrupt occur in cycle t and let there be `sync` instructions in the code, i.e., $synced_code?$ holds. We set $i := sI(dec, t+1)$. Instruction fetch is then correct, i.e.,*

$$c_I^{t+1}.IR = \begin{cases} c_S^i.M_4[c_S^i.DPC] & \textit{if } c_S^i.DPC \bmod 4 = 0 \\ 0^{32} & \textit{otherwise} \end{cases}$$

Note that in order to prove the above lemma in PVS, three additional helper lemmas are needed in PVS; one of these lemmas even uses induction. Ideally, an automated method could handle this lemma without the need to manually split it into four lemmas. Alternatively, the correctness of the memory is given by the following lemma. Note that the complexity of this lemma stems from the fact that we allow both floating point stores that access 4 or 8 bytes of memory as well as integer stores accessing 1, 2, or 4 bytes.

Lemma 2 *Let $corr?(t)$ hold and let no interrupt occur in cycle t. The memory is then correct in cycle $t+1$, i.e., $c_I^{t+1}.M = c_S^{sI(M,t+1)}.M$.*

Any automation of these lemmas by sophisticated methods would be highly welcome, even if they only involve partial automation. The same holds for the integration of interrupts in the correctness proof. We are convinced that the corresponding 18% of the proof could be automated to a high degree. As an example for the integration of interrupts, consider the synchronicity of interrupts with respect to the memory. Similar to the correctness of instruction fetch, five helper lemmas are actually needed in PVS in order to prove the following claim and one of them uses induction.

Lemma 3 *Let $corr?(t)$ hold and let an interrupt occur in cycle t. The memory is then correct with interrupts in cycle $t+1$, i.e., $c_I^{t+1}.M = c_S^{sI(wb,t+1)}.M$.*

The lion's share of the VAMP proof effort was spent in the verification of the memory unit with its caches and the

three floating point units, i.e., more than 60%. Many inductive proofs are carried out for the different units. A high degree of automation seems most complex in this part of the proof. On the other hand, there are also many combinational or simple one-cycle lemmas for the execution units similar to the term-level and the instantiation of the Tomasulo algorithm for the top-level proof. Hence, a degree of automation of 50% or more for the execution units does not seem unrealistic.

D. *Generalization to the Verification of Complete state-of-the-art processors*

In the VAMP project, a very specific processor has been completely formally verified. However, the design is cleanly modularized into the core, the execution units, and the memory system. Therefore, it is possible to plug in a different memory system or more optimized execution units by only verifying their local correctness and combining it with the remaining part of the original VAMP correctness proof. In addition, the memory system and its proof is parameterized, e.g., on the associativity and size of the caches. Both the modularization and the parameterization make the VAMP proof to some extent generic. In the VAMP, there are five specific execution units with a total of eight reservation stations which can hold up to six source operands and instructions deliver up to four results, whereas the proof of the Tomasulo algorithm is parameterized over these four numbers. Hence, a different number of, e.g., execution units or operands would require additional effort in the verification of the actual *implementation*—which is *not* parameterized—with respect to the Tomasulo algorithm. A different number of source operands or results and a new execution unit would be needed, e.g., in order to handle fused multiply-add instructions or MMX and SIMD-extensions.

Since the VAMP proof is modularized, it is possible to switch the VAMP core with some other scheduling algorithm. Note, however, that this would require the proof of a gate-level *implementation* of the scheduling algorithm, and not only a proof of the algorithm. This also applies to superscalar cores.

The VAMP uses the delayed PC scheme for instruction fetch. Integrating branch prediction instead would require a considerable effort on the core proof since the correctness of instruction cancellation is proved for interrupts only as the very last step of the overall proof. Cancellation due to misprediction, however, happens frequently even without interrupts, i.e., at the lowest level of the proof.

While the VAMP memory system is well parameterized, there are some extensions that would require a large effort. Consider, e.g., non-blocking caches which also require a new optimized control and thus, the whole memory system proof needs to be redone. On the other hand, for the addition of store buffers, the existing memory system could just be re-used; some additional effort would only be needed to show the correctness of forwarding from the store buffers and the synchronicity of interrupts with respect to the visible memory.

