Automatic Formal Correspondence Checking of ISA and RTL Microprocessor Description

Lukáš Charvát, Aleš Smrčka, and Tomáš Vojnar
Brno University of Technology, FIT, IT4Innovations Centre of Excellence, Božetěchova 2, CZ-61266 Brno, Czech republic
{icharvat,smrcka,vojnar}@fit.vutbr.cz

***Abstract*—The paper proposes an automated approach with a formal basis designed for checking correspondence between an RTL implementation of a microprocessor and a description of its instruction set architecture (ISA). The goals of the approach are to find bugs not discovered by functional verification, to minimize user intervention in the verification process, and to provide a developer with practical results within a short period of time. The main idea is to use bounded model checking to check that the output produced by automatically derived RTL and ISA models of a given processor are the same for each instruction and each possible input. Although the approach does not provide full formal verification, experiments with the approach confirm that due to a different way it explores the state space of the design under test, it can find bugs not found by functional verification, and is thus a useful complement to functional verification.**

I. Introduction

As the complexity of hardware is growing over the last decades, automation of its development is crucial. To facilitate the automation, high-level models are used increasingly during the design process. Specifically, in the case of microprocessor design, various tool chains, such as LISA [1] or Codasip [2], [3], [4], take advantage of the availability of high- and low-level descriptions and provide automatic generation of HDL designs, simulators, assemblers, disassemblers, and compilers.

In the current microprocessor design tool chains, verification of the designs is typically performed by simulation and/or functional verification (testing). Simulation is commonly used to obtain some initial understanding about the design (e.g., to see whether ISA contains sufficient instructions) or to check the performance of the design. Functional verification requires a golden specification, which must be provided manually by the developers. Moreover, even extensive functional verification can miss many deeper bugs. Therefore, a use of formal verification is desirable even if it is applied in a bounded way. Unfortunately, formal verification is not a common part of the current microprocessor design tool chains.

In this paper, we propose an automated approach built on a formal basis and intended to be used within an automated microprocessor design tool chain for checking correspondence between an RTL implementation of a microprocessor and a description of its instruction set architecture (ISA). Our approach is original in its very high level of automation: the only user inputs are an RTL implementation, an ISA description (possibly complemented by a specification of assumed restrictions on the possible values of instruction operands), and a time limit for the verification.

The main idea behind our approach is to use bounded model checking (BMC) to compare the outputs produced by automatically derived RTL and ISA models of a given processor for all possible instructions and their inputs. In order to guarantee that some useful result is obtained in a given time limit, each instruction is checked in parallel for several bit-widths of its input, and the maximum bit-width for which a result is obtained in the given time limit is used.

Since our approach uses BMC, it does not consider any mutual influence among the instructions, and it can limit the bit-width of input data, it does not provide full formal verification. However, our experience shows that the approach is complementary to functional verification, and due to a different way of exploring the state space of the verified design, it can find bugs not found by functional verification.

The approach has been implemented within the Codasip Studio IDE [2] and successfully tested on several case studies. The experiments included a non-trivial real-life single-pipelined processor in which, during its development, our approach revealed three previously unknown bugs confirmed by the developers. The experiments have shown that almost every instruction of a simple pipeline processor (of a form commonly used in light-weight embedded devices) is verified within seconds. Shortened input data were used only in a few cases, typically for instructions such as multiplication (and even in such cases, one can argue that most typical bugs would anyway manifest even for shortened input).

Plan of the Paper: Section II provides an overview of related work. Section III provides a background on the Codasip design flow for which our approach is optimized. The main idea of the proposed method is described in Section IV. Section V provides more details on the way we model processors and on the actual verification process. Experiments are discussed in Section VI. Section VII concludes the work.

