BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

AUTOMATED VERIFICATION IN HW/SW CO-DESIGN

AUTOMATICKÁ VERIFIKACE V PROCESU SOUBEŽNÉHO NÁVRHU HARDWARE A SOFTWARE

PHD THESIS
DISERTAČNÍ PRÁCE

AUTHOR — **Ing. LUKÁŠ CHARVÁT**
AUTOR PRÁCE

SUPERVISOR — **Prof. TOMÁŠ VOJNAR, Ph.D.**
ŠKOLITEL

CO-SUPERVISOR — **Ing. ALEŠ SMRČKA, Ph.D.**
ŠKOLITEL SPECIALISTA

BRNO 2019

Abstract

The subject of the thesis is to design new hardware verification techniques optimized for a process of HW/SW co-design in which hardware and software are developed in parallel to speed up the development of new embedded systems. Currently, microprocessor co-design tools typically allow to verify designs by simulation and/or functional verification. However, even extensive functional verification can miss some non-trivial bugs. Therefore, formal verification has become more and more desirable in recent years. As opposed to testing and bug-hunting techniques that only aim at detecting flaws, the goal of formal verification is to rigorously prove that the system is indeed correct. Formal verification is, however, a very demanding task, and even though a lot of progress has been achieved in this area, formal verification is far from being able to fully automatically check all relevant properties of complex designs without a significant and costly human involvement in the verification process. The thesis deals with these challenges by focusing on verification techniques based on formal approaches, but possibly relaxing or limiting their precision and generality to achieve full automation. Further, the thesis also focuses on the efficiency of the proposed techniques and their ability to deliver continuous feedback about the verification process. Special attention is devoted to the development of formal methods for checking the equivalence of microprocessor designs on various levels of abstraction. Although these designs cannot be behaviorally equivalent, they are required to give mutually corresponding results when executing the same input program, which is a property difficult to achieve. As another considered topic, the thesis proposes methods for checking correctness of mechanisms preventing data and control hazards in single-pipelined implementations of microprocessors. The approaches described in this thesis has been implemented in the form of several tools which, after examining designs of multiple pipelined microprocessors, were able to deliver promising experimental results.

Abstrakt

Předmětem dizertační práce je návrh nových technik pro verifikaci hardwaru, které jsou optimalizovány pro použití v procesu souběžného vývoje hardwaru a softwaru. V rámci tohoto typu vývoje je hardware spolu se software vyvíjen paralelně s cílem urychlit vývoj nových systémů. Současné nástroje pro tvorbu mikroprocesorů stavějící na tomto stylu vývoje obvykle umožňují vývojářům ověřit jejich návrh využitím různých simulačních technik a/nebo za pomoci tzv. funkční verifikace. Společnou nevýhodou těchto přístupů je, že se zaměřují pouze na hledání chyb. Výsledný produkt tedy může stále obsahovat nenalezené netriviální defekty. Z tohoto důvodu se v posledních letech stává stále více žádané nasazení formálních metod. Na rozdíl od výše uvedených přístupů založených na hledání chyb, se formální verifikace zaměřuje na dodání rigorózního důkazu, že daný systém skutečně splňuje požadované vlastnosti. I když bylo v uplynulých letech v této oblasti dosaženo značného pokroku, tak aktuální formální přístupy nemají zdaleka schopnost plně automaticky prověřit všechny relevantní vlastnosti verifikovaného návrhu bez výrazného a často nákladného zapojení lidí v rámci verifikačního procesu. Tato práce se snaží řešit problém s automatizací verifikačního procesu jejím zaměřením na verifikační techniky, ve kterých je ale záměrně kladen menší důraz na jejich přesnost a obecnost za cenu dosažení plné automatizace (např. vyloučením potřeby ručně vytvářet modely prostředí). Dále se práce také zaměřuje na efektivitu navrhovaných technik a jejich schopnost poskytovat nepřetržitou zpětnou vazbu o verifikačním procesu (např. v podobě podaní informace o aktuálních stavu pokrytí). Zvláštní pozornost je pak věnována vývoji formálních metod ověřujících ekvivalenci návrhů mikroprocesorů na různých úrovních abstrakce. Tyto návrhy se mohou lišit ve způsobu, jakým jsou vnitřně zpracovány programové instrukce, nicméně z vnějšího pohledu (daného např. obsahem registrů viditelných z pozice programátora) musí být jejich chování při provádění stejného vstupního programu shodné. Jako další téma se práce dále věnuje návrhu metod pro verifikaci správnosti mechanismů zabraňujících výskytu datových a řídících hazardů v rámci linky zřetězeného zpracování instrukcí. Veškeré metody popsané v této práci byly implementovány ve formě několika nástrojů. Aplikací těchto nástrojů pro verifikaci návrhů netriviálních procesorů bylo dosaženo slibných experimentálních výsledků.

