Using Formal Verification of Parameterized Systems in RAW Hazard Analysis in Microprocessors

FIT BUT Technical Report Series

Lukas Charvat, Ales Smrcka, and Tomas Vojnar

Technical Report No. FIT-TR-2014-04
Faculty of Information Technology, Brno University of Technology

Last modified: October 22, 2014

Using Formal Verification of Parameterized Systems in RAW Hazard Analysis in Microprocessors

Lukas Charvat, Ales Smrcka, and Tomas Vojnar

Brno University of Technology, FIT, IT4Innovations Centre of Excellence
Božetěchova 2, 612 66, Brno, Czech Republic
{icharvat, smrcka, vojnar}@fit.vutbr.cz
http://www.fit.vutbr.cz

Abstract. Implementation of a pipeline-based execution of instructions in purpose-specific microprocessors is an error prone task, which implies a need of proper verification of the resulting design. Various techniques were proposed for this purpose, but they usually require a significant manual intervention of the developers. In this work, we propose a novel, highly automated approach for discovering RAW hazards in in-order pipelined instruction execution. Our approach combines static analysis of data paths to detect anomalies and possible hazards, followed by a transformation of detected problematic paths to a parameterised system (PS), and a subsequent formal verification to check the possibility of unhandled hazards using techniques for formal verification of PSs. We have implemented our approach and successfully applied it on multiple non-trivial microprocessors.

1 Introduction

For many current, highly demanding embedded applications, rather complex purpose-specific microprocessors are developed, including those with pipelined execution. Development of such processors requires the designer to reason about mutual interference of sequences of instructions in the pipeline where tricky errors can easily arise. Automated or semi-automated support of the development process, including suitable verification techniques, is hence highly needed. Various techniques have been proposed for this purpose, but they are still rather limited in terms of their generality, scalability, and/or degree of automation.

Our long-term goal is to develop a set of verification techniques with formal roots, each of them specialised in checking absence of a certain kind of errors in purpose-specific microprocessors. 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.

In our previous work [6], we proposed, with the above goal in mind, a fully automated approach for checking correctness of the implementation of individual instructions. In this paper, we aim at *read-after-write (RAW) hazards* in microprocessors with a single pipeline. 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.

Our approach for verifying that a single-pipeline microprocessor is free of RAW hazards starts by using a simple static analysis to find all data paths which could transfer data in a way causing an RAW hazard. Subsequently, we use SMT solving to check whether some of these paths could be enabled under some conditions. If so, we use parametric formal verification over a specially derived model of the data path to check whether there exists some sequence of instructions that could generate such conditions. If there is no such case, RAW hazards are handled properly by the processor.

Our approach concentrates mostly on the control parts of a design, for which current formal methods usually scale well, and minimizes the need of reasoning over data. We have implemented our approach in a prototype tool, and we present experimental evidence showing that our approach does indeed provide promising results in practice.

Plan of the Paper Section 2 presents an overview of the related work addressing validation of single-pipeline microprocessors. Section 3 defines the needed notions. Section 4 presents our verification approach, followed by experiments presented in Section 5. Finally, Section 6 concludes the paper.

2 Related Work

Showing absence of RAW hazards is a native part of checking conformance between an RTL design and a formally encoded ISA description. The perhaps most cited approach to such checking is the so-called flushing technique [5], which has been extended, e.g., in [13, 20, 11], to handle rather complicated designs with multi-cycle execution units, exceptions, and branch prediction. The main challenge of these works is to overcome the semantic gap between the different levels of a processor description. Dealing with this issue typically requires a significant user intervention consisting in providing various additional assertions about the design or in transforming it to a purpose-specific description language.

In [12], the so-called self-consistency check that compares possible executions of each instruction in two scenarios is introduced. The comparison is made wrt. a property given by the user, e.g., a property concerning RAW hazards which deals with (i) executions of an instruction enclosed by any (random) instructions within the pipeline and (ii) executions of the same instruction surrounded by `NOP` instructions only. If the self-consistency check succeeds, conformance of the RTL and ISA descriptions of a processor can be established by separately showing conformance of the RTL/ISA descriptions of each individual instruction. The main drawback of the approach is that it requires the enclosing instructions from the first run not to violate a so-called consistent state of the microprocessor, which has to be manually defined by the user.

In [1], a formal model based on a notion of stages, parcels (instructions), and hazards has been introduced. Once the user defines predicates needed for describing the pipeline, the design can be automatically formally proven correct under a correctness criterion given in the work. Another, a bit similar approach has been proposed in [14]. The approach introduces an abstract formal model whose components are to be linked by the user with the concrete cycle-accurate implementation through a number of mappings. Afterwards, IPC [18] is used to check several properties implying correctness of

the pipeline behaviour. Again, both of the above methods require a significant manual user intervention.

Compared with the above approaches, we do not aim at full conformance checking between RTL and ISA implementations. Instead, we address one specific property—namely, absence of problems caused by RAW hazards. On the other hand, our approach is almost fully automated—the only step required from the user is to identify the architectural registers.

3 Definitions

A *processor structure graph* (PSG) is a tuple $G = (V, E, s, t)$. V is a finite set of vertices that is the union $V = V_s \cup V_f$ of a set V_s of *storages* and a set V_f of *Boolean circuits*, $V_s \cap V_f = \emptyset$. V_s includes five types of storages, meaning that

$$V_s = V_a \cup V_p \cup V_\mu \cup V_{rp} \cup V_{wp}$$

where V_a, V_p, and V_μ are sets of *architectural*, *pipeline*, and *micro-architectural registers*, respectively, while V_{rp} and V_{wp} denote sets of *read* and *write ports of memories* (with all the sets being pairwise disjoint). We assume that architectural registers and memory ports are in the set of *programmer visible storages* $V_{pv} = V_a \cup V_{rp} \cup V_{wp}$. We also expect micro-architectural registers to only hold the state of processor's control logic and to be not used to carry data interchanged between programmer visible registers. Moreover, we expect all storages to have a unit write and zero read delay. Longer access times (e.g., for memory ports) can be modelled by introducing sequentially connected registers emulating the required delay. The set of Boolean circuits is the union of two types of circuits $V_f = V_{mx} \cup V_g$ where V_{mx} is a set of circuits implementing *multiplexers* and V_g is a set of the remaining (generic) circuits, $V_{mx} \cap V_g = \emptyset$.

Next, E denotes a finite set of *transfer edges*. Let $\mathbb{T} = \{\mathtt{d}, \mathtt{q}, \mathtt{cl}, \mathtt{st}, \mathtt{addr}, \mathtt{en}, \mathtt{sel}, \mathtt{m}\} \cup \{\mathtt{a}_i, \mathtt{case}_i \mid i \in \mathbb{N}\}$ be the set of *connection types* whose meaning will become clear below. Finally, $s : E \to V \times \mathbb{T}$ assigns to each edge its source vertex and its connection type, and $t : E \to V \times \mathbb{T}$ assigns to each edge its target vertex and its type of connection. The sets V and E and the functions s and t must fulfil the following criteria:

- Each architectural, micro-architectural, or pipeline register $v_r \in V_a \cup V_\mu \cup V_p$ has exactly one inbound data-in edge $e_d \in E$ s.t. $t(e_d) = (v_r, \mathtt{d})$ and arbitrary many outbound data-out edges $e_q^i \in E$ s.t. $s(e_q^i) = (v_r, \mathtt{q})$ for $i \in \mathbb{N}$.
- In addition, each pipeline storage $v_p \in V_p$ has exactly one inbound stall $e_{st} \in E$ and clear edge $e_{cl} \in E$ s.t. $t(e_{st}) = (v_p, \mathtt{st})$ and $t(e_{cl}) = (v_p, \mathtt{cl})$.
- Each memory port $v_{port} \in V_{rp} \cup V_{wp}$ has exactly one inbound addressing edge $e_{addr} \in E$ s.t. $t(e_{addr}) = (v_{port}, \mathtt{addr})$.
- Each memory write port $v_{wp} \in V_{wp}$ has exactly one inbound data-in edge $e_d \in E$ s.t. $t(e_d) = (v_r, \mathtt{d})$.
- Each memory read port has arbitrary many outbound data-out edges $e_q^i \in E$ s.t. $s(e_q^i) = (v_r, \mathtt{q})$ for $i \in \mathbb{N}$.