II. Related Work

A lot of research effort has been invested into development of formal methods for hardware verification in the past decades. When concentrating on microprocessors, theorem proving (cf., e.g., [5], [6], [7]) and automatic generation of properties satisfied by a given design (e.g., [8], [9], [10], [11]) belong among well-known and often used approaches (both of them, however, require a significant level of expertise and/or user intervention). More automation is offered by the approach of model checking based on a systematic exploration of

the state space of the verified system. Recently, the approach of bounded model checking (BMC) [12], exploring the state space of a verified system up to some depth only, and related approaches such as IPC [13] have become very popular in practice, leveraging the recent advances in automatic decision procedures, especially, SAT solvers [14], [15], [16].

Most work on automated formal verification of pipelined microprocessors based on BMC and SAT solvers can be separated in two main branches: verification of the microprocessor wrt. general properties and correspondence checking between the ISA and RTL implementation.

In [17], the author proposes general properties of the correct behavior of a typical single-pipelined implementation of a microprocessor. These properties together with an ADL description of a processor are then converted to a BMC problem to find possible counterexamples [18]. The main advantage of the approach is the overall verification time, which is about a second for a middle-sized microprocessor [17]. A disadvantage of the approach is that it produces false negatives on optimized data-paths that do not correspond to the definition of a typical pipelined processor.

Automated correspondence checking of an RTL design against a specific, formally encoded ISA description is considered, e.g., in [19]. They propose a verification approach that consists of: (1) choosing an arbitrary legal starting RTL state, (2) symbolically computing the corresponding ISA state by finishing partially executed instructions in the pipeline, (3) symbolically running one instruction in both the ISA and RTL models, and (4) comparing the programmer-visible parts of the designs. The approach exploits the logic of equality with uninterpreted functions and memories (EUFM) which allows for an abstraction of functional units and memories while completely modeling the control of a processor. In [20], EUFM is extended by positive equality of uninterpreted functions (PEUF) which greatly reduces the time needed for verification. The works [21], [22] further extend the approach by using positive equality of uninterpreted functions for modeling functional units, superscalar processors with multicycle execution units, exceptions, and branch prediction. Since the approach uses uninterpreted functions for operators unsupported by EUFM and/or PEUF, the verification may fail (or take too much time) on RTL designs with optimized operations.

Another, yet similar approach to correspondence checking of the control of a microprocessor is described in [23]. The work proposes a method of automatic formal verification of a pipelined implementation against its ISA specification by using IPC to prove that all assertions of all instructions are satisfied and to prove validity of assumptions and consequents of instructions in every possible chain of instructions. For this purpose, a mapping of high-level ISA to RTL has to be provided which, however, requires a manual user intervention.

Compared to the above approaches, our approach aims at *no user intervention* and thus minimal expertise of the user even when applying the approach on an *optimized design*. Although the approach does not provide full formal verification, it can find bugs not found by functional verification.

```
1 element reg represents regs {
2   assembler { "r" idx=unsigned };
3   binary { idx=0b[4] };
4   return { idx; };
5 }
6 element instr__add {
7   use reg as { dst, sA, sB };
8   assembler { "ADD" dst "," sA "," sB };
9   binary { 0b0101 dst sA sB };
10  semantics {
11    regs[dst] = regs[sA] + regs[sB];
12    cf = fn_add_carry(regs[sA], regs[sB]);
13  };
14 }
```

Fig. 1: A description of the `add` instruction in CodAL

III. Background: Codasip Design Flow

Our work was originally motivated by a request to provide some support for verification on a formal basis for the Codasip Studio IDE [2], but the proposed method can be used within other microprocessor development tool chains too if they are able to provide all needed information about the processor (as discussed below). Codasip aims at rapid processor development and supports a simultaneous creation of an ISA description and an RTL implementation. The ISA description (developed in the Codasip-specific CodAL language illustrated in Fig. 1) serves as an instruction-accurate model (IA), which is used to automatically generate a simulator, assembler, disassembler, and extractor of instruction semantics for the target compiler [24], [25]. The RTL design (e.g., VHDL), cycle-accurate simulator, and profiler are automatically generated from a cycle-accurate description (CA) that is developed in the CodAL language too.

