HADES: Microprocessor Hazard Analysis via Formal Verification of Parameterized Systems

Lukáš Charvát Aleš Smrčka Tomáš Vojnar
Brno University of Technology, FIT, IT4Innovations Centre of Excellence
Božetěchova 2, 612 66 Brno, Czech Republic
{icharvat,smrcka,vojnar}@fit.vutbr.cz

HADES[1] is a fully automated verification tool for pipeline-based microprocessors that aims at flaws caused by improperly handled data hazards. It focuses on single-pipeline microprocessors designed at the register transfer level (RTL) and deals with read-after-write, write-after-write, and write-after-read hazards. HADES combines several techniques, including data-flow analysis, error pattern matching, SMT solving, and abstract regular model checking. It has been successfully tested on several microprocessors for embedded applications.

1 Introduction

Implementation of pipeline-based execution of instructions in purpose-specific microprocessors, often used, e.g., in embedded applications, is an error-prone task, which implies a need of proper verification of the resulting designs. Formal verification of such microprocessors—despite they are much simpler than common processors for mainstream computing—is a very challenging task. One way how to deal with it is to develop a set of verification techniques specialised in checking absence of a certain kind of errors in such microprocessors. Here, the main idea is that, this way, a high degree of automation and scalability can be achieved since only parts of a design related to a specific error are to be investigated. The above idea has been followed, e.g., in the works [6, 7] that proposed fully automated approaches for (1) checking correctness of individual execution of processor instructions and (2) for verifying absence of read-after-write (RAW) hazards when the instructions are pipelined. In [8], the approach was extended by covering write-after-write (WAW) and write-after-read (WAR) hazards.

To be more precise, an *RAW hazard* arises when an instruction writes to a storage that some later instruction reads, but it is possible for the later instruction to read an old value being rewritten by the earlier instruction. A *WAW hazard* refers to a situation when an instruction writes to a storage and rewrites a result stored by some later instruction which already finished its execution. A *WAR hazard* arises when a later instruction write to a destination before it is read by the previous instruction. There are also non-data hazards. *Structural hazards* deal with sharing resources by instructions in a pipeline. *Control hazards* arise when an instruction is executed improperly due to an unfinished update of a program counter. This paper, however, concentrates on data hazards only.

In particular, the paper presents the HADES tool, developed by VeriFIT research group at FIT BUT, that implements a slightly improved version of the approaches proposed in [7, 8]. Namely, after briefly discussing related works, we specify how the input of HADES looks like, we describe its architecture and implementation, and provide experimental results on a larger set of microprocessors than in [7, 8]. Moreover, we include a more detailed discussion of the needed verification time and its decomposition to the computing times needed by the different analysis phases implemented in HADES. We close the paper by a discussion of possible future improvements of the HADES tool.

[1] www.fit.vutbr.cz/research/groups/verifit/tools/hades/

J. Bouda et al. (Eds.): 11th Doctoral Workshop
on Mathematical and Engineering Methods
in Computer Science (MEMICS 2016)
EPTCS 233, 2016, pp. 87–93, doi:10.4204/EPTCS.233.9