Keywords

Formal Verification, Microprocessor, Hardware / Software Co-design, Architecture Description Language, RTL-ISA Equivalence Checking, Pipeline Hazard, Parametric Systems.

Klíčová slova

Formální verifikace, mikroprocesor, souběžný návrh hardware a software, jazyk pro popis architektury, formální ověřování ekvivalence, hazardy v lince zřetězení, parametrické systémy.

Reference

CHARVÁT, Lukáš. *Automated Verification in HW/SW Co-design.* Brno, 2019. PhD thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Prof. Tomáš Vojnar, Ph.D. Co-supervisor Ing. Aleš Smrčka, Ph.D.

Automated Verification in HW/SW Co-design

Declaration

Hereby I declare that this PhD thesis was prepared as an original author's work under the supervision of Prof. Tomáš Vojnar and Dr. Aleš Smrčka. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

....
Lukáš Charvát
August 9, 2019

Acknowledgements

I would like to thank my supervisors Prof. Tomáš Vojnar and Dr. Aleš Smrčka for their valuable comments, suggestions, and inspiring consultations during the supervision of this work. Further, I thank all former and current members of the VeriFIT group for their advice, fruitful discussions, and friendly working environment. Last but not least, I would like to thank my family, especially my wife, and friends for their endless support, strong motivation, and ceaseless patience.

The work presented in this thesis was supported by the Czech Science Foundation (under the project P103/10/0306), the Czech Ministry of Education (under the projects COST OC10009, MSM 0021630528, 14-11384S), the Czech Ministry of Industry and Trade (under the project FR-TI1/038), the EU/Czech IT4Innovations Centre of Excellence project CZ.1.05/1.1.00/02.0070, the IT4IXS: IT4Innovations Excellence in Science project LQ1602, and the BUT projects FIT-S-11-1, FIT-S-12-1, FIT-S-14-2486.

Contents

Chapter 1

Prologue

Embedded systems are massively deployed in almost every electronic device that we now use in our everyday life. For embedded systems, customized *application-specific instruction-set processors* (ASIPs) are often designed. These processors have specific functions of hardware available through special instructions in order to achieve required performance criteria and low power consumption. A significant part of embedded system costs includes prices that are required for (i) design of hardware architecture, (ii) its physical realization, and (iii) design of software.

If we consider costs of the physical realization as fixed, the only way for further lowering of the price of an embedded system is to reduce the time that is needed for the design of hardware and software. In order to achieve that, the trend is to develop both hardware and software in parallel in a process of the so-called *hardware/software co-design*. The automation of common tasks that are a part of the co-design process is another crucial factor for successful and fast development. To facilitate automation, specialized *architecture description languages* (ADLs) are frequently utilized during the microprocessor design process. Specifically, in the case of microprocessor design, various integrated frameworks [125, 28, 1] take advantage of the availability of the high- and low-level ADL descriptions and provide automatic generation of hardware description language (HDL) designs and tool-chains including, e.g., simulators, assemblers, disassemblers, and compilers.