Our method uses both the IA and CA descriptions to automatically perform conformance checking between them. From the instruction-accurate model, we use: (i) the set of all instructions, (ii) the binary representation of each instruction and its format (i.e., information about which bits represent the operator, operands, and immediate data), and (iii) the semantics of the instructions. From the low-level, cycle-accurate model, we use: (iv) the types of memories and register files together with the number of read and write ports and (v) the identification of the write-back pipeline stage. Furthermore, in case of processors with multicycle instructions, we need to know the maximum number of cycles each instruction needs to complete its execution.

As stated above, for our approach, it is crucial to know the set of instructions to be checked as well as their semantics. However, there is no notion of instructions in the CodAL language as can be seen in Fig. 1. Nevertheless, the assembly syntax description can be used instead. This syntax is based on a context-free grammar generating a finite language (ensured by the CodAL compiler). Hence, if all words of the language are systematically generated, a list of instructions is obtained. This extraction is supported by Codasip as a part of its automatic generator of a C compiler, which needs to know every instruction included in the instruction set of the modeled

processor. Codasip also extracts a C-language description of the behavior of each instruction and converts it to an SSA-based format with a few simple optimizations.

IV. The Main Idea of the Proposed RTL-ISA Correspondence Checking

We concentrate on checking correspondence between the behavior of an RTL design of a microprocessor and its ISA description on the level of an *independent execution* of each instruction. By the independent execution, we mean the execution of an instruction surrounded by no-operation instructions (`nop`). Hence, our approach does not aim at finding errors related to the use of pipelines, multiple ALUs, caches, etc. We, however, believe that such an approach is still useful, which is supported by our experiments described further on that allowed us to find several errors in a real-life microprocessor (not found by functional testing). Moreover, as briefly discussed in the conclusions, in the future, we plan to complement our current approach by another verification phase building on the correctness of execution of each instruction in isolation, aiming at errors related to the use of pipelines and other advanced microprocessor features (hence splitting the verification into several simpler phases).

The proposed method uses bounded model checking as an automated reasoning engine. A typical approach to use (bounded) model checking is to encode the specification (ISA in our case) as a temporal formula using the specification language of the chosen model checker. Unfortunately, for complex instructions, this is a rather complicated task. Therefore, we use a more straightforward translation of the ISA specification into a behavioral model described in the modeling (not specification) language of the model checker. We thus generate two behavioral models: namely, an RTL and ISA model of the given processor. These models are then equipped with an environment model, including architectural registers, memories, the program counter, and input/output ports. All these models are composed together, and BMC is used to check that if both of the processor models start with the same state of their environment (including the same instruction to be executed), their environments equal after the execution too. For this purpose, we have implemented an automated generator of models from ISA descriptions and translator of VHDL to RTL models, created abstract models of memories and register files, and a top-level model controlling the ISA, RTL, and environment models as well as comparing their execution.

Our approach uses similar principles as [19], but since we are interested in verification of a single instruction only, we can consider the reset state of the RTL model as a starting point. This also eliminates the need to make the symbolic execution reach in a potentially costly way the corresponding starting ISA state. The top-level control of verifying a single instruction can be summarized as follows: (1) Initialize the environment of the given RTL and ISA model. (2) Symbolically execute one cycle of the ISA model (covering all possible cases that may arise). (3) Stall the ISA model and reset the RTL model to ensure that it is in a stable state. (4) Symbolically execute the RTL model for the needed number of cycles (depending on the write-back pipeline stage or on the number of cycles of a multicycle instruction). (5) Stall the RTL model to ensure that no more changes on architectural resources are made. (6) Finally, check whether the environments of the RTL and ISA model are equal. In the first step of the initialization of the environment, the program memory is filled with an instruction to be verified, other architectural resources are left random to simulate all possible inputs for the instruction. If the environments of the RTL and ISA models are found different in Step 6, an error in the implementation of the instruction initially set in the program memory was found. In the next section, all these steps are described in more details.