- Each write and read port pair $(v_{wp}, v_{rp}) \in V_{wp} \times V_{rp}$ of the same memory is connected with edge e_m s.t. $s(e_m) = (v_{wp}, \mathtt{m})$ and $t(e_m) = (v_{rp}, \mathtt{m})$.
- Each storage $v_s \in V_s$ has exactly one inbound enable edge $e_{en} \in E$ s.t. $t(e_{en}) = (v_s, \mathtt{en})$.
- Each circuit $v_g \in V_g$ implementing a Boolean function $g(a_1, ..., a_n)$ has exactly one inbound edge for each argument of g s.t. $t(e_{a_i}) = (v_g, \mathtt{a}_i)$ for all $1 \leq i \leq n$.
- Each multiplexer $v_{mx} \in V_{mx}$ implementing a case selection function $switch(sel, case_1, ..., case_n)$ has exactly one inbound edge for each argument of $switch$ s.t. $t(e_{sel}) = (v_{mx}, \mathtt{sel})$ and $t(e_{case_i}) = (v_{mx}, \mathtt{case}_i)$ for all $1 \leq i \leq n$.
- Each circuit $v_f \in V_f$ has arbitrary many outbound result edges $e_q^i \in E$ s.t. $s(e_q^i) = (v_f, \mathtt{q})$ for $i \in \mathbb{N}$.
- There are no other types of edges other than the ones described above.
- There is no cycle in the graph consisting from vertices representing Boolean circuits only.

Due to the above restriction to at most one inbound edge for a single connection type, one can use a simpler notation to uniquely describe the edges. In particular, an edge $e \in E$ that satisfies $t(e) = (v, \mathtt{c})$, $v \in V$, can be encoded using the expression $v.\mathtt{c}$.

A PSG $G = (V, E, s, t)$ induces a transition system $(C, \rightarrow)$ where $C = \mathbb{B}^n$ is the set of its configurations where n is a sum of bit-width of all storages and $\rightarrow \subseteq C^2$ is the transition relation. We use $c[v_r]$, resp. $c[v_m, i]$, to denote the bit-vector value of the register $v_r \in V_a \cup V_p \cup V_\mu$, resp. the ith cell of the memory with port $v_m \in V_{rp} \cup V_{wp}$, in the configuration $c \in C$. We abuse the above notation $c[e]$ to express the value transferred over edge $e \in E$ in configuration c as well. Now, assume standard register next state function f_{v_r} for each register v_r of the PSG where the register v_r is written a value transferred through $v_r.\mathtt{d}$ edge iff $v_r.\mathtt{cl}$ and $v_r.\mathtt{st}$ edges both transfer "0" and $v_r.\mathtt{en}$ transfers "1" in configuration c. We assume priorities $cl > st > en$ of control signals in the case when more than is activated. Similar next state function $f_{v_m,i}$ exists for each memory cell i which is written a value transferred through its writing port v_{wp} and its edge $v_{wp}.\mathtt{d}$ iff $v_{wp}.\mathtt{en}$ edge transfers "1" and $v_{wp}.\mathtt{addr}$ edge transfers i in configuration c. Then, the relation $\rightarrow$ contains transition $c \rightarrow c'$ iff (i) $c'[v_r] = f_{v_r}(c[v_r.\mathtt{d}], c[v_r.\mathtt{en}], c[v_r.\mathtt{cl}], c[v_r.\mathtt{st}])$ for all $v_r \in V_a \cup V_p \cup V_\mu$, and (ii) $c'[v_m, i] = f_{v_m,i}(c[v_r.\mathtt{d}], c[v_r.\mathtt{en}], c[v_r.\mathtt{addr}])$ for all memories with port $v_m \in V_{wp}$ and their cells i.

Given $k > 1$ and vertices $v_1, v_k \in V$ of a PSG, a walk from v_1 to v_k is an alternating sequence of vertices and edges $\langle v_1, e_1, v_2, ..., v_k \rangle$ where $v_2, ..., v_{k-1} \in V$, $e_1, ..., e_{k-1} \in E$, and every two subsequent vertices are incident with an edge present between them, i.e., $s(e_i) = (v_i, \mathtt{c_i})$, $t(e_i) = (v_{i+1}, \mathtt{c_{i+1}})$ for each $1 \leq i < k$ and $\mathtt{c_1}, ..., \mathtt{c_k} \in \mathbb{T}$. A path from v_1 to v_k is a walk where no vertex appears twice, i.e., $i \neq j \Rightarrow v_i \neq v_j$ for all $1 \leq i, j \leq k$.

Since our approach builds on analysing conditions that hold in certain stages of execution of a given instruction, we now introduce a notion of edge and path conditions. An *edge condition* is a pair (e, b), denoted $e \rightsquigarrow b$, meaning that the value transferred over the edge $e \in E$ is equal to some constant or variable b of bit-vector logic. By $\mathbb{E}$, we denote the set of all such edge conditions. A *path condition* $c^{\pi}_{v_{mx}}$ is a special type of edge condition defined for a multiplexer $v_{mx} \in V_{mx}$ that is part of a path

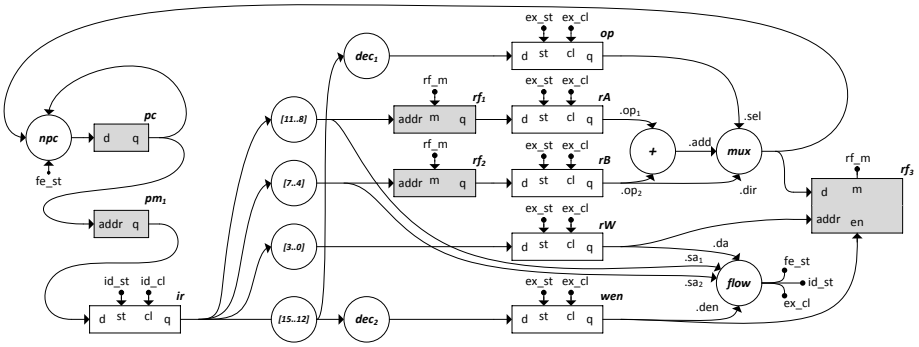

Fig. 1: A processor structure graph of a part of a RISC CPU.

$\pi = \langle ..., e_{in}, v_{mx}, e_{out}, ... \rangle$, $e_{in}, e_{out} \in E$. The path condition $c^{\pi}_{v_{mx}} = v_{mx}.\texttt{sel} \rightsquigarrow b$ captures the value b of the selector edge $v_{mx}.\texttt{sel}$ of the multiplexer v_{mx} required for the data on the inbound edge e_{in} to be propagated to the outbound edge e_{out}. The edge condition may be used to represent constraint over the set of configurations C of the transition system induced by the PSG. Thus, we also define a mapping $\alpha \colon \mathbb{E} \to 2^C$ s.t. $\alpha(e \rightsquigarrow b) := \{c \in C \mid c[e] = b\}$ and its pointwise extension $\alpha(K) := \bigcap_{k \in K} \alpha(k)$.