As long as general performance optimizations keep the overall structure of the VAMP, i.e., the existing pipeline stages, an adaption of the proof to the optimized version is

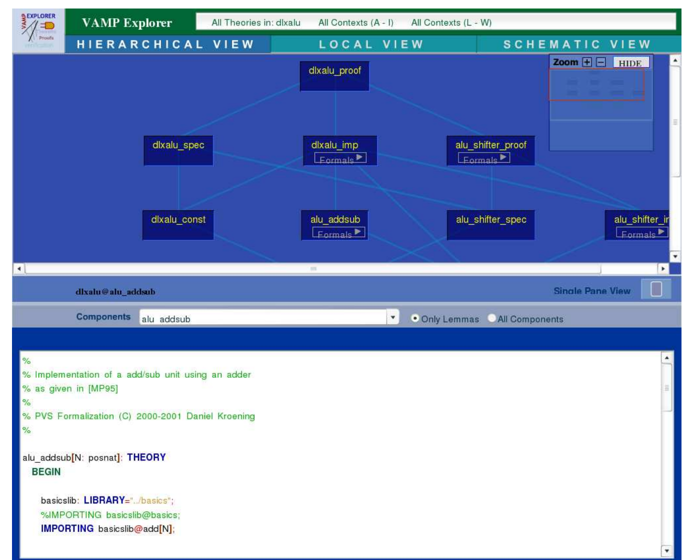

Fig. 2. Screen-shot of the current VAMPExplorer

straightforward and fairly easy to carry out. However, re-balancing the design by introducing an additional pipeline stage, e.g., for timing or power saving reasons, or even aiming for deep super-pipelining as introduced in [41] would require a huge proof effort.

V. The VAMPExplorer: Visualizing the VAMP

The complexity of the VAMP specification, implementation and verification makes it difficult to identify structural information or to navigate through specific details within an unformatted plain text description as in conventional PVS files. Even the additional formatting functionality provided in the PVS environment does not significantly improve the accessibility to the structures of the VAMP. The goal is to support establishing the open source VAMP descriptions as a reference design for complete processor verification to a broad audience of various backgrounds and encourage modular verification of selected properties/components with alternative, preferably automated, verification techniques. It is therefore the goal of our visualization efforts to make the VAMP specification, implementation and verification details better accessible and easier to understand. The depth of the hierarchy and the level of connectivity of the VAMP files complicate this task. We have a focus on providing an intuitive graphical interface for identifying and navigating through VAMP theories and their relations while allowing concurrent access and summary of different aspects (specification, PVS implementation, synthesized Verilog Implementation, properties and related proofs) of a single component. In this way PVS-specific statements are also related to traditional conceptions of hardware design in the form of schematics or Verilog code.

The complexity of the hierarchies makes it necessary that a high degree of automation is involved in the data and property extraction from the PVS files and the preparation of the dynamic hierarchical structural views. For broad accessibility we chose an intuitive web-based user interface. Some information can already be automatically extracted from the VAMP

files using commands provided by PVS itself. Unfortunately, these commands do not yield sufficiently detailed semantic information such as the relationship between synthesizable components or the classification of a particular declaration that is needed for the VAMPExplorer. We therefore implemented our own parser using the grammar that comes with PVS. In this way we can extract all the information relevant for the VAMPExplorer based on the syntax trees and store this information in XML format which is supported by many visualization engines.

We chose to use Macromedia Flash as our web front-end because of its broad web accessibility and its support for rapid prototyping and dynamic content. The XML output of the parser is directly used by our Flash interface. The visualization uses a directed graph in order to show the relationships between theories or groups of theories as shown in Fig. 2. The interface also provides facilities that allow users to compare information in different components. The system is available online at [46].

VI. Conclusion

It is our goal to improve the practicality of complete formal verification for complex state-of-the-art processors. For this purpose we discuss the details and and the generalization of challenges that occur in the complete formal verification effort of the VAMP as one of the pioneering examples in complete formal processor verification. We suggest the use of the VAMP sources as a reference design and benchmark suite for the (automatic) formal verification of components as they are integrated in a complex modular processor design. For improved accessibility we develop the VAMPExplorer GUI for an intuitive access to the VAMP specification, implementation and verification details. Future work includes extensions of the VAMPExplorer to enhanced automatically generated schematic views and the handling of even more complex real-live proof structures as they occur for example in combined hardware-software systems [50].

References

[1] S. Ben-David, C. Eisner, D. Geist, and Y. Wolfsthal, "Model Checking at IBM," in *Formal Methods in Systems Design*. Kluwer Academic Publishers, 2003, pp. 101–108.

[2] C. Berg, "Formal verification of an IEEE floating point adder," Master's thesis, Saarland University, Germany, 2001.

[3] C. Berg and C. Jacobi, "Formal verification of the VAMP floating point unit," in *CHARME*, ser. LNCS, vol. 2144. Springer, 2001, pp. 325–339.

[4] C. Berg, C. Jacobi, and D. Kröning, "Formal verification of a basic circuits library," in *IASTED International Conference on Applied Informatics*. ACTA Press, 2001.