V. A More Detailed Description of the Approach

A. Generation of the ISA Model

To derive the ISA model of a processor, we use the output of the Codasip semantics extractor, which consists of the instruction syntax and the semantics generated for each possible combination of operands of the instruction. The way these combinations are encoded within an instruction word is called *instruction format*. The description of the syntax includes the name of the instruction and its unique assembler and binary representation. The binary representation divides the instruction word into constant and operand parts. The constant parts are usually used to express the opcode and addressing mode, while the operand parts mark the position of the code of operands within an instruction word. The semantics description uses an SSA-based representation.

In Fig. 2, the information extracted for the `add` instruction is shown. This instruction works with three 16-bit register operands: it adds the last two (`reg1`, `reg2`) and stores the result into the first one (`reg0`). Based on the result of the addition, the carry flag (`cf`) is set. The `regop(rf, idx)` operation used on lines 4, 5, 7 represents reading/writing of a value stored at the index `idx` within the register file `rf`. The `reg(r, 0)` operation used on line 9 means reading/writing from/to the register `r` (not in a register file). The `iN` operator where `N` stands for a positive integer is a bit-width specifier. The operation `add` represents the addition itself, while `carry_add` computes the value of the carry after the addition. Auxiliary variables introduced due to using the SSA-form can be recognized by their `%` prefix.

When generating the ISA model, we translate the output described above into the Cadence SMV language [26]. This formal modeling language is used mainly because of its wide support in various model checking tools.

The ISA model is obtained by translating the semantics of each format of each instruction separately. The obtained translations are used as different branches of the ISA model. The branch to be executed is chosen according to the contents of the so-called *fetch* vector that is added to the ISA model since a description of the fetch stage is not included in the output of the semantics extractor. The value of this vector is initialized according to the instruction format (line 12 in Fig. 2) by the top-level model discussed below.

```
 1 /* Name */
 2 instr instr__add__reg__reg__reg__,
 3   /* Semantics */
 4   %tmp0 = i16 regop(regs, reg1);
 5   %tmp1 = i16 regop(regs, reg2);
 6   %tmp2 = add(%tmp0, %tmp1);
 7   regop(regs, reg0) = %tmp2;
 8   %tmp3 = carry_add(%tmp0, %tmp1);
 9   reg(cf, 0) = %tmp3;,
10   /* Syntax */
11   "ADD" reg0 "," reg1 "," reg2,
12   0b0101 reg0[3,0] reg1[3,0] reg2[3,0]
```

Fig. 2: The output from the Codasip semantics extractor for the `add` instruction

The translation of the particular instruction formats relies on the interface of the chosen model of architectural resources. We, in particular, represent single registers as binary vectors with signals we, d, and q in their interface. These signals have the same meaning as those used in a D-latch. Similarly, memories and register files with m read and n write ports are mapped to binary matrices having an interface with signals $we_0, ..., we_m$, $wa_0, ..., wa_m$, $d_0, ..., d_m$, $re_0, ..., re_n$, $ra_0, ..., ra_n$, $q_0, ..., q_n$.

The actual translation of the semantics of the particular instruction formats is then based on rewriting each operation in the semantics description into its SMV implementation. For that, we built a library of SMV implementations of all the operations that may appear in the output of the Codasip semantics extractor. Some of them are natively supported by SMV (i.e., they map to some SMV operation), some are replaced by multiple SMV operations. For an illustration of the translation, see Fig. 3 which shows the result of translating the `add` instruction. Note, e.g., the extraction of operands from the *fetch* vector (lines 12-14) and the translation of the `carry_add` operation (line 8 in Fig. 2) using the operations plus and bit extraction (lines 25, 26 in Fig. 3).