Our approach further uses the common notion of a parameterised system (PS) operating on a linear topology where processes (i.e., executed instructions) may perform local transitions or universally/existentially guarded global transitions [7, 17, 2]. For our purposes, it is enough to consider global transitions only. A PS is a pair $P = (Q, \Delta)$ where Q is a finite set of states of a process and Δ is a set of transition rules over Q. A transition rule is of the form $\mathbb{Q}j < i \colon G \models q \to q'$ where $\mathbb{Q} \in \{\forall, \exists\}$, $G \subseteq Q$ and $q, q' \in Q$. A PS induces a transition system whose configurations are finite words over Q. A configuration $q_1 ... q_i ... q_n$, $1 \leq i \leq n$, changes to $q_1 ... q'_i ... q_n$ when the ith process goes from its state q_i to q'_i using some of the transition rules. The rule can be applied only if its guard is satisfied. For example, the meaning of the guard $\forall j < i \colon G$ is "for every process j to the left from i (in the linear topology), the jth process should be in a state that belongs to the set G". Wihout loss of generality, during a verification, we assume a PS with a fairness assumption that during transition of between two configurations, every process must yield a step.

We will work with the reachability problem given by a PS P, a regular set $I \subseteq Q^+$ of initial configurations, and a regular set $Bad \subseteq Q^+$ of bad configurations. In particular, we assume Bad to be given as the upward closure of a finite set $B \subseteq Q^+$ of minimal bad configurations, this is, $Bad = \{c \in Q^+ \mid \exists b \in B \colon b \sqsubseteq c\}$ where $\sqsubseteq$ is the usual sub-word relation (i.e., $u \sqsubseteq s_1 ... s_n \Leftrightarrow u = s_{i_1} ... s_{i_k}$ for some $1 \leq i_1 \leq ... \leq i_k \leq n$, $0 \leq k \leq n$). Now, let $R \subseteq Q^*$ denote the set of all reachable configurations. We say that the system P is safe wrt. I and Bad iff no bad configuration is reachable, i.e., $R \cap Bad = \emptyset$.

4 The Proposed Verification Approach

This section describes our approach for verifying that the logic of data-flow control prevents RAW hazards, making it impossible for a later instruction to read incorrect (not yet updated) data, use them for a computation, and write them into some programmer visible storage. We expect the processor under verification to be described by a PSG, which can be easily obtained from a description of the processor on the register transfer level (RTL) written in common languages, such as VHDL or Verilog. For our experiments, we have, in particular, implemented a translation from RTL generated from the CodAL hardware description language [8] to PSGs.

Our approach consists of the following steps: (i) a data-flow analysis intended to distinguish particular stages of the pipeline, (ii) a consistency check of a correct implementation of the particular pipeline stages, (iii) a static analysis identifying constraints over data-paths of instructions that can potentially cause an RAW hazard, (iv) generation of a PS modelling mutual interaction between potentially conflicting instructions, and (v) an analysis of the constructed PS.

Example 1. Throughout the section, we will be illustrating the different steps of our approach on a running example depicted in Fig. 1. The figure shows a PSG describing a part of a simple RISC CPU. The depicted part of the CPU is used during execution of arithmetic and load/store instructions. To remain lucid, the PSG does not include any logic responsible for execution of branch instructions, and it shows only selected parts of an instruction decoder (circuits dec_1 and dec_2). Also, the write enable signal for the register pc as well as the read enable signals for the ports rf_1, rf_2 (read ports of register file rf), and pm_1 (which is a port to the program memory) are omitted because of their constant setting to "1".

In the CPU, the computation starts by using the content of the program counter pc to address the read port pm_1 of the program memory. An instruction fetched from the memory is stored into the storage ir representing the fetch register. The fetched instruction word is split by the circuits denoted as $[x..y]$ to its operand and opcode parts. Values of the operands are used to address the read ports rf_1, rf_2, and the write port rf_3 of the register file. The opcode part is sent to the decoder to determine the type of the ALU operation to be performed and to select its destination by setting its enable connection (in our case, the write port rf_3 of the register file) to "1".

The node $flow$ represents the flow logic of the controller which is responsible for dealing with RAW hazards on the register file. The flow logic implements the function $flow(sa_1, sa_2, den, da) = den \wedge (sa_1 = da \vee sa_2 = da)$[1] which checks the value of the enable den and the address da also connected to the write port rf_3 of an earlier instruction with the addresses sa_1, sa_2 of the read ports rf_1, rf_2 of a later instruction. In case the later instruction wants to read from the same address as the writing one, the flow logic uses stall and clear signals, transferred by edges such as $ir.\texttt{st}$ or $wen.\texttt{cl}$, to insert a pipeline bubble between the instructions. ◁

[1] For simplicity, we do not strive for an optimal hazard solution.

4.1 Data-Flow Analysis

The input of our approach (apart from the PSG) is the identification of architectural registers including the program counter. We use a simple data flow analysis to get the number of pipeline stages and the mapping of storages and logic functions into the pipeline stages. We define a *pipeline stage* as a sub-graph of a PSG responsible for executing a single-cycle step of an instruction. The pipeline stage that a vertex (representing some storage or function) of a PSG belongs to is given by the minimum number of cycles needed to propagate data from the input of the program counter (assumed to be read from a fictive stage 0) to the output of the given vertex. Hence, as a particular case, the program counter itself belongs to stage 1.

The simple data-flow analysis that we use starts from the program counter and its stage 1 and propagates the 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. This analysis gives us the mapping $\varphi\colon V_s \to \mathbb{S}, \mathbb{S} = \{0, ..., n\}$, $n \in \mathbb{N}$, of storages to pipeline stages.

Table 1: Pipeline stages and potential hazards.

Storage	Type	Stage	Write stages	Read stages	Potential
		φ	φ_{wr}	φ_{rd}	hazard
pc	arch	1	$\{0,1,2,3\}$	$\{0,1\}$	✓
pm_1	port	–	$\emptyset$	$\{1\}$	×
ir	pipe	2	$\{1,2,3\}$	$\{0,1,2\}$	–
rf_1	port	–	$\emptyset$	$\{2\}$	✓
rf_2	port	–	$\emptyset$	$\{2\}$	✓
op	pipe	3	$\{2,3\}$	$\{0,3\}$	–
rA	pipe	3	$\{2,3\}$	$\{0,3\}$	–
rB	pipe	3	$\{2,3\}$	$\{0,3\}$	–
rW	pipe	3	$\{2,3\}$	$\{0,1,2,3\}$	–
wen	pipe	3	$\{2,3\}$	$\{0,1,2,3\}$	–
rf_3	port	4	$\{3\}$	$\emptyset$	✓

Subsequently, by looking at all the storages from which there is a path to a given storage not passing through any further storage, we can easily get the *write stage* mapping $\varphi_{wr}\colon V_s \to 2^{\mathbb{S}}$ describing which stages directly influence the value of the given storage. In case of the program counter, we always add the fictive stage 0. Likewise, by looking at all target storages that can be reached from a given storage by a path not passing through any further storage, one can derive the *read stage* mapping $\varphi_{rd}\colon V_s \to 2^{\mathbb{S}}$ describing which stages outputs (directly connected to targeted storages) are influenced by the given storage. Pipeline stages of the storages from the PSG of Fig. 1 and the corresponding read and write stages, computed as described above, are shown in Table 1.

4.2 Consistency Checking

The second step of our method is consistency checking which checks whether the flow logic assures a correct in-order execution of all instructions through all the identified pipeline stages. This means that all instructions which are fetched from the program memory should flow from the first stage to the last stage while maintaining their execution order with no loss or duplication of an instruction. We check whether the flow logic obeys a set of rules which express how the control connections ($\texttt{en}, \texttt{st}, \texttt{cl}$) of storages in adjacent pipeline stages should be set. In particular, we use a strengthened variant of the rules proposed in [15].