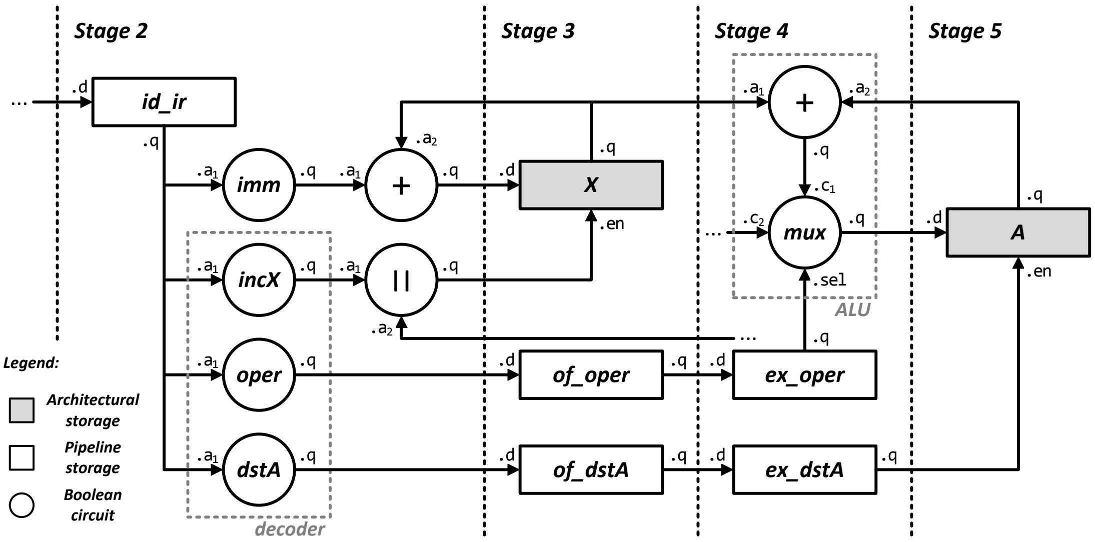

Figure 1: A processor structure graph of a part of a CPU with an accumulator architecture.

Related Work. Verifying that there are no hazards in a pipelined microprocessor is quite crucial. Hence, it has become a native part of checking conformance between an RTL design and a formally encoded description of an instruction set architecture (ISA), and many approaches with formal roots have been proposed for this purpose. Among them, one can find, e.g., the following approaches [5, 13, 1, 14, 15, 20, 12]. However, these methods typically require a significant manual user intervention—either in a form of specifying the consistent state of the microprocessor or defining predicates describing pipeline behaviour. Compared with such approaches, HADES does not aim at full conformance checking of RTL and ISA implementations. Instead, it addresses one specific property—namely, absence of problems caused by pipeline hazards. On the other hand, HADES is almost fully automated—the user is required to identify the architectural resources (such as registers and memory ports) and the program counter only.

2 Input Models

HADES focuses on microprocessors with a single pipeline and in-order execution. The tool expects storages (registers and memories) to have a unit write and zero read delay. Multicycle delay storages can be easily simulated by a chain of unit storages. The tool also assumes that pipeline internal registers which carry data interchanged between programmer visible storages are controlled by stall and clear signals.

The tool expects the processor under verification to be described by a so-called *processor structure graph* (PSG in short) which represents the internal structure of the processor. A PSG is an oriented graph that consists of vertices (storages or boolean circuits) and edges (control and data connections). An example of a simple PSG is depicted in Figure 1. It shows a part of a simple microprocessor with an accumulator architecture with two architectural registers: *X* (a memory index register) and *A* (an accumulator). For the sake of brevity, the PSG does not exhibit control connections of pipeline registers. In the CPU, an instruction fetched from the memory is stored into the storage *id_ir* representing the instruction register. The decoder determines the type of the operation of arithmetic logic unit and identifies its destination by activating the appropriate enable connection (`en`) of the X or A register. An early auto-increment of register X can be performed in stage 3. Such a feature allows the CPU to execute sequences of instructions working with juxtaposed data in the memory without a penalty (brought, e.g., by unnecessary stalls of the pipeline) which would be present if the update of X was done in a later stage.

A design of a processor on the register transfer level (RTL) written in a common hardware design language like VHDL or Verilog can be easily converted into a PSG.

3 The Verification Approach of HADES

The verification approach of HADES was proposed in [7, 8]. It leverages the current advances in SMT solvers for bit-vector logic and in formal verification of systems with a parameterized number of processes—for short, referred to as *parameterized systems* (PSs) below. The main idea is to reduce the problem of finding hazards that may arise when executing an in advance unknown number of in advance unknown instructions[2] to a parametric verification problem where the successive instructions are modelled by processes, which gradually pass through the processor. In particular, it turns out that one can use the common notion of PSs operating on a linear topology where the processes (i.e., instructions being executed) may perform local transitions or universally/existentially guarded global transitions [9, 18, 2].

More precisely, the approach consists of the following steps: (1) a data-flow analysis intended to distinguish particular stages of the pipeline, (2) a consistency check of a correct implementation of the particular pipeline stages, (3) a static analysis identifying constraints over data-paths of instructions that can potentially cause data hazards, (4) generation of a PS modelling mutual interaction between potentially conflicting instructions, and (5) an analysis of the constructed parameterized system.