In the current microprocessor design frameworks, an initial understanding about the design (e.g., to see whether an instruction set contains enough instructions, to check the performance of the design) is done by simulation. After this step, verification of the designs is typically performed. Currently, simulation-based approaches such as testing and functional verification are very popular. Testing is based on the observation of the behavior of the verified system in a limited number of situations (e.g., for cases considered as crucial by the designer) and, therefore, it provides only a partial guarantee of the system's correctness. Functional verification automates the testing process by generating a set of constrained/random test vectors and by comparing the behavior of the system for these vectors with the behavior specified by a reference model, the so-called *golden specification*, which must be provided manually by the developers. However, even extensive functional verification, like any other bug-hunting technique, can still miss non-trivial bugs. Therefore, the use of formal verification is very desirable. Its goal is to rigorously prove that the system is indeed correct. That is, if no issue is found by a formal method, the system is guaranteed to conform to the given specification. Unfortunately, formal verification is not a common part of the current microprocessor design frameworks.

Formal methods can be categorized into three basic categories (with not completely sharp boundaries): *theorem proving*, *static analysis*, and *model checking*. Theorem proving, also called *deductive verification*, is based on deducing properties of a verified system from various logical axioms and assumptions about the system. The process often requires a significant manual intervention. Static analysis attempts to avoid execution of the system being examined, and instead analyses and gathers approximate (and often conservative) information about the system from the source code, and thus it may produce many false alarms. Model checking systematically explores the state space of the examined system. Unlike in static analysis, if some abstraction is used, it typically comes with an automated refinement technique that allows the approach to automatically exclude spurious counterexamples to the verified properties.

An ideal formal approach should be sound and complete, so an error is reported if and only if there is a real error in a system, otherwise the system is said to be correct. Moreover, the approach should be fully automated and terminating. Satisfying these ideal properties is, however, very costly (or impossible if a source of unboundness such as parametrization is involved) due to the state explosion problem that is usually hit (or due to the implied undecidability for the case of unbounded state spaces). To provide efficiency and high automation, completeness or even soundness are sometimes sacrificed leading to error detection methods built on formal roots. Such a method may be still quite useful as it can discover flaws that would stay hidden otherwise, which is most often caused due to a different way of state space traversal.

Aim of The Thesis. In accordance with the above, the thesis aims at developing new verification techniques with formal roots with an emphasis on full automation (without a need to manually create models of the environment of the verified system), efficiency, and ability to deliver continuous feedback, e.g., actual coverage about the verification process. Within the thesis, special attention is devoted to the development of formal methods that check the equivalence of designs on various levels of abstraction. These designs cannot be behaviorally equivalent (due to their different abstraction level), but they are required to give mutually corresponding results when executing the same input program, which is a property difficult to achieve. Another considered topic is development of methods for checking correctness of mechanisms preventing data and control hazards in pipelined implementations of microprocessors. The above-described techniques should, in particular, be optimized for the class of ASIPs broadly used in light-weight embedded devices.

As the first step towards the aim, we focused on automatic checking of correspondence of *instruction-set-architecture* (ISA) and *register-transfer-level* (RTL) descriptions of a microprocessor. The correspondence means that after starting in the same initial states of resources (such as registers, memories, and devices connected to the microprocessor) and executing the same program, both models will always end up in states in which the resources have equivalent contents. The ISA (instruction-accurate, high-level) description captures the behavior of an instruction without consideration of complex parts (such as pipelines, buses, etc.) that are part of the RTL (cycle-accurate, low-level) specification. The existence of ISA description in early phases of processor development is critical because it allows one to generate the previously mentioned tool-chains that are necessary to create software when its RTL description is still being designed. Because the software is created over a model that is different from the one delivered with the final product, conformance of these two models must be guaranteed. The correspondence checking can be also useful if the RTL specification is automatically generated from the ISA description to verify the correctness of such a generator.

Regarding the correspondence checking topic, in [31, 32], we proposed a novel technique that copes with this problem, although not taking the influence of complex parts of the processor (pipelines, buses, etc.) into account. Even with this simplification, one has to deal with the large bit-width of registers and size of memories and register files. The proposed approach deals with this problem by using abstraction and reduction techniques that are described later in this thesis. The approach has been experimentally implemented within Codasip IDE [1] and successfully tested in several case studies. The experiments include a non-trivial single-pipelined processor in which the approach revealed three previously unknown bugs. The experiments also show that instructions of single-pipelined processors can be verified within seconds.