Let us have a transition system $(C, \rightarrow)$ induced by the verified PSG, a set $V_t = V_a \cup V_p \cup V_{wp}$ and mappings $\mathit{stalled}$, $\mathit{cleared}\colon V_t \rightarrow 2^C$ defined as:

$$\mathit{stalled}(v_t) = \{c \in C \mid (v_t \in V_p \wedge c[v_t.\texttt{st}] = 1) \vee c[v_t.\texttt{en}] = 0\}$$

$$\mathit{cleared}(v_t) = \{c \in C \mid (v_t \in V_p \wedge c[v_t.\texttt{cl}] = 1) \vee (v_t \notin V_p \wedge c[v_t.\texttt{en}] = 0)\}$$

assigning to each storage $v_t \in V_t$ a set of configurations s.t. the storage v_t is not written, resp. cleared. Then, the following rules are checked:

- If some pipeline register of a stage s is stalled or not written, then all pipeline and architectural registers and write ports of the stage s have to be stalled or not written, i.e., for all $v_p \in V_p$, $v_t \in V_t$:

  $$(\varphi(v_p) = \varphi(v_t)) \Rightarrow (\mathit{stalled}(v_p) \subseteq \mathit{stalled}(v_t)) \tag{1}$$

  The rule follows the idea that an instruction carried by a pipeline stage cannot be fragmented.
- If some pipeline register in stage s is stalled, then all pipeline and architectural registers and write ports of stage $s + 1$ have to be stalled, not written, or cleared, i.e., for all $v_p \in V_p$, $v_t \in V_t$:

  $$(\varphi(v_p) = \varphi(v_t) - 1) \Rightarrow (\mathit{stalled}(v_p) \subseteq \mathit{stalled}(v_t) \cup \mathit{cleared}(v_t)) \tag{2}$$

  This rule is transcription of the Equation (15) from [15] and prevents duplication of an instruction.
- If some pipeline register in stage s is stalled, then all pipeline and architectural registers and write ports of stage $s - 1$ have to be stalled or not written, i.e., for all $v_p \in V_p$, $v_t \in V_t$:

  $$(\varphi(v_p) = \varphi(v_t) + 1) \Rightarrow (\mathit{stalled}(v_p) \subseteq \mathit{stalled}(v_t)) \tag{3}$$

  This rule is transcription of the Equation (16) from [15] and prevents an instruction to be lost.
- If some pipeline register in stage s is cleared, then all pipeline registers of stage s have to be cleared and all architectural registers and write ports must not be written, i.e., for all $v_p \in V_p$, $v_t \in V_t$:

  $$(\varphi(v_p) = \varphi(v_t)) \Rightarrow (\mathit{cleared}(v_p) \subseteq \mathit{cleared}(v_t)) \tag{4}$$

  Similarly to the rule (1), this rule prevents a fragmentation of an instruction.

We check the above rules using SMT solver [4, 16] for bit-vector logic. Let $\mathcal{F}$ denote all formulae of such a logic. Further, assume an existence of operator $^\star : E \to \mathcal{F}$ that maps edges of PSG to variable references of bit-vector logic s.t. $e_1^\star = e_2^\star \Leftrightarrow s(e_1) = s(e_2)$ for each $e_1, e_2 \in E$. We can create formula $\psi(e) \in \mathcal{F}$ that expresses computation of the value for any $e \in E$. The $\psi(e)$ is recursivelly defined as

$$\psi(e) := \begin{cases} e^\star = f(e_1^\star, ..., e_m^\star) \wedge \psi(e_1) \wedge ... \wedge \psi(e_m) & s(e) = (v_f, \mathtt{q}) \in V_f \times \{\mathtt{q}\} \\ \mathit{True} & \text{otherwise} \end{cases}$$

Then, e.g., the inclusion, used by rule (4), $\mathit{cleared}(v_p) \subseteq \mathit{cleared}(v_t)$ can be checked by evaluating

$$\neg sat((\psi(v_p.\mathtt{cl}) \wedge v_p.\mathtt{cl}^\star = 1) \wedge (\psi(v_t.\mathtt{cl}) \wedge v_t.\mathtt{cl}^\star = 0))$$

for $v_t \in V_p$, and

$$\neg sat((\psi(v_p.\mathtt{cl}) \wedge v_p.\mathtt{cl}^\star = 1) \wedge (\psi(v_t.\mathtt{en}) \wedge v_t.\mathtt{en}^\star = 1))$$

for $v_t \in V_t \setminus V_p$, i.e. $v_t \in V_a \cup V_{wp}$.

4.3 Static Detection of Potential RAW Hazards

In the next step, a static hazard analysis examines the PSG and the pipeline stage mappings φ, φ_{wr}, φ_{rd} determined by the data-flow analysis and identifies a finite set of so-called *hazard cases*. Each hazard case describes one possible source of an RAW hazard. In the construction of hazard cases, the most interesting step is the derivation of the so called *minimal influence path*, the other elements of hazard tuples are obtained by a simple examination of the mappings φ, φ_{wr}, φ_{rd}. We define an *influence path* as a path $\langle v_1, e_1, ..., v_k \rangle$ where the value read from a programmer visible storage $v_1 \in V_{pv}$ can influence a value stored to an programmer visible storage $v_{pv} \in V_a \cup V_{wp}$ by writing to a *target* storage $v_k \in V_s$. Each influence path must fulfill the following set of properties: (i) The target storage v_k must be either (a) an architectural register or a write port, i.e., the case when $v_k = v_{pv}$, or (b) a pipeline register s.t. $t(e_{k-1}) = (v_k, \mathtt{cl})$. Indeed, clearing of the pipeline register v_k will surely influence all programmer visible storages that belong to stages $s \geq \varphi(v_k)$. Next, (ii) the influence path must not traverse through stall connections of pipeline registers. Such paths cannot influence the value of any programmer visible register. Their only impact can be that they stall a stage but in such a case the previously established satisfaction of the consistency rules assures the correct conservation of all the partially executed instructions. Finally, (iii) access stages of elements along the path must conform to those obtained during the data-flow analysis and these stages have to form an increasing sequence. Otherwise, there could not be any instruction capable of a data transfer along the influence path.

An error in the RAW hazard prevention logic is manifested upon the first write of incorrect data into some programmer visible storage of the design. Therefore, it is sufficient to further work only with the minimal influence path which is an influence path where $v_i \notin V_a \cup V_{wp}$ and $t(e_{i-1}) \notin V_p \times \{\mathtt{cl}\}$ for all $1 < i < k$. To discover the

minimal influence paths in the given PSG, one can think of using a standard breadth-first search with the rules (i-iii) and the minimality checked on-the-fly.

A hazard case $(v_w, s_w, v_r, s_r, v_t, s_t, \pi)$ consists of (i) a programmer visible storage v_w (i.e., a register or a writing port of the memory), (ii) its write stage $s_w \in \varphi_{wr}(v_w)$, (iii) a reading storage v_r such that $v_r = v_w$ if v_r is a register or such that v_r and v_w are ports of the same memory, (iv) the read stage $s_r \in \varphi_{rd}(v_r)$ such that $s_r < s_w$ in order that the storage is read before it is written to evoke the RAW hazard, (iv) a target storage v_t where the potentially incorrect value read from v_r is stored, (v) a stage $s_t \in \varphi_{wr}(v_t)$, $s_r \leq s_t$, in which the incorrect value is stored, and (vi) a minimal influence path π describing how data are propagated from v_r to v_t between the stages s_r and s_t. Note that since the definition of a hazard case speaks about 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. Further, note that the case when $s_r = s_w$ is not included since the consistency checking guarantees an isolated execution of the instructions.