Identification of Pipeline Stages. A simple data flow analysis is used to derive the number of pipeline stages implemented in a given processor and to assign storages and logic functions into the pipeline stages. A *pipeline stage* is defined as a sub-graph of the PSG responsible for executing a single-cycle step of an instruction. The pipeline stage of a PSG vertex (representing some storage or function) is given by the minimum number of cycles needed to propagate data from the input of the program counter (assumed to be in a fictive stage 0) to the output of the given vertex.

Consistency Checking. The second step of the method is consistency checking that checks whether the flow logic assures a correct in-order execution of all instructions through all the identified pipeline stages. This step checks whether the flow logic obeys a set of rules that express how the control connections (i.e., enable, stall, and clear signals) of storages in adjacent pipeline stages should be set. In short, the rules require that an instruction carried by a pipeline stage cannot be fragmented, duplicated, or lost. In particular, a strengthened variant of the rules proposed in [16] is used.

Static Detection of Potential Hazards. Next, a static hazard analysis over the PSG with annotated pipeline stages is performed to identify a finite set of so-called *hazard cases*. Each hazard case describes one possible source of a hazard. A hazard case consists of a programmer visible source storage (i.e., a register or a writing port of the memory), target storage, reading and writing stages, and an influence path describing how data propagate between the stages. Since the definition of a hazard case speaks about

[2] Note that one cannot simply restrict the checking to a number of instructions given by the number of pipeline stages since the processor can get to different internal states after having processed some number of instructions of some kind.

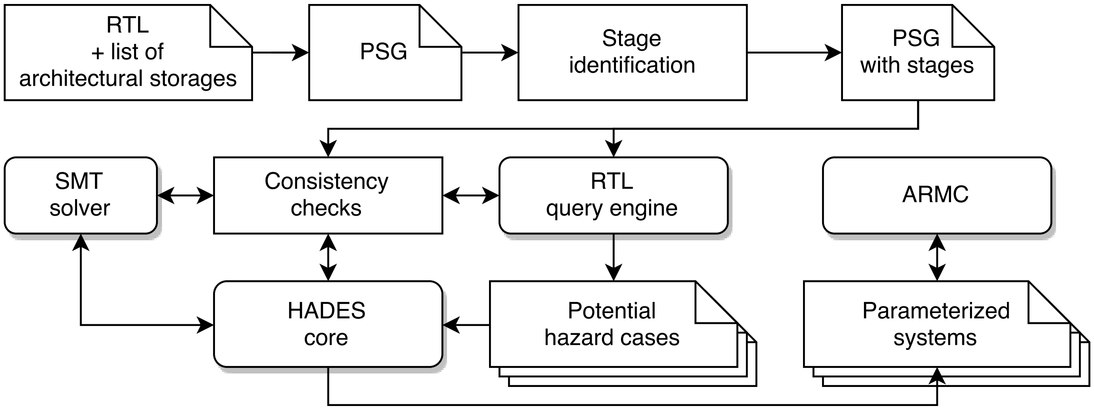

Figure 2: HADES architecture.

storages, their access stages, and the path along which the problematic data is transferred, it is not related to a single instruction only but to an entire class of instructions.

Generation of PSs Modelling the Possible Hazards. In this stage, a PS for each identified hazard case is generated. The main component of the PS is a finite automaton whose instances represent instructions passing the pipeline. A state of the automaton identifies the class of instructions that the particular instance represents[3], the execution stage into which the instruction got, and the conditions that must hold for the instruction to proceed such that a flow of data along the path associated with the given hazard case is caused. The transitions of the automata can be guarded by referring to the states of the automata representing instructions that surround the given instruction in the pipeline. Their generation is pruned by checking whether the conditions behind the states of the involved automata do not exclude each other. Further, regular sets of initial and bad configurations are generated. Initial configurations represent simply an arbitrary sequence of instructions waiting for entry into the pipeline. Bad configurations are specified separately for the different types of hazards considered—e.g., for RAW hazards, they say that a later instruction finished reading before an earlier instruction committed writing.

Analysis of the Generated PS. As the last step, it is verified that the bad configurations are not reachable from the initial configurations in the generated PS. For that, *abstract regular model checking* can be used [4].