Further, we have extended the above-proposed correspondence checking by another verification phase devoted to the verification of the so-called pipeline hazards. Hazards in the instruction pipeline are problems caused by inadequate synchronisation of earlier and later instructions running concurrently through the pipeline that may cause potential corruption of the data used by the instructions. Three common types of pipeline hazards are data, control, and structural hazards. In the thesis, we focus on the first two of them. An example of such a hazard is the so-called *read-after-write* (RAW) data hazard. Here, a later-started instruction uses data supposed to be produced by an earlier-started instruction, but the earlier instruction has not yet managed to proceed far enough in the pipeline to write the data into the storage used by the later instruction. The later instruction then stores a potentially wrong result of its execution, obtained by dealing with the obsolete data.

To address these issues, in [34, 35, 36, 37], we propose a novel, highly-automated approach for discovering the above-listed kinds of hazards within in-order pipelined instruction execution. The approach combines (i) static analysis of data paths to detect anomalies and possible hazards, followed by (ii) a transformation of detected problematic paths to a parametric system, and (iii) a subsequent formal verification using techniques for formal verification of the parametric systems. The approach has been implemented in a tool called Hades [37] and, in this thesis, we present promising experimental results applying the tool to multiple pipelined microprocessors.

Outline. The rest of the thesis is organized as follows. Chapter 2 gives an overview of microprocessor architectures together with an introduction to the former and contemporary techniques used during the design of embedded systems. Chapter 3 briefly describes the most common architecture description languages and frameworks for processor design. Chapter 4 is an introduction to selected topics of formal verification. Chapter 5 provides an overview of related work in the field of microprocessor verification. Chapter 6 presents the main goals of the thesis. Chapter 7 describes our newly proposed technique for automatic generation of abstract models of memories that can be used for efficient formal verification of hardware designs. Next, Chapter 8 presents a new automated approach built on a formal basis that we use for checking correspondence between an RTL implementation of a microprocessor and its ISA description. Further, Chapter 9 describes our novel technique utilizing static analysis of data paths and formal verification of parameterized systems in order to discover flaws caused by improperly handled pipeline hazards. Finally, Chapter 10 concludes the thesis.

Chapter 2

Embedded System Design

Since the last decades of the 20th century, one can observe the ever-increasing popularity of *built-in systems* such as (smart) TVs, cell phones, entertainment systems, or network-connected devices. This caused a significant increase in demand for *embedded systems*. By the embedded system, we typically mean a combination of hardware and software together with other mechanical components intended to perform a dedicated function (often) in real-time computing constraints. Embedded systems often reside in machines that are expected to run continuously for years without errors and (in certain cases) recover autonomously if an error occurs. Today, it is very common that a final product consists of several co-operating but individually designed embedded systems [107, 91].

As the capabilities of the embedded systems are still growing, they are now widely deployed across multiple fields. For instance, the use of embedded systems in the automotive industry allowed the implementation of complex algorithms (e.g., in fuel injection) which resulted in lower emissions and higher fuel efficiency. The higher computing power of embedded devices also helps in airplane tracking and navigation systems which now allow for safe landing even in adverse weather conditions. Another example comes from the automated household control industry. Here, the recent development of the so-called *Internet of Things* (IoT) enabled smart control of home temperature control systems via connected thermostats. Besides the fact that such a thermostat can be controlled remotely via a mobile application, it can also learn the owner's typical day-to-day behavior (e.g., working hours, weekend routines) and perform heating/cooling optimization in order to lower household running costs.

The above-mentioned rapid evolution of the embedded systems has been largely sustained by research and innovation in the field of system design methodologies. The co-operated design of both hardware and software, the so-called *hardware/software co-design*, is one of them. Even though it is not a new discipline (as since the era of the first computers, designers have always considered mutual dependence between hardware and software), the growing complexity of the embedded systems, increasing time-to-market pressure, and system costs bring new challenges for the co-design methodology [91]. A significant part of these challenges can be overcome by design automation. This translates to an increased demand for development of new co-design tools that would speed up the implementation and verification tasks.