Example 2. Consider results of data-flow analysis computed for the PSG from Fig. 1 shown in Table 1. In the table, one can see that there is a potential RAW hazard on storage rf because, for example, its read port rf_2 can be read in stage 2 ($\varphi_{rd}(rf_2) = \{2\}$), and its write port rf_3 can be written in stage 3 ($\varphi_{wr}(rf_3) = \{3\}$). Therefore, the subsequent verification will include a check whether a RAW hazard is indeed possible between two classes of instructions. The first one consists of instructions writing to rf_3 at address a in stage 3. The other includes instructions reading from rf_2 at the same address a in stage 2. In the PSG, there are several target storages, such as rf_3 and pc, where the reading instruction can store potentially incorrectly fetched data. Thus, multiple hazard cases need to be considered. For example, assume the rf_3 as a target. There are multiple minimal influence paths leading to rf_3, e.g., $\pi_1 = \langle rf_2, rB.\texttt{d}, rB, +.\texttt{op}_2, +, mux.\texttt{add}, mux, rf_3.\texttt{d}, rf_3\rangle$ or $\pi_2 = \langle rf_2, rB.\texttt{d}, rB, mux.\texttt{dir}, mux, rf_3.\texttt{d}, rf_3\rangle$. This means that the following two hazard cases need to be investigated: $(rf_3, 3, rf_2, 2, rf_3, 3, \pi_1)$ and $(rf_3, 3, rf_2, 2, rf_3, 3, \pi_2)$. An analogical process is then applied for pc as the target. ◁

4.4 Generation of a PS Modelling the Possible Hazards

We will now describe how the behaviour of the instructions given by constraints of a hazard case can be modelled and verified for correctness using a PS $P = (Q, \Delta)$. In the system P, we map n instructions in the pipeline to n processes in a linear array (with the earliest on the left). Initially, they are in a state saying that their execution has not started. Then, they proceed through individual stages of the pipeline during which they may interact with each other by the means of pipeline flow logic, e.g., an earlier instruction may force a later instruction to be stalled, or cleared. Finally, the instructions end up in a state denoting that they left the pipeline.

Let us have a hazard case $(v_w, s_w, v_r, s_r, v_t, s_t, \pi)$. In the system P, we model interactions among three classes of processes (and hence 3 types of instructions) $\mathbb{K} = \{w, rw, any\}$. The w-class includes every instruction that writes to storage v_w in stage s_w. The rw-class includes instructions that read from storage v_r in stage s_r, perform

a data computation that involves the data path π, and write to storage v_t in stage s_t. The any-class instructions are used as pipeline filler representing ANY other instruction. The set of states of a parameterized system P is then given by pairs $(k, s) \in \mathbb{K} \times \mathbb{S}$ where k given a type of an instruction and s gives the stage in which the instruction is currently executing. Hence, $Q \subseteq \mathbb{K} \times \mathbb{S}$, and we will use the notation q_s^k to denote a stage $(k, s) \in Q$. For a pipeline of length m, the sequence $q_0^k, ..., q_m^k$ records each step of a k-class instruction in the pipeline.

In order to define transition relation Δ, we expect existence of a mapping ξ and predicates R^{st}, R^{cl}, R^{cf}. Each of these elements is described in more detail in the following subsections.

4.4.1 Determination of Edge Conditions for States of the Parameterized System

The mapping $\xi\colon Q \to 2^{\mathbb{E}}$ describes behaviour of an instruction using a set of edge conditions that hold in the given state $q \in Q$ and it is constructed by examination of the currently processed hazard case. For a w-class instruction, we need to express enabling of a write for storage v_w at stage s_w. Therefore, we put $\xi(q_{s_w}^w) = \{v_w.\texttt{en} \rightsquigarrow 1\}$. In a case that storage v_w is a memory port, the set of edge conditions $\xi(q_{s_w}^w)$ additionally includes a condition $v_w.\texttt{addr} \rightsquigarrow a$ where a is a symbolic address, i.e. a name of bit-vector variable denoting an arbitrary address. We set the value of $\xi(q_s^w)$ to $\emptyset$ for all $s \in \mathbb{S} \setminus \{s_w\}$. The mapping ξ for states of rw-class instruction is sligthly complicated. We introduce $C_\pi \subseteq \mathbb{E} \times \mathbb{S}$ as stage-marked conditions for a given influence path π assigning an edge condition with a stage for which the condition must be valid such that the whole path π propagates its data. For states q_i^{rw}, $i \in \mathbb{S}$ of rw-class instruction, the mapping ξ is defined as $\xi(q_i^{rw}) := \{c \in \mathbb{E} \mid (c, i) \in C_\pi\}$.

Now, we will show how the set C_π can be computed using Alg. 1. The algorithm works with a stage counter n. An actual stage represented by the value of the counter n is used to mark edge conditions that are newly added to the set C_π. The counter n is initially set so its value corresponds to the stage s_r, i.e., the stage where storage v_1 is read. At first (line 2), based on the type of storage v_1, the algorithm determines edge conditions that must hold at stage s_r to enable reading from the storage v_1. Note that if v_1 is a read port, we use the same address a as in the previous case for writing by a w-class instruction. On every visit of a vertex representing pipeline register, the stage counter n is incremented. On lines 3–6, the algorithm iterates over vertices of the path π. Further, the set C_π is constructed during this iteration. A visit of a multiplexer or read port vertex means that C_π is extended with a new edge condition (line 5). In the case of a multiplexer, the newly added condition is a path condition $c_{v_i}^\pi$ describing a necessary setting of a selector edge $v_i.\texttt{sel}$ required for a propagation data over the path π through multiplexer v_i. In the case of a read port, an edge condition to enable read from the port is necessary if the path π traverses through its address connection. This is necessary because bare setting of address without port enabling does not propagate any information. Similarly, to retain the data influence, an edge condition for enabling of a write to v_k is required in the case when path π ends with an address or a data edge (lines 7–9 of the algorithm).

Algorithm 1 Extraction of path edge conditions

Require: A sequence $\langle v_1, e_1, ..., v_k \rangle$ to be an influence path in PSG $G = (V, E, s, t)$, reading stage $s_r \in \varphi_{rd}(v_1)$, writing stage $s_w \in \varphi_{wr}(v_k)$, and address a (in case of $v_1 \in V_{rp}$).

Ensure: A set C_π containing stage-marked edge conditions.

1: $n := s_r$

2: $C_\pi := (v_1 \in V_{rp})$ **?** $\{(v_1.\texttt{en} \rightsquigarrow 1, s_r), (v_1.\texttt{addr} \rightsquigarrow a, s_r)\}$ **:** $\emptyset$

3: **for** $i := 2$ **to** $k - 1$ **do**

4: $\quad n := (v_i \in V_p)$ **?** $(n + 1)$ **:** n

5: $\quad C_\pi := C_\pi \cup \begin{cases} \{(c^\pi_{v_i}, n)\} & v_i \in V_{mx} \\ \{(v_i.\texttt{en} \rightsquigarrow 1, n)\} & v_i \in V_{rp} \wedge e_{i-1} = v_i.\texttt{addr} \\ \emptyset & \text{otherwise} \end{cases}$

6: **end for**

7: **if** $e_{k-1} = v_k.\texttt{addr} \vee e_{k-1} = v_k.\texttt{d}$ **then**

8: $\quad C_\pi := C_\pi \cup \{(v_k.\texttt{en} \rightsquigarrow 1, s_w)\}$

9: **end if**

4.4.2 Computation of an Influence of the Flow Logic

The predicates $R^{st}, R^{cl} \subseteq \mathbb{S} \times Q^2$, $R^{cf} \subseteq Q^2$ reflect the pipeline's control logic which disallows some instruction interleavings. The predicate $R^{st}(s, q^{k_1}_{s_1}, q^{k_2}_{s_2})$, resp. $R^{cl}(s+1, q^{k_1}_{s_1}, q^{k_2}_{s_2})$, evaluates to true for $s, s_1, s_2 \in \mathbb{S}$, $k_1, k_2 \in \mathbb{K}$ iff edge conditions $\xi(q^{k_1}_{s_1}) \cup \xi(q^{k_2}_{s_2})$ that hold in states $q^{k_1}_{s_1}$, $q^{k_2}_{s_2}$ lead to stalling, resp. clearing, of the stage s, resp. $s + 1$. The predicate R^{cf} holds iff conditions $\xi(q^{k_1}_{s_1}) \cup \xi(q^{k_2}_{s_2})$ are in contradiction meaning that concurrent presence of instructions in states $q^{k_1}_{s_1}$, $q^{k_2}_{s_2}$ is impossible within the verified design. In order to express values of these predicates, we use a mapping $unroll \colon Q \to 2^C$ where C is a set of configurations of the TS $T = (C, \to)$ induced by the PSG, and a predicate $csat \subseteq 2^{\mathbb{E}} \times 2^Q$.