4 HADES Implementation

The HADES tool implements the above sketched approach and consists of several components depicted in Figure 2. HADES reads in an RTL description of the processor to be verified and converts it into its internal PSG representation. Currently, HADES supports the RTL format of CodAl which is an architectural description language for processor design [10]. For other RTL languages like VHDL and Verilog where architectural storages are not explicitly identified, a list of architectural storages with an explicit identification of the program counter must be provided.

[3] Three classes are distinguished—write instructions, read instructions, and other instructions.

Table 1: Experimental results.

Processor /		Simpl.	Data Flow	Consistency			Parameterized System				Total	Hazard
variant		Time [s]	Analysis [s]	Checking [s]			Generation and Verification [s]				Time [s]	Cases [#]
				rtl	smt	core	rtl	smt	armc	core		
TinyCPU	S	0.05	0.01	<0.01	0.25	0.49	0.01	0.38	5.44	6.71	13.34	5
	SA	0.06	0.02	<0.01	0.33	0.60	0.02	1.00	11.58	20.84	34.45	8
	B	0.05	0.01	<0.01	0.25	0.44	0.01	0.38	5.08	5.95	12.17	5
	BA	0.07	0.02	<0.01	0.33	0.63	0.03	1.03	11.02	18.28	31.41	8
	SF	0.06	0.02	<0.01	0.30	0.51	0.02	0.77	10.82	13.89	26.39	11
	SFA	0.07	0.02	<0.01	0.34	0.68	0.04	1.88	20.42	43.09	66.54	18
SPP8	S	0.27	0.04	0.01	0.43	0.85	0.05	2.02	20.81	36.24	60.72	27
	B	0.25	0.03	0.01	0.40	0.82	0.07	2.16	20.35	43.19	67.28	27
SPP16	S	0.27	0.05	0.01	0.44	0.90	0.04	1.90	19.99	36.33	59.93	27
	B	0.30	0.05	0.01	0.43	0.88	0.07	2.16	19.75	42.29	65.94	27
Codea2	SF	0.81	0.13	0.01	0.59	1.04	0.94	32.91	224.73	527.34	788.49	239
CompAcc	SFA	0.27	0.04	0.01	0.54	1.06	0.11	5.60	65.83	98.05	171.87	38
	BFA	0.28	0.05	0.01	0.55	1.04	0.28	7.74	66.03	158.56	234.94	53
DLX5	S	0.47	0.08	0.01	1.09	2.23	0.22	8.96	140.40	205.69	359.15	25
	SA	0.54	0.10	0.01	1.12	2.44	0.37	17.54	250.78	460.75	733.65	59
	B	0.62	0.12	0.01	1.07	2.40	0.33	9.47	138.55	316.08	468.65	25
	BA	0.65	0.12	0.01	1.15	2.69	0.48	19.28	247.98	745.16	1017.52	59

S Stalling Logic B Bypassing Logic F Flag Register(s) A Auto-increment Logic

The input PSG is normalized and simplified (conditional branching is replaced by multiplexors, value propagation is applied, redundant nodes and edges are removed, etc.). For that, the *RTL query engine* of HADES, which allows one to search for data-paths and substitute parts of the RTL design described by a PSG, is used. The engine uses a LISP-like syntax both for queries and their output, and it can handle basic RTL constructs like signals, registers, logic gates, as well as memory and its ports.

Subsequently, pipeline stages are identified by a simple data-flow analysis. Intuitively, the analysis propagates so far computed stages forward through the PSG, always taking the minimum of values incoming to a vertex and adding one whenever a storage other than a read port (which has a zero delay) is passed.

Next, instances of the consistency rules for the particular design are derived using RTL query engine. The rules are checked using an SMT solver for bit-vector logic. HADES is compatible with all SMT solvers accepting SMT2 formula description. In particular, for the below experiments, Z3 [17] was used.

Further, given a PSG with annotated stages, the HADES core repeatedly utilizes the RTL query engine (written in C++) and the SMT solver to extract potential hazard cases and to generate the appropriate PSs for them. The generated PSs are then checked using the abstract regular model checker of [3] (implemented in OCaml over the Timbuk tree automata library [11], however, tree automata degenerated to word automata are used only).

Note that the different hazard cases are are independent, and hence, in the future, the generation of the PSs and their verification can be run in parallel.