B. Modeling Large Architectural Resources

While single architectural registers or small memories can be modeled directly as binary vectors or matrices, modeling large memories or register files in such a way could lead to a state space explosion during the verification. Therefore, we use an abstraction technique similar to the one of [27]. The technique exploits the fact that the number of values stored in memory cells that must be remembered is limited wrt. the depth of the analyzed BMC problem. The interface of the abstracted memory is left the same, but internally, an access table is used. Every access, i.e., a write/read to/from the memory, is recorded in the form of an address-value pair[1]. If the memory is accessed again, its access table is searched first. If there exists a record with the given address, a value that corresponds to the address is returned/modified. Otherwise, a new record is created. Note, however, that sometimes this abstraction could use more bits than the actual implementation. Hence, a decision whether or not to use the abstraction is done

[1]A similar approach is applied when the processor uses I/O ports and buses.

```
 1 -- Variant instr__add__reg__reg__reg__
 2 -- Definitions
 3 reg0 : array 3..0 of boolean;
 4 reg1 : array 3..0 of boolean;
 5 reg2 : array 3..0 of boolean;
 6 _tmp0 : array 15..0 of boolean;
 7 _tmp1 : array 15..0 of boolean;
 8 _tmp2 : array 15..0 of boolean;
 9 _tmp3 : boolean;
10 _tr_tmp0 : array 16..0 of boolean;
11 -- Transitions
12 reg0[3..0] := fetch[11..8];
13 reg1[3..0] := fetch[7..4];
14 reg2[3..0] := fetch[3..0];
15 regs_re0 := 1;
16 regs_ra0 := reg1;
17 _tmp0 := regs_q0;
18 regs_re1 := 1;
19 regs_ra1 := reg2;
20 _tmp1 := regs_q1;
21 _tmp2 := (_tmp0 + _tmp1);
22 regs_we0 := 1;
23 regs_wa0 := reg0;
24 regs_d0 := _tmp2;
25 _tr_tmp0 := (_tmp0 + _tmp1);
26 _tmp3 := _tr_tmp0[16];
27 cf_we := 1;
28 cf_d := _tmp3;
```

Fig. 3: Instruction semantics translated to SMV

based on the knowledge of the number of state variables that are to be used in each of the cases.

Another technique that we use is data-domain reduction. The technique sacrifices soundness in favor of speed in which a potential flaw is discovered. It under-approximates the bit-width of the architectural resources by setting selected bits permanently to zero. We get back to the particular way we use data-domain reduction in Section V-D.

C. The Top-Level Model

The top-level model controls initialization, symbolic execution, and stalling of the ISA and RTL models and their environment. For that, three special variables are used: a *clock counter* and two *halt signals*. The clock counter increments its value with every cycle of the symbolic execution of ISA and RTL models. It is used for detecting the end of the verification process. The ISA and RTL halt signals are connected to every resource of the ISA and RTL models, respectively, and are used to signal them to keep their values, hence to stall the whole ISA and RTL models.

In the first step of the verification of some instruction format (to verify all formats, the verification is run for each format separately), the program memory of the RTL model is initialized such that upon the first read access, the same fetch vector that was assigned to the ISA model and that describes the instruction format chosen to be verified is read from the program memory. Further read accesses, even from the same address, will produce the fetch vector representing the `nop` instruction. This behavior ensures that the processor will execute the verified instruction only. The fetch vector is defined bit per bit according to the binary coding of the instruction (cf. line

12 in Fig. 2) in the following way: each bit corresponding to a constant (operation code or addressing mode) is set to the value of that constant, other bits are left random to simulate all possible inputs. Other architectural resources such as data memories and register files are initialized to random values which, in the initial state only, are shared by the ISA and RTL models to ensure that both models have the same inputs.

In the next step, the ISA model is symbolically executed for a single clock cycle. Since the ISA model of an instruction semantics is encoded as a function of instruction inputs, which are known after the initialization step, a single clock cycle is needed for architectural resources of the ISA model to store new values. The ISA model and its architectural resources are then stalled using the ISA halt signal, and the RTL model is reset to its initial stable state.