To provide necessary background, the following sections of this chapter describe some of microprocessor and hardware architectures that are typically used in the embedded devices. The last section then discusses how the HW/SW co-design methodology can help to find the most suitable microprocessor for the given task within a short time and at a low cost.

2.1 General-Purpose Microprocessors

The first embedded systems based on microprocessors started to appear in the 1960s. A well-known example of such a system is Apollo Guidance Computer [55]. In early stages, the embedded systems were produced in series counting only limited number of units. An early example of a mass-produced system is the D-17 guidance computer used for navigation of Minuteman I intercontinental ballistic missiles [88]. Due to the mass production, the price of microprocessors had fallen which led to their spread across a wide spectrum of industry sectors. Now, microprocessors can be found in almost any electronic device.

From the component point of view, a very basic microprocessor consists of the following main parts: (i) internal memory (register files, cell memory), (ii) an arithmetic logical unit (ALU), and (iii) the control unit [108]. The microprocessor registers can be typically split into one of the following categories: general-purpose registers (GPRs), index registers (IRs), and the program counter (PC). The GPRs are used to store temporary data within the microprocessor. The IRs modify operand addresses during the run of a program, typically for doing vector and/or array operations. In the case of the Von Neumann memory organization, program and computational data are commonly stored in a single memory whereas, in the case of the Harvard architecture, the program code is kept separate from the program data. The PC is an index register that contains the address (location) of the instruction being executed at the current time. The purpose of the ALU is then to perform arithmetic and logical operations on source data. The data sourcing and their transfer to the ALU inputs are performed by the control unit which controls flow inside the processor. Besides the data flow, the control unit also contains components built around the PC register which are responsible for loading (i.e., fetch logic) and decoding instructions (i.e., instruction decoder).

Each microprocessor can execute a set of instructions. The instruction set typically reflects the structural, functional, and operative principles of the processor. The most influential factors that have an impact on the microprocessor instruction set are the following: (i) processor registers, (ii) size of memory units (data types), (iii) addressing modes, (iv) memory architecture (e.g., Von Neumann vs Harvard), (v) interruption and exception handling [107, 91].

In the pioneer era of microprocessor development, almost every processor has its own instruction set. Therefore, programs written for a particular microprocessor were only hardly portable to another processor. Over the last decades several standardized instruction sets emerged, for instance, `i386` [68], `amd64` [7], `armv7` [8], or `riscv` [115]. The contemporary general-purpose microprocessors use the same set of instructions, even if their inner design is often entirely different. While still maintaining the same instruction set, modern microprocessors build on additional concepts, such as instruction pipelines, branch prediction, and/or microinstruction architecture to better fulfill performance expectations.

The processor pipelining means splitting the overall execution of the instruction into smaller parts named *execution stages*. This is particularly useful, for example, in a situation when one clock pulse latches a value into a register or begins a calculation and it takes too much time for the value to be stable at the outputs of the register or for the calculation to complete. As the number of pipeline stages grows, a given stage can be implemented with simpler circuitry, which may let the processor clock run faster [121].

Almost all pipelined processors do (at least simple) branch prediction because they have to speculatively fetch the next instruction before the current instruction is finished [107]. The prediction is typically handled by a circuit known as a branch predictor. This part