5 Experimental Evaluation

We have tested HADES on five processors: *TinyCPU* is a small 8-bit processor, mainly used for testing new verification methods. *SPP8* is an 8-bit ipcore with 3 pipeline stages, 16 general-purpose registers, and a RISC instruction set with 9 instructions. *SPP16* is a 16-bit variant of SPP8 with a more complex memory model. *Codea2* is a 16-bit processor for signal processing applications. 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. *CompAcc* is an 8-bit processor based on an accumulator architecture. Finally, *DLX5* is a 5-staged 32-bit processor able to execute a subset of the instruction set of the DLX architecture [19] (with no floating point support).

Compared with [6, 7, 8], we enriched the number of variants for the above introduced processors, which gave us 17 unique test cases in total. The variants of the particular processors differ in the following aspects: (i) the way how data hazards are avoided (pipeline stalling and clearing, data bypassing), (ii) the presence of flag / status registers, and (iii) utilization of so-called *auto-increment* (AI) logic. The AI logic is a feature allowing for an early incrementation[4] of the value of a register for memory addressing just before (pre-increment) or right after (post-increment) it is read. The AI feature usually brings a more efficient execution of sequences of instructions accessing the processor's memory (e.g., computation upon long arrays in cyber-security CPUs), but it also introduces potential WAW and WAR hazards that must be handled properly.

Besides the modifications in our test cases, we improved the HADES tool as well. This includes an addition of dynamic programming techniques (e.g., paths found in PSG are hashed and reused) and a faster pipe-based communication (instead of previously used file-based) between the HADES core and the RTL query engine.

We conducted a series of experiments on a PC with Intel Core i7-3770K @ 3.50GHz and 16 GB RAM with results shown in Table 1. The first columns give the verified processor, its variant, the time needed for the PSG simplification and its data flow analysis. The next columns give the duration of the consistency checking and the time spent by verification of the PSs that are created for each hazard case. The times are split to the times consumed by the different parts of the HADES architecture.

The following column gives the overall verification time, which remains in the order of minutes even for complex designs. Moreover, HADES also scales well with the growing size of the processor datapath as can be seen by comparing the times obtained for *SPP8* and *SPP16*. It should be noted that the amount of time consumed by the tool's core can be reduced by using a direct API of the SMT solver used instead of the current implementation that relies on exporting (potentially large) formulas in the `smt2` file format. (On the other hand, the current implementation does not depend on any particular SMT solver.) Finally, the last column represents the number of hazard cases that had to be generated and checked. This number differs from the one computed in [7, 8] due to HADES newly does not include hazard cases on the program counter among data hazards. These cases will be treated in separate control hazard detection phase, which is currently under implementation. Note that each hazard case represents a separate task so the part of generation and verification of PSs can be parallelized in the future.

During the experiments, we identified a flaw in a RAW hazard resolution when accessing the data memory in a development version of the *SPP8* processor.

6 Conclusions and Future Work

We have presented the main ideas, architecture, and evaluation of HADES—a tool for fully-automated discovery of data hazards in pipelined microprocessors. In the future, we plan to extend HADES with methods for verification of other processor features, such as control hazards. We also plan to parallize some parts of HADES and extend it with a compiler from VHDL and Verilog IP cores to the HADES input format.

Acknowledgement. The work was supported by the Czech Science Foundation project 14-11384S, the IT4IXS: IT4Innovations Excellence in Science project (LQ1602), and the internal BUT project FIT-S-14-2486.

[4] The incrementation typically takes place in an execution stage of the processor's pipeline.

References

[1] M. D. Aagaard (2003): *A Hazards-Based Correctness Statement for Pipelined Circuits*. In: *Proc. of CHARME'03*, *LNCS* 2860, Springer, pp. 66–80, doi:10.1007/978-3-540-39724-3_8.

[2] P. A. Abdulla, F. Haziza. & L. Holik (2013): *All for the Price of Few (Parameterized Verification through View Abstraction)*. In: *Proc. of VMCAI'13*, *LNCS* 7737, Springer, pp. 476–495, doi:10.1007/s10009-015-0406-x.

[3] A. Bouajjani, P. Habermehl, L. Holiḱ, T. Touili & T. Vojnar (2008): *Antichain-Based Universality and Inclusion Testing over Nondeterministic Finite Tree Automata*. In: *Proc. of CIAA'08*, LNCS 5148, Springer, doi:10.1007/978-3-540-70844-5_7.