[5] S. Beyer, C. Jacobi, D. Kröning, and D. Leinenbach, "Correct hardware by synthesis from PVS," 2002, internal Report, available at http://www-wjp.cs.uni-sb.de/publikationen/BJKL02.pdf.

[6] S. Beyer, C. Jacobi, D. Kröning, D. Leinenbach, and W. Paul, "Instantiating uninterpreted functional units and memory system: functional verification of the VAMP," in *CHARME*, ser. LNCS, vol. 2860. Springer, 2003, pp. 51–65.

[7] S. Beyer, "Putting it all together – formal verification of the VAMP," Ph.D. dissertation, Saarland University, Germany, 2005.

[8] S. Beyer, C. Jacobi, D. Kröning, D. Leinenbach, and W. Paul, "Putting it all together - formal verification of the VAMP," *International Journal of Software Tools for Technology Transfer, Special Issue on 'Recent Advances in Harware Verification' (to appear)*, 2005.

[9] S. Beyer, C. Jacobi, D. Leinenbach, and W. J. Paul, "The VAMP project," Website, 2003, http://www-wjp.cs.uni-sb.de/projects/verification.

[10] B. C. Brock and W. A. Hunt, "The DUAL-EVAL hardware description language and its use in the formal specification and verification of the FM9001 microprocessor," *Formal Methods in System Design*, vol. 11, pp. 71–107, July 1997.

[11] B. C. Brock, W. A. Hunt, and M. Kaufmann, "The FM9001 microprocessor proof," Computational Logic Inc., Tech. Rep. 86, 1994.

[12] J. R. Burch and D. L. Dill, "Automatic verification of pipelined microprocessors control," in *CAV*, ser. LNCS, vol. 818. Springer, 1994, pp. 68–80.

[13] R. W. Butler, P. S. Miner, M. K. Srivas, D. A. Greve, and S. P. Miller, "A bitvectors library for PVS," NASA Langley Research Center, Tech. Rep. 110274, Aug 1996.

[14] Y.-A. Chen, E. M. Clarke, P.-H. Ho, Y. Hoskote, T. Kam, M. Khaira, J. W. O'Leary, and X. Zhao, "Verification of all circuits in a floating-point unit using word-level model checking," in *FMCAD*, ser. LNCS, vol. 1166. Springer, 1996, pp. 19–33.

[15] M. Cornea-Hasegan, "IA-64 floating point operations and the IEEE standard for binary floating-point arithmetic," *Intel Technology Journal*, Q4, 1999.

[16] W. Damm and A. Pnueli, "Verifying out-of-order executions," in *CHARME*. Chapman & Hall, 1997, pp. 23–47.

[17] A. P. Eiriksson, "The formal design of 1M-gate ASICs," in *FMCAD*, ser. LNCS, vol. 1522. Springer, 1998, pp. 49–63.

[18] M. Hillebrand, "Address spaces and virtual memory: Specification, implementation, and correctnesss (under appraisal, draft available at www-wjp.cs.uni-sb.de/publikationen/hil05.pdf)," Ph.D. dissertation, Saarland University, Germany, 2005.

[19] R. Hosabettu, "Systematic verification of pipelined microprocessors," Ph.D. dissertation, University of Utah, CS Department, 2000.

[20] R. Hosabettu, M. Srivas, and G. Gopalakrishnan, "Proof of correctness of a processor with reorder buffer using the completion functions approach," in *CAV*, ser. LNCS, vol. 1633. Springer, 1999, pp. 47–59.

[21] W. A. Hunt and J. Sawada, "Verifying the FM9801 microarchitecture," *IEEE Micro*, pp. 47–55, May-June 1999.

[22] *ANSI/IEEE standard 754–1985, IEEE Standard for Binary Floating-Point Arithmetic*, Institute of Electrical and Electronics Engineers, 1985.

[23] C. Jacobi, "Formal verification of complex out-of-order pipelines by combining model-checking and theorem-proving," in *CAV*, ser. LNCS, vol. 2404. Springer, 2002.

[24] ——, "Formal verificaton of a fully IEEE compliant floating point unit," Ph.D. dissertation, Saarland University, Germany, 2002.

[25] C. Jacobi and C. Berg, "Formal verification of the VAMP floating point unit," in *Formal Methods in System Design*. Kluwer Academic Publishers, 2005, to appear.

[26] C. Jacobi, K. Weber, V. Paruthi, and J. Baumgartner, "Automatic formal verification of fused-multiply-add FPUs," in *DATE*. IEEE Computer Society, 2005, pp. 1298–1303.