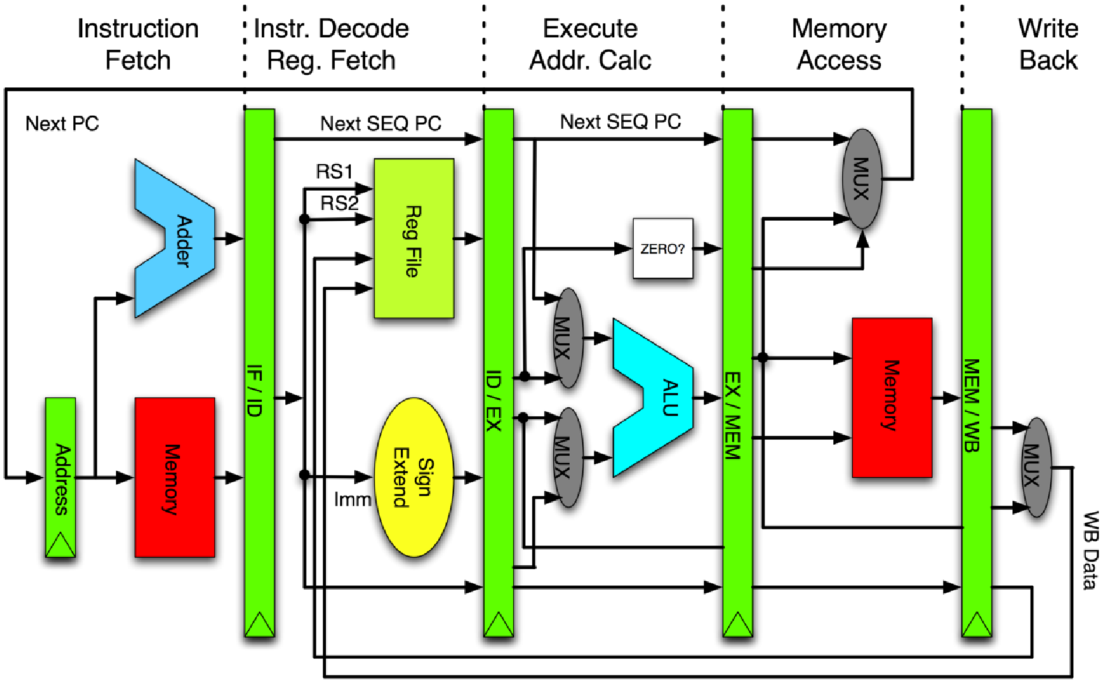


Figure 2.1: A typical organization of a simple microprocessor with a single pipeline.

of a processor determines whether a conditional branch (jump) in the instruction flow of a program is likely to be taken or not. Therefore, branch predictors are important in today's modern processors to achieve high performance.

The microprogram architecture is a type of microprocessor architecture where high-level instructions are performed by executing several lower-level instructions (microinstructions). The microprogram architecture firstly appeared in [138, 139]. Soon after, the instructions had become so complex that the use of the so-called microprogram controller became inevitable. Later, the opposite concept of a *reduced instruction set computer* (RISC) appeared. It used simple instructions and avoided the need for the microprogrammed controller. However, it subsequently merged with a *complete instruction set computer* (CISC) paradigm and microprogrammed controllers started to be used more frequently again. The main advantages of the microprogrammed architectures are that new (high-level) instructions may be added quickly and that developers can fix certain design errors in the instruction processing just by changing the underlying microinstructions [108, 91]. In the thesis, we will further assume that all presented models/examples are on the microinstruction level if not stated otherwise.

Taken all together, the typical organization of a simple microprocessor with a single pipeline is shown in Fig. 2.1. In such a microprocessor, instructions are processed in the next described steps. First, the instruction is loaded from the program memory. Then it is decoded to an operation code (opcode) and an address section. The opcode identifies the operation to be performed (e.g., addition, multiplication) while the address part contains the operand specification or immediate value. These operands can be registers, memory addresses, input ports, etc. In the third stage, which is often called the *execution stage*, result values and memory access addresses are calculated according to the opcode. Next, in the *memory access stage*, the data memory is read and/or written. Finally, in the *write-back stage*, the registers are written.

From the point of view of embedded systems, the use of general-purpose microprocessors is advantageous for several reasons. Most of the benefits come from the fact that the microprocessor itself represents a universal calculation unit. This allows the same microprocessor to be used for various computation required in different embedded systems. Moreover, extending design with additional connections to other parts of the system can be quickly made using existing solutions which greatly reduces the time required for system design. Finally, one of the biggest benefits is a variety of available well-documented and tested software tools that support program development (such as compilers and debuggers) [91]. Thus, especially in the case of lower production volumes, the use of a general-purpose microprocessor is typically less costly than designing an application-specific integrated circuit or an application-specific instruction-set processor (that are described more in the next sections).

The universal nature of the general-purpose microprocessors could be, however, also their main disadvantage. In specialized applications (e.g., video filtering), the general-purpose microprocessors typically have lower performance and higher energy consumption when compared to specifically crafted circuits or processors.