Next, the RTL model is symbolically executed for $t_{wb} + 1$ cycles where t_{wb} represents the write-back stage of the pipeline (or the number of cycles of a multi-cycle instruction to get to the write-back stage), and the additional clock cycle is used for architectural resources to store new values. The RTL model with its architectural resources are then stalled using the RTL halt signal to ensure that no more changes happen on the RTL level.

Finally, the results of the symbolic executions of the ISA and RTL models are checked for correspondence. Since the behavior of some instructions is defined only for a specific range of values of the operands, the correspondence is not just identity. In particular, the developers must explicitly specify which restrictions of the possible operand values they assume in a form of assertions (e.g., by some pragma in the IA model). The property expressing the required correspondence is then an invariant of the following form:

$$(clk = t_{wb} + 2) \Rightarrow (\bigwedge_{a \in A} a \Rightarrow \bigwedge_{r \in R} (r_{ISA} = r_{RTL}))$$

where clk is the clock counter, A denotes the set of restrictions on operands, R is a set of architectural resources, and r_{ISA} (r_{RTL}) represents a value of architectural resource r of the ISA model (the RTL model), respectively. The time $t_{wb}+2$ represents the overall time for symbolic executions of ISA and RTL models.

D. The Use of BMC and its Parallelization

For the actual verification of the correspondence property, we use the ability of the SMV model checker to convert a given verification problem to a BMC problem of a specified depth. In particular, in our case, the depth of the problem is $t_{wb}+2$ which is sufficient because no further changes are made to the architectural resources after that time. The problem is represented in CNF using the DIMACS format and exported to be solved using a SAT solver. It is possible to map the CNF terms back to variables of the ISA and RTL models, thus in case of a flawed RTL design, the encountered problem can be presented to the developers in terms of the original variables.

In fact, we do not generate a single BMC problem for each format of each instruction, but four of them to be solved in parallel. These four problems differ in the data-domain reduction used, in particular: no reduction, a 1/2 reduction (when 50 % of the bits of bit-vectors of memory units of data memory and register files are used only), a 1/4 reduction, and 1/8 reduction. A time limit is then applied for solving each of these problems, and the result of the lowest reduction for which the appropriate problem is solved in time is used. The time limit is derived from the overall time limit for the verification of the whole processor (given by the user) divided by the number of all formats of all instructions. This limitation ensures that the whole verification process will terminate within the specified time.

VI. Experiments

We have implemented the above described method in a prototype tool and tested it on three processors: *TinyCPU* is a small 8-bit test processor with 4 general-purpose registers and 3 instructions that we mainly used to prove our concept. *SPP* is an 8-bit ipcore with 16 general-purpose registers and a RISC instruction set consisting of 9 instructions. *Codea2* is a 16-bit processor partially based on MSP430 microcontroller developed by Texas Instruments [28]. It is equipped with 16 general-purpose registers, 15 special registers, a flag register and an instruction set including 41 instructions where each may use up to 4 available addressing modes.

Our experiments were run on a PC with Intel Core i7-3770K @3.50GHz and 32 GB RAM using Cadence SMV (build from 05-25-11) and GlueMinisat (version 2.2.5) [29] as an external SAT solver. The results can be seen in Table I. The columns represent the processor being verified, the number of instructions in its instruction set, the number of formats of all instructions (IFs), which gives the number of the (parallelized) BMC problems to be solved. The next columns represent the results obtained from the verification: the number of IFs which have been successfully verified with no data-reduction, the number of IFs which have been successfully verified at least for some data-reduction, and columns representing numbers of IFs successfully verified for a given data-reduction. Avg. time is the average time per instruction format for the verification process of a given instruction format.