[27] D. Kröning, S. Müller, and W. Paul, "Proving the correctness of pipelined micro-architectures," in *3ITG-/GI/GMM-Workshop Methoden und Beschreibungsprachen zur Modellierung und Verifikation von Schaltungen und System*. VDE Verlag, 2000, pp. 89–98.

[28] ——, "Proving the correctness of processors with delayed branch using delayed PCs," *Numbers, Information and Complexity*, pp. 579–588, 2000.

[29] D. Kröning, "Formal verification of pipelined microprocessors," Ph.D. dissertation, Saarland University, Germany, 2001.

[30] D. Kröning and W. Paul, "Automated pipeline design," in *DAC*. ACM Press, 2001, pp. 810–815.

[31] D. Leinenbach, "Implementierung eines maschinell verifizierten Prozessors," Master's thesis, Saarland University, Germany, 2002.

[32] S. Mangelsdorf, R. Gratias, R. Blumberg, and R. Bhatia, "Functional verification of the HP PA 8000 processor," *Hewlett-Packard Journal*, vol. 1-13, 1997.

[33] K. McMillan, *Symbolic model checking*. Kluwer, 1993.

[34] ——, "Verification of an implementation of Tomasulo's algorithm by compositional model checking," in *CAV*, ser. LNCS, vol. 1427. Springer, 1998.

[35] ——, "Parameterized verification of the FLASH cache coherence protocol by compositional model checking," in *CHARME*, ser. LNCS, vol. 2144. Springer, 2001.

[36] C. Meyer, "Entwicklung einer Laufzeitumgebung für den VAMP-Prozessor," Master's thesis, Saarland University, Germany, 2002.

[37] J. S. Moore, T. Lynch, and M. Kaufmann, "A mechanically checked proof of the AMD5K86 floating point division program," *IEEE Transactions on Computers*, vol. 47, no. 9, pp. 913–926, 1998.

[38] S. M. Müller and W. J. Paul, *Computer Architecture. Complexity and Correctness*. Springer, 2000.

[39] J. O'Leary, X. Zhao, R. Gerth, and C.-J. H. Seger, "Formally verifying IEEE compliance of floating-point hardware," *Intel Technology Journal*, Q1, 1999.

[40] S. Owre, N. Shankar, and J. M. Rushby, "PVS: A prototype verification system," in *CADE 11*, ser. LNAI, vol. 607. Springer, 1992, pp. 748–752.

[41] J. Preiß, "Complexity and correctness of a super-pipelined processor," Ph.D. dissertation, Saarland University, Germany, 2005.

[42] D. M. Russinoff, "A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7 processor," *LMS Journal of Computation and Mathematics*, vol. 1, pp. 148–200, 1998.

[43] ——, "A case study in formal verification of register-transfer logic with ACL2: The floating point adder of the AMD Athlon processor," in *FMCAD*, ser. LNCS, vol. 1954. Springer, 2000.

[44] J. Sawada and W. A. Hunt, "Processor verification with precise exceptions and speculative execution," in *CAV*, ser. LNCS, vol. 1427. Springer, 1998.

[45] ——, "Verification of the FM9801 microprocessor: An out-of-order microprocessor model with speculative execution, exceptions, and self-modifying code," *Formal Methods in Systems Design*, vol. 20, no. 2, pp. 187–222, March 2002.

[46] P.-M. Seidel and N. Ayewah, "The VAMP explorer," Website, 2005, http://www.natidea.com/projects/VAMPExplorer/.

[47] X. Shen, Arvind, and L. Rudolph, "CACHET: an adaptive cache coherence protocol for distributed shared-memory systems," in *International Conference on Supercomputing*. ACM Press, 1999, pp. 135–144.

[48] A. Slobodova and K. Nagalla, "Formal verification of floating point multiply add on Itanium processors," in *Workshop on Designing Correct Circuits*, Mar. 2004.

[49] J. Stoy, X. Shen, and Arvind, "Proofs of correctness of cache-coherence protocols," in *FME*, ser. LNCS, vol. 2021. Springer, 2001.

[50] The Verisoft Consortium, "The Verisoft project," Website, 2003, http://www.verisoft.de.

[51] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," in *IBM Journal of Research and Development*. IBM, 1967, vol. 11 (1), pp. 25–33.

[52] M. N. Velev, "Comparative study of strategies for formal verification of high-level processors," in *ICCD*, 2004, pp. 119–124.

[53] M. N. Velev and R. E. Bryant, "Superscalar processor verification using efficient reductions of the logic of equality with uninterpreted functions to propositional logic," in *CHARME*, ser. LNCS, vol. 1703, 1999.

[54] ——, "Formal verification of superscale microprocessors with multicycle functional units, exception, and branch prediction," in *DAC*. ACM, 2000.