The idea of the $unroll$ mapping is to provide, for some state q^k_s, all the configurations of the TS T for which it is possible to get the microprocessor to the given state. To compute the configurations, we need to take into account the given state and its previous and next states, i.e. the configurations of q^k_s and configurations of the predecessors of the state (q^k_i, $i < s$) and successors of the state (q^k_i, $i > s$). To illustrate this, suppose that $v_1.\texttt{q} \rightsquigarrow 7, v_2.\texttt{en} \rightsquigarrow 1 \in \xi(q^k_{s-1})$ holds in state q^k_{s-1} and the PSG is such that $s(v_2.\texttt{d}) = (v_1, \texttt{q})$, $v_1, v_2 \in V_\mu$. Since it takes a single cycle to propagate input to output in a micro-architectural register, we can easily derive that the value of the edge $v_2.\texttt{q}$ is 7 for state q^k_s. Now, assume that there is variable identified by ${}^k_{s-1}v_1.\texttt{q}^\star$ resp. ${}^k_s v_2.\texttt{q}^\star$ of bit-vector logic (BVL) for SMT solver representing the value which holds the edge $v_1.\texttt{q}$ in state q^k_{s-1} resp. the edge $v_2.\texttt{q}$ in state q^k_s. The BVL formulas over these variables define the configurations of the TS.

To describe the construction of the mapping $unroll$, we use a sequence of configurations ${}^k_s F = \langle {}^k_s K^{(-s)}, \ldots, {}^k_s K^{(0)}, \ldots, {}^k_s K^{(m-s)} \rangle$, $m = \max(\mathbb{S})$ with the idea that an element of the sequence ${}^k_s K^{(i)}$ represents possible configurations of TS T for a k-class instruction in stage i computed and denoted relatively to the stage s. The following rules over elements of the sequence ${}^k_s F$ have to be satisfied: (i) Configurations represented by the ${}^k_s F$'s elements must conform to edge conditions previously determined via the ξ mapping, i.e., ${}^k_s K^{(i)} \subseteq \alpha(\xi(q^k_{i+s}))$ for $-s \leq i \leq (m - s)$. (ii) For negatively primed elements, it may be possible to perform step towards the origin ${}^k_s K^{(0)}$, i.e.,

$\forall c_2 \rightarrow c_1 \colon c_1 \in {}^k_s K^{(i)} \Rightarrow c_2 \in {}^k_s K^{(i-1)}$ for all $-s < i \leq 0$. (iii) An analogical rule is applied for positively primed elements, i.e., $\forall c_1 \rightarrow c_2 \colon c_1 \in {}^k_s K^{(i)} \Rightarrow c_2 \in {}^k_s K^{(i+1)}$ where $0 \leq i < (m - s)$. The element ${}^k_s K^{(0)}$ of the sequence ${}^k_s F$ then represents the sought set ${}^k_s K$ of possible T's configurations for an instruction in the state q^k_s.

Algorithm 2 Procedure for computation of the mapping $unroll$.

Require: A state $q^k_s \in Q$.
Ensure: Formula ${}^k_s F^\star$ symbolically describes possible configurations of the TS induced by PSG for an instruction in the state q^k_s.

1: $m := \max(\mathbb{S})$
2: ${}^k_s F^\star := {}^k_s K^{\star(-s)} \rightarrow ... \rightarrow {}^k_s K^{\star(-1)} \rightarrow K^\star \rightarrow {}^k_s K^{\star(1)} \rightarrow ... \rightarrow {}^k_s K^{\star(m-s)}$
3: **for** $i \in \mathbb{S}$ **do**
4: $\quad d := i - s$
5: $\quad$ **for** $e \rightsquigarrow b \in \xi(q^k_i)$ **do**
6: $\quad\quad {}^k_s F^\star := {}^k_s F^\star \wedge \begin{cases} e^\star = b & d = 0 \\ {}^k_s e^{\star(d)} = b & \text{otherwise} \end{cases}$
7: $\quad$ **end for**
8: **end for**
9: **return** ${}^k_s F^\star$

An implementation of a procedure computing $unroll$ on the bit-vector logic (BVL) level is demonstrated in the Alg. 2. Here, the * operator is used to represent BVL encoded versions of the elements from the previous description of $unroll$. Particularly, an expression $K^\star$ can be seen as formula that expresses computation of the value of each edge in the PSG, i.e., $\bigwedge_{e \in E} \psi(e)$. We first perform $\max(\mathbb{S})$ unrolls of the transition relation $\rightarrow$ (line 2). We use indexes ${}^k_s$ in sets of edge conditions also for identifiers of BVL variables, so they are unique w.r.t the input state q^k_s. Then, we restrict (lines 5–7) each set of configurations by edge conditions that must hold in particular stage for any k-class instruction.

The $csat$ predicate determines satisfiability of a set of edge conditions $I \subseteq \mathbb{E}$ for the pipeline with partially executed instructions in states $S \subseteq Q$. Its computation can be done by using the previously defined $unroll$ function as:

$$csat(I, S) := \bigcap_{q \in S} unroll(q) \cap \bigcap_{c \in I} \alpha(c) \neq \emptyset$$

One can think of computing $csat$ on the level of BVL as querying a SMT solver for $sat(\bigwedge_{q \in S} unroll(q) \wedge \bigwedge_{e \rightsquigarrow b \in I} e^\star = b)$.

Now, the predicate $csat$ can be exploited in order to compute R^{st}, R^{cl} as well as R^{cf}. Let a set $\bar{\mathbb{S}} \subset \mathbb{S}$, $\bar{\mathbb{S}} := \{s \in \mathbb{S} \mid \exists v \in V_s \colon \varphi(v) = s\}$ contains all non-fictive stages of the analysed processor and $\varrho \colon \bar{\mathbb{S}} \rightarrow V_p$ is a mapping that chooses a representative pipeline register from the given stage. The value of R^{st} can be expressed as

$$R^{st}(s, q^{k_1}_{s_1}, q^{k_2}_{s_2}) := \neg csat(\{\varrho(s).\texttt{st} \rightsquigarrow 0\}, \{q^{k_1}_{s_1}, q^{k_2}_{s_2}\}) \vee \neg csat(\{\varrho(s).\texttt{en} \rightsquigarrow 1\}, \{q^{k_1}_{s_1}, q^{k_2}_{s_2}\}).$$

for all $s \in \bar{\mathbb{S}}$, $s_1, s_2 \in \mathbb{S}$, $k_1, k_2 \in \mathbb{K}$. Thus, we ask whether stall (enable) edge of the representative register $\varrho(s)$ from stage s, $\varrho(s).\mathtt{st}$ ($\varrho(s).\mathtt{en}$), cannot transfer zero (one) value in configurations where the pipeline contains instructions in states $q_{s_1}^{k_1}$ and $q_{s_2}^{k_2}$ of their execution. The use of both stall and enable control edges of the representative $\varrho(s)$ is legal since we have already established the validity of the consistency rules (1) and (4). Likewise, we put

$$R^{cl}(s, q_{s_1}^{k_1}, q_{s_2}^{k_2}) := \neg csat(\{\varrho(s).\mathtt{cl} \rightsquigarrow 0\}, \{q_{s_1}^{k_1}, q_{s_2}^{k_2}\})$$

To compute the value of $R^{cf}(q_{s_1}^{k_1}, q_{s_2}^{k_2})$, we only need to determine whether concurrent presence of instructions in states $q_{s_1}^{k_1}$ and $q_{s_2}^{k_2}$ is possible. Therefore,

$$R^{cf}(q_{s_1}^{k_1}, q_{s_2}^{k_2}) := \neg csat(\emptyset, \{q_{s_1}^{k_1}, q_{s_2}^{k_2}\})$$

For fictive stages $s \in \mathbb{S} \setminus \bar{\mathbb{S}}$, we put the value of all the predicates to $False$.