[4] A. Bouajjani, P. Habermehl & T. Vojnar (2004): *Abstract Regular Model Checking*. In: *Proc. of CAV'04*, *LNCS* 3114, Springer, pp. 197–202, doi:10.1007/978-3-540-30579-8_19.

[5] J. R. Burch & D. L. Dill (1994): *Automatic Verification of Pipelined Microprocessor Control*. In: *Proc. of CAV'94*, *LNCS* 818, Springer, pp. 68–80, doi:10.1007/3-540-58179-0_44.

[6] L. Charvat, A. Smrcka & T. Vojnar (2012): *Automatic Formal Correspondence Checking of ISA and RTL Microprocessor Description*. In: *Proc. of MTV'12*, IEEE, pp. 6–12, doi:10.1109/mtv.2012.19.

[7] L. Charvat, A. Smrcka & T. Vojnar (2014): *Using Formal Verification of Parameterized Systems in RAW Hazard Analysis in Microprocessors*. In: *Proc. of MTV'14*, IEEE, pp. 83–89, doi:10.1109/mtv.2014.21.

[8] L. Charvát, A. Smrčka & T. Vojnar (2015): *Microprocessor Hazard Analysis via Formal Verification of Parameterized Systems*. In: *Proc. of EUROCAST'15*, *LNCS* 9520, Springer, pp. 605–614, doi:10.1007/978-3-319-27340-2_75.

[9] E. Clarke, M. Talupur & H. Veith (2006): *Environment abstraction for parameterized verification*. In: *Proc. of VMCAI'06*, *LNCS* 3855, Springer, pp. 126–141, doi:10.1007/11609773_9.

[10] Codasip Ltd. (2013): *CodAL Architecture Description Language*.

[11] T. Genet: *Timbuk: A Tree Automata Library*. http://www.irisa.fr/lande/genet/timbuk.

[12] K. Hao, S. Ray & F. Xie (2014): *Equivalence Checking for Function Pipelining in Behavioral Synthesis*. In: *Proc. of DATE'14*, IEEE, pp. 1–6, doi:10.7873/date.2014.163.

[13] R. B. Jones, C. H. Seger & D. L. Dill (1996): *Self-Consistency Checking*. In: *Proc. of FMCAD'96*, *LNCS* 1166, Springer, pp. 159–171, doi:10.1007/bfb0031806.

[14] A. Koelbl, R. Jacoby, H. Jain & C. Pixley (2009): *Solver Technology for System-level to RTL Equivalence Checking*. In: *Proc. of DATE'09*, IEEE, pp. 196–201, doi:10.1109/date.2009.5090657.

[15] U. Kuhne, S. Beyer, J. Bormann & J. Barstow (2010): *Automated Formal Verification of Processors Based on Architectural Models*. In: *Proc. of FMCAD'10*, IEEE, pp. 129–136.

[16] P. Mishra, H. Tomiyama, N. Dutt & A. Nicolau (2002): *Automatic Verification of In-Order Execution in Microprocessors with Fragmented Pipelines and Multicycle Functional Units*. In: *Proc. of DATE'02*, IEEE, pp. 36–43, doi:10.1109/date.2002.998247.

[17] L. De Moura & N. Bjorner (2008): *Z3: An Efficient SMT Solver*. In: *Proc. of TACAS'08*, *LNCS* 4963, Springer, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[18] K. S. Namjoshi (2007): *Symmetry and completeness in the analysis of parameterized systems*. In: *Proc. of VMCAI'07*, *LNCS* 4349, Springer, pp. 299–313, doi:10.1007/978-3-540-69738-1_22.

[19] D. A. Patterson & J. L. Hennessy (2012): *Computer Organization and Design: The Hardware / Software Interface*, fourth edition. Morgan Kaufmann, Boston, doi:10.1016/c2013-0-08305-3.

[20] M. N. Velev & P. Gao (2011): *Automatic Formal Verification of Multithreaded Pipelined Microprocessors*. In: *Proc. of ICCAD'11*, IEEE, pp. 679–686, doi:10.1109/iccad.2011.6105403.