The time limit was set to 3s for TinyCPU and to 15s for SPP. For SPP, the limit is close to the time that is needed for generation of the BMC problems to be solved (i.e., the time needed for the semantics extraction together with the translation to SMV and the subsequent derivation of the BMC problems in DIMACS), which took on average 1.38 s per instruction format. The average time needed for SAT solving was 0.11 s per instruction format. Pushing the limit below this bound would lead to unusable results.

To illustrate the use of the verification time limit in our approach, we provide experiments with Codea2 for two different time limits: 1200 s and 3600 s. The former is close to the bound described above (most of the time is taken by the semantics extraction, and the SMV and DIMACS translations: 3.37 s per instruction format on average). The latter

TABLE I: Verification results

Processor / time limit	No. of instructions	No. of instr. formats	Proved no reduction	Partially proved	Part. proved $1/2$ reduction	Part. proved $1/4$ reduction	Part. proved $1/8$ reduction	Avg. time
TinyCPU / 3 s	3	3	3	-	-	-	-	0.18 s
SPP / 15 s	9	9	9	-	-	-	-	1.49 s
Codea2 / 1200 s	41	319	53	266	58	176	32	3.66 s
Codea2 / 3600 s	41	319	273	46	46	-	-	6.01 s

limit leaves more time for SAT solving (2.64 s in contrast of 0.29 s per instruction format on average). As can be seen, every instruction format was proved at least for the reduction factor of 8 (for a 16 bit processor, this means that 2 bits per register have only been used). Moreover, multiplication instructions (46 instruction formats) were the only ones that were too complex to be proved fully in the extended time limit.

By verifying the development version of Codea2, we found three flaws. Each of them was related to setting the carry flag during arithmetic instructions. All the flaws were confirmed as real flaws by the processor development team.

VII. Conclusion

We have proposed a method of checking correspondence between the ISA and RTL description of a microprocessor through BMC. Despite its formal roots, the approach does not provide full formal verification since it checks each instruction in isolation and also possibly limits the bit-width of the data being manipulated. However, as confirmed by our experimental results, the approach can be still quite useful in that it can find real errors not found by functional verification (due to the different ways these approaches exercise the state space of the verified systems). Moreover, the approach is almost fully automated, hence not requiring any additional efforts from the developers (apart from possibly describing their assumptions about limited values of instruction arguments). Furthermore, the approach allows for an easy control of the verification time, exploiting parallelization in order to increase usefulness of the results that can be obtained in the given time.

For the future, we plan to complement the approach proposed in the paper by another verification phase that will build on that the instructions work correctly in isolation and will concentrate on their possibly undesirable interference in the pipeline only. This approach is motivated by trying to split the problem of processor verification into several simpler tasks.

Acknowledgement: This work was supported by the Czech Science Foundation (project P103/10/0306), the Czech Ministry of Education (projects COST OC10009, MSM 0021630528), the Czech Ministry of Industry and Trade (project FR-TI1/038), the EU/Czech IT4Innovations Centre of Excellence CZ.1.05/1.1.00/02.0070, and the BUT projects FIT-S-11-1 and FIT-S-12-1.

References

[1] 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 Transaction on CAD*, 23(12, 2004.

[2] Codasip Studio for Rapid Processor Development, www.codasip.com.

[3] Z. Přikryl, K. Masařík, T. Hruška, and A. Husár, "Fast Cycle-Accurate Interpreted Simulation", Proc. of MTV, 2009.

[4] Z. Přikryl, K. Masařík, T. Hruška, and A. Husár, "Generated Cycle-Accurate Profiler for C Language", Proc. of DSD, 2010.

[5] R. Hosabettu, G. Gopalakrishman, and M. Srivas, "Verifying Advanced Microarchitectures that Support Speculation and Exceptions", Proc. of CAV, 2000.

[6] J. Harrison, "Floating-point Verification using Theorem Proving", *Formal Methods for Hardware Verification*, LNCS 3936, 2006.