Example 3. Consider the first hazard case from Example 2. We will demonstrate the reasoning done in order to define value of the predicate $R^{st}(2, q_2^{rw}, q_3^{w})$, that is, whether concurrent presence of two instructions in states q_2^{rw}, q_3^{w} implies stalling of stage 2. Presence of an w-class instruction in state q_3^{w} implies validity of the edge conditions $\xi(q_3^{w}) = \{rf_3.\mathtt{en} \rightsquigarrow 1, rf_3.\mathtt{addr} \rightsquigarrow a\}$ (meaning that a write to register a is enabled). Similarly, existence of rw-class instruction in state q_2^{rw} means that the edge conditions $\xi(q_2^{rw}) = \{rf_2.\mathtt{en} \rightsquigarrow 1, rf_2.\mathtt{addr} \rightsquigarrow a\}$ (enabling reading from register a) to hold.

Now, we take arbitrary representative pipeline storage from stage 2, e.g., register ir. This can be done because the requirement for stalling of all storages of the stalled stage is a subject to verify by the consitency rule (1). Stalling of the stage 2 thus necessary means setting the value of its stall edge $ir.\mathtt{st}$ to "1". The value of $ir.\mathtt{st}$ is computed by *flow* circuit. Therefore, the value of the predicate $R^{st}(2, q_2^{rw}, q_3^{w})$ corresponds to $\mathit{flow}(sa_1, sa_2, den, da) = den \wedge (sa_1 = da \vee sa_2 = da)$. Because $rf_3.\mathtt{en} \rightsquigarrow 1 \in \xi(q_3^{w})$ and both edges $\mathit{flow}.\mathtt{den}$, $rf_3.\mathtt{en}$ have the same source vertex wen and its connection $\mathtt{q}$, one can derive that $\mathit{flow}.\mathtt{den} \rightsquigarrow 1$. Similarly, because $rf_3.\mathtt{addr} \rightsquigarrow a$, $rf_2.\mathtt{addr} \rightsquigarrow a \in \xi(q_3^{w}) \cup \xi(q_2^{rw})$, and the pairs of edges $(rf_3.\mathtt{addr}, \mathit{flow}.\mathtt{da})$, resp. $(rf_2.\mathtt{addr}, \mathit{flow}.\mathtt{sa}_2)$, share common source vertex rW, resp. $[7..4]$, we can state $\mathit{flow}.\mathtt{da} \rightsquigarrow a$ and $\mathit{flow}.\mathtt{sa}_2 \rightsquigarrow a$. Thus, $\mathit{flow}(sa_1, a, 1, a) = 1 \wedge (sa_1 = a \vee a = a)$ and so $R^{st}(2, q_2^{rw}, q_3^{w})$ surely evaluates to true. ◁

4.4.3 Construction of Transition Relation of the Parameterized System

Finally, one can define transition relation Δ of the parameterized system P. Let mappings γ^{st}, γ^{cl}, $\gamma^{cf} : Q \to 2^Q$ returns a set of instruction states so that concurrent appearance of any of its element with an instruction i in state $q_s^k \in Q$ causes i to stall, clear, resp. conflict. Moreover, let $\gamma^{le} : Q \to 2^Q$ be a mapping that assigns each state $q_s^k \in Q$ a set of states where instruction is in the lower or equal stage in comparison to the stage s. The formal definition of the above described mappings is:

$$\gamma^{st}(q_s^k) := \{q \in Q \mid R^{st}(s, q_s^k, q)\}$$
$$\gamma^{cl}(q_s^k) := \{q \in Q \mid R^{cl}(s+1, q_s^k, q) \wedge \neg R^{st}(s, q_s^k, q)\}$$
$$\gamma^{cf}(q_s^k) := \{q \in Q \mid R^{cf}(q_s^k, q)\}$$
$$\gamma^{le}(q_s^k) := \{q_{s_1}^{k_1} \in Q \mid s_1 \leq s \wedge k_1 \in \mathbb{K}\}$$

Note that for successful clearing of instruction in stage s, it is required that the stage s is not stalled at the same time. Otherwise, the instruction would not be cleared.

Now, an instruction stays in the same state q_s^k, i.e., yields a step $q_s^k \to q_s^k$, $s \in \mathbb{S}$, $k \in \mathbb{K}$, iff there is an earlier instruction in a state causing the later instruction to stall or when stage s is already occupied. Formally,

$$\exists j < i \colon G \models q_s^k \to q_s^k \in \Delta \Leftrightarrow G = (\gamma^{st}(q_s^k) \cup \gamma^{le}(q_s^k)) \setminus \gamma^{cl}(q_s^k)$$

An instruction in state q_s^k is canceled, i.e., yields a step $q_s^k \to q_{s+1}^{any}$, $s \in \mathbb{S} \setminus \{\max(\mathbb{S})\}$, $k \in \mathbb{K}$ if there exists an earlier instruction in a state causing the later instruction to clear. More formally,

$$\exists j < i \colon G \models q_s^k \to q_{s+1}^{any} \in \Delta \Leftrightarrow G = \gamma^{cl}(q_s^k)$$

For conflicting case, presence of instruction in state q_s^k is considered as spurious if there exists an earlier instruction in a state so that $R^{cf}(q_{s_1}^{k_1}, q_{s_2}^{k_2})$ is positive. To avoid false alarms, a transition $q_s^k \to q_{s+1}^{any}$, $s \in \mathbb{S} \setminus \{\max(\mathbb{S})\}$, $k \in \mathbb{K}$ is yielded in such a case. Formally,

$$\exists j < i \colon G \models q_s^k \to q_{s+1}^{any} \in \Delta \Leftrightarrow G = \gamma^{cf}(q_s^k)$$

Finally, an instruction can proceed to next stage, i.e., yield a step $q_s^k \to q_{s+1}^k$, $s \in \mathbb{S} \setminus \{\max(\mathbb{S})\}$, $k \in \mathbb{K}$ if none of the above rules is applied. More precisely,

$$\forall j < i \colon G \models q_s^k \to q_{s+1}^k \in \Delta \Leftrightarrow G = Q \setminus \bigcup_{x \in \{st, cl, cf, le\}} \gamma^x(q_s^k)$$

The shown construction of transition relation Δ expects that the parameterized system P will be verified with a fairness assumption that during each change of its configuration every process (instruction) must yield a step.

Initially, any instruction of w, rw as well as any class can enter the pipeline. Therefore, the regular set I of initial states is $(\{w, rw, any\} \times \{0\})^+$. In order to describe bad configurations, we have to determine the number of cycles h, during which an rw-class instruction must not store a value into v_t because it would necessary mean that the written data were incorrectly fetched. From the hazard case, we know that data supposed to be written to v_t are computed in stage s_t, and the computed value is *committed* to v_t in the next cycle, i.e., in stage $s_t + 1$. To ensure that data computed in stage s_t are correct, no write to v_w can occur for $s_t - s_r$ cycles. Otherwise, during the execution of an rw-class instruction, there would be an update of storage v_r that is read by the instruction and thus the instruction would perform computation with incorrect data. Because

a RAW hazard is exhibited only after commitment of the incorrectly fetched data to v_t, the value of h is given by $(s_t + 1) - s_r$. Hence, for w-class instructions, we consider h subsequent states following the one involving write to v_w, i.e., $q^w_{s_w+1}, ..., q^w_{s_w+h}$, as *hazardous*. A configuration is considered as bad if it includes an occurrence of a hazard state followed by a state $q^{rw}_{s_t+1}$ of an rw-class instruction.

4.5 Verification of The Parameterized System

The reachability problem defined by the parameterized system P and by the sets of initial and bad states can be checked using techniques described, e.g., in [2, 3].