[7] S. Beyer, C. Jacobi, D. Kröning, D. Leinenbach, and W. J. Paul, "Putting it All Together—Formal Verification of the VAMP", *STTT*, 8(4–5), 2006.

[8] F. Rogin, T. Klotz, G. Fey, R. Drechsler, S. Rülke, "Automatic Generation of Complex Properties for Hardware Designs", Proc. of DATE, 2008.

[9] P. Mishra and N. Dutt, "Specification-Driven Directed Test Generation for Validation of Pipelined Processors", *ACM Transactions on Design Automation of Electronic Systems*, 13(3), 2008.

[10] T. N. Dang, A. Roychoudhury, T. Mitra, and P. Mishra, "Generating Test Programs to Cover Pipeline Interactions", Proc. of DAC, 2009.

[11] M. Chen and P. Mishra, "Property Learning Techniques for Efficient Generation of Directed Tests", *IEEE Trans. on Computers*, Vol. 60, 2011.

[12] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu, "Bounded Model Checking", *Advances in Computers*, 2003.

[13] M. Ngyuen, M. Thalmaier, M. Wedler, J. Bormann, D. Stoffel, and W. Kunz, "Unbounded Protocol Compliance Verification usign Interval Property Checking with Invariants", *IEEE Trans. on CAD*, 27(11), 2008.

[14] M. N. Velev, "Using Automatic Case Splits and Efficient CNF Translation to Guide a SAT-Solver when Formally Verifying Out-of-Order Processors", *AI&MATH*, 2004.

[15] M. N. Velev, "Efficient Translation of Boolean Formulas to CNF in Formal Verification of Microprocessors", Proc. of ASP-DAC, 2004.

[16] M. N. Velev, "Exploiting Signal Unobservability for Efficient Translation to CNF in Formal Verification of Microprocessors", Proc. of DATE, 2004.

[17] P. Mishra and N. Dutt, "Functional Verification of Programmable Embedded Architectures: A Top-Down Approach", Springer, 2005.

[18] H. Koo and P. Mishra, "Functional Test Generation Using Design and Property Decomposition Techniques", *ACM Transactions on Embedded Computing Systems*, 8(4), 2009.

[19] J. R. Burch and D. L. Dill, "Automatic Verification of Pipelined Microprocessor Control", Proc. of CAV, 1994.

[20] R. E. Bryant, S. German, and M. N. Velev, "Exploiting Positive Equality in a Logic of Equality with Uninterpreted Functions", Proc. of CAV, 1999.

[21] M. N. Velev, P. Gao, "Automated Debugging of Counterexamples in Formal Verification of Pipelined Microprocessors", ASP-DAC, 2010.

[22] M. N. Velev and P. Gao, "Automatic Formal Verification of Multi-threaded Pipelined Microprocessors", Proc. of ICCAD, 2011.

[23] U. Kühne, S. Beyer, J. Bormann, and J. Barstow, "Automated Formal Verification of Processors Based on Architectural Models", Proc. of FMCAD, 2010.

[24] A. Husár, M. Trmač, J. Hranáč, T. Hruška, K. Masařík, D. Kolář, and Z. Přikryl, "Automatic C Compiler Generation from Architecture Description Language ISAC", Proc. of MEMICS, 2010.

[25] M. Trmač, A. Husár, J. Hranáč, T. Hruška, and K. Masařík, "Instructor Selector Generation from Architecture Description", MEMICS, 2010.

[26] K. L. McMillan, Cadence SMV, www.kenmcmil.com/smv.html.

[27] M. K. Ganai, A. Gupta, P. Ashar, "Verification of Embedded Memory Systems using Efficient Memory Modeling", Proc. of DATE, 2005.

[28] Codea2 Core IP in Codasip Studio, www.codasip.com/products/codea2/.

[29] H. Naneshima, K. Iwanuma and K. Inoue, GlueMinisat, appeared in *SAT Competition 2011*, sites.google.com/a/nabelab.org/glueminisat/, 2011.