Example 4. Consider the first hazard case described in Example 2 to be present in five staged CPU. The execution of a w-class instruction writing to register file port rf_3 is described by a process going through the sequence of states $q^w_0, q^w_1, q^w_2, q^w_3, q^w_4, q^w_5, q^w_6$ where rf_3 is written in state q^w_3 and q_0, q_6 denote initial, resp. final, state. The execution of the rw-class instructions reading from port rf_2 and writing to port rf_3 passed through the sequence of states $q^{rw}_0, q^{rw}_1, q^{rw}_2, q^{rw}_3, q^{rw}_4, q^{rw}_5, q^{rw}_6$. Here, the port rf_2 is read in state q^{rw}_2, and port rf_3 is written in state q^{rw}_3 with its value committed in state q^{rw}_4. Because the value of rf_3 is committed in state q^{rw}_4, i.e., in stage 4, and rf_2 is read in state q^{rw}_2, i.e., in stage 2, the length h is $4 - 2 = 2$ causing states q^w_4, q^w_5 to be hazard states. Thus, the set B of minimal bad configurations is $\{q^w_4 q^{rw}_4, q^w_5 q^{rw}_4\}$. A chosen parametric verification method can then be used to check whether a bad configuration, e.g., $q^{any}_6 q^w_5 q^{rw}_4 q^{any}_3 q^{any}_2 q^{any}_1 q^{any}_0$, is reachable. ◁

5 Experimental Evaluation

We have implemented the above described method in a prototype tool [10] and tested it on five processors: *TinyCPU* is a small 8-bit processor that we mainly use for testing of new verification methods. *SPP8* is an 8-bit ipcore with 3 pipeline stages, 16 general-purpose registers, and a RISC instruction set consisting of 9 instructions. *SPP16* is a 16-bit variant of the previous processor with a more complex memory model. *Codea2* is a 16-bit processor dedicated for signal processing applications [9]. 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. Finally, *DLX5* is a 5-staged 32-bit processor able to execute a subset of the instruction set of the DLX architecture [19] (without floating point instructions). Some of the processors were in multiple variants that differ from each other, e.g., in the way how RAW hazards are avoided or in having some additional instruction specific registers such as a register for storing MSB bits of the product, or flag registers.

We conducted series of experiments on a PC with Intel Core i7-3770K @3.50GHz and 16 GB RAM with results presented in Table 2. The columns give the verified processor, its variant, the time needed for the data flow analysis, the duration of the consistency checking, the time spent by verification of the PSs that are created based on each hazard case, and the overall verification time. The last column represents the number of hazard cases that had to be verified during the model verification phase. Note

Table 2: Verification times.

Processor / variant		Data flow analysis [s]	Consistency checking [s]	Par. Mod. verif. [s]	Total time [s]	Hazard Cases [#]
TinyCPU	S	8	1	12	21	9
	SF	10	3	21	34	19
	B	8	1	13	22	9
SPP8	S	28	6	62	96	40
	B	29	6	67	102	40
SPP16	S	34	7	89	130	58
	B	33	7	97	137	58
Codea2	SFH	162	18	856	1036	281
DLX5	S	77	26	378	481	43
	B	123	29	590	742	44

S stalling logic B bypassing logic F flag register(s) H special register(s)

that each hazard case represents a separate task so the part of model verification can be parallelized.

By verifying the above processors, we identified a flaw in a RAW hazard resolution when accessing of the data memory in a development version of the SPP8 processor.

6 Conclusion

We have presented an approach that combines data-flow analysis and methods for formal verification of PSs in order to discover incorrectly handled RAW hazards in the RTL implementation of pipelined microprocessors. The approach was developed with the aim to be highly automated, not requiring any additional efforts from the developers (apart from specifying the architectural registers). We have implemented the approach and successfully tested it on several non-trivial microprocessors where the approach was able to discover previously unknown flaws caused by unhandled hazards.

In the future, we plan to complement the approach proposed in the paper by techniques suitable for verification of other processor features, such as write-after-write and control hazards. This is motivated by our general idea of trying to split processor verification into several simpler, specialised tasks.

Acknowledgement: This work was supported by the Czech Science Foundation under the project 14-11384S, the EU/Czech IT4Innovations Centre of Excellence project CZ.1.05/1.1.00/02.0070, and the internal BUT projects FIT-S-12-1 and FIT-S-14-2486.

References

1. M. D. Aagaard. A hazards-based correctness statement for pipelined circuits. In *Proc. of Correct Hardware Design and Verification Methods (CHARME'03)*, volume 2860 of *LNCS*, pages 66–80. Springer, 2003.

2. P. A. Abdulla, F. Haziza., and L. Holik. All for the price of few (parameterized verification through view abstraction). In *Proc. of Verification, Model Checking, and Abstract Interpretation (VMCAI'13)*, volume 7737 of *LNCS*, pages 476–495. Springer, 2013.
3. A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. In *Proc. of 16th International Conference on Computer Aided Verification (CAV'04)*, volume 3114 of *LNCS*, pages 197–202. Springer, 2004.
4. R. Brummayer and A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Proc. of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, volume 5505 of *LNCS*, pages 174–177. Springer, 2009.
5. J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Proc. of Computer Aided Verification (CAV'94)*, volume 818 of *LNCS*, pages 68–80. Springer, 1994.
6. L. Charvat, A. Smrcka, and T. Vojnar. Automatic formal correspondence checking of ISA and RTL microprocessor description. In *Proc. of Microprocessor Test and Verification (MTV'12)*, pages 6–12. IEEE, 2012.
7. E. Clarke, M. Talupur, and H. Veith. Environment abstraction for parameterized verification. In *Proc. of Verification, Model Checking, and Abstract Interpretation (VMCAI'06)*, volume 3855 of *LNCS*, pages 126–141. Springer, 2006.
8. Codasip Ltd. *CodAL Architecture Description Language*, 2013.
9. Codea2 Core IP in Codasip Studio. `www.codasip.com/products/codea2/`, 2013.
10. Hades Hardware Verification Tool. `www.fit.vutbr.cz/research/groups/verifit/tools/hades/`, 2014.
11. K. Hao, S. Ray, and F. Xie. Equivalence checking for function pipelining in behavioral synthesis. In *Proc. of Design, Automation and Test in Europe (DATE'14)*, pages 1–6. IEEE, 2014.
12. R. B. Jones, C. H. Seger, and D. L. Dill. Self-consistency checking. In *Proc. of Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *LNCS*, pages 159–171. Springer, 1996.
13. A. Koelbl, R. Jacoby, H. Jain, and C. Pixley. Solver technology for system-level to RTL equivalence checking. In *Proc. of Design, Automation and Test in Europe (DATE'09)*, pages 196–201. IEEE, 2009.
14. U. Kuhne, S. Beyer, J. Bormann, and J. Barstow. Automated formal verification of processors based on architectural models. In *Proc. of Formal Methods in Computer-Aided Design (FMCAD'10)*, pages 129–136. IEEE, 2010.
15. P. Mishra, H. Tomiyama, N. Dutt, and A. Nicolau. Automatic verification of in-order execution in microprocessors with fragmented pipelines and multicycle functional units. In *Proc. of Design, Automation and Test in Europe (DATE'02)*, pages 36–43. IEEE, 2002.
16. L. De Moura and N. Bjorner. Z3: An efficient SMT solver. In *Proc. of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
17. K. S. Namjoshi. Symmetry and completeness in the analysis of parameterized systems. In *Proc. of Verification, Model Checking, and Abstract Interpretation (VMCAI'07)*, volume 4349 of *LNCS*, pages 299–313. Springer, 2007.
18. M. Ngyuen, M. Thalmaier, M. Wedler, J. Bormann, D. Stoffel, and W. Kunz. Unbounded protocol compliance verification usign interval property checking with invariants. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 27(11), 2008.
19. D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware / Software Interface*. Morgan Kaufmann, Boston, fourth edition, 2012.
20. M. N. Velev and P. Gao. Automatic formal verification of multithreaded pipelined microprocessors. In *Proc. of International Conference on Computer Aided Design (ICCAD'11)*, pages 679–686. IEEE, 2011.