2.2 Application-Specific Integrated Circuits

The so-called *application-specific integrated circuits* (ASICs) are the opposite of the universal architectures. They are made for a particular purpose to meet the challenging design constraints typically given in terms of performance, energy consumption, and chip size. The downside is the high cost and time consumption required for their design. Thus, the use of ASIC is especially viable for mass production where development costs are distributed among a large number of manufactured units [91].

In the 1980s, much effort was invested to find a technology which would be easy and reliable enough to be practically used in application-specific systems. One of the first technologies of this type was *Uncommitted Logic Array* (ULA) [113] which is a chip consisting of basic building blocks (i.e., standard logic cells or gateways) that can perform basic calculations. Customization of the chip is done by modification of a metal mask which connects the individual parts that can be achieved, for instance, by breaking certain connections. As the technology evolved, the number of gates on the chip rapidly rose to allow the development of very complex circuits on a single chip.

The ASIC design process is rather complex. It can be roughly divided into the following steps. The first step consists of a specification of the system requirements. Then, a model of the system is created. It is usually described by the language appropriate for system design, the so-called *hardware description language* (HDL) such as VHDL [66] or Verilog [65]. The model is verified whether it meets the original requirements (typically using simulation). If the verification is successful, one can process with a synthesis of the ASIC logic. The design is converted into a set of basic building blocks (standard cells or gateways) of the logic array. These building blocks are then mapped on the logic array. After that, interconnections are created to form the final design. Next, the ASIC is analyzed whether the final system works like expected (i.e., whether the specification criteria are still met). Finally, masks are fabricated and the manufacturing of the circuit can begin [91].

Although ASICs typically dominate in the terms of speed and power efficiency, their building costs are becoming more and more prohibitive mostly because the design cost and longer time-to-market period cannot be amortized over multiple applications.

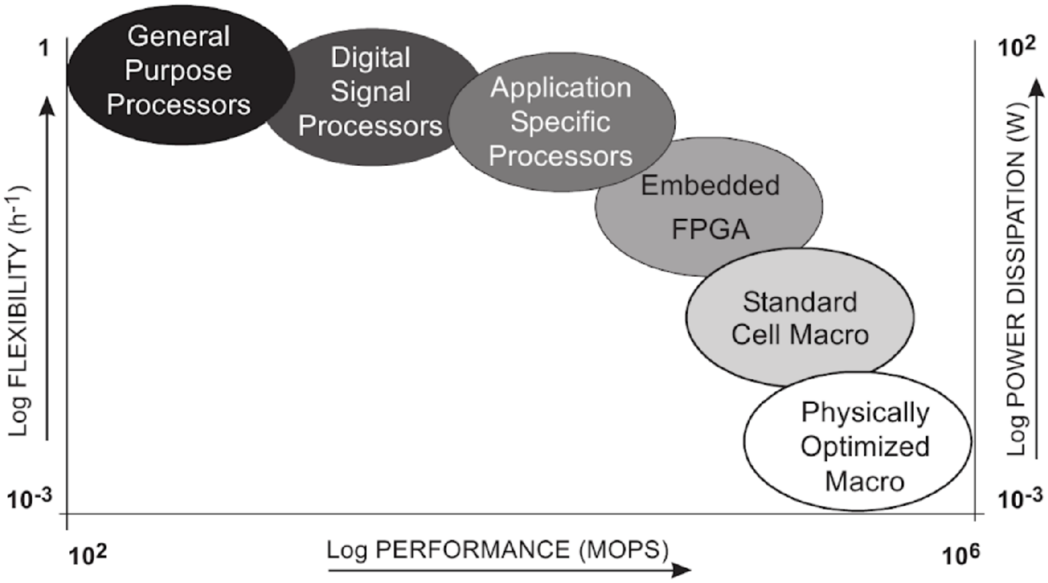


Figure 2.2: Trade-off between flexibility and performance among various components used in embedded systems. Source: [119].

2.3 Application-Specific Instruction-Set Processors

The instruction set of an *application-specific instruction-set processor* (ASIP) is built in a way so it benefits a specific application by the ability to perform specific operations through special instructions. In general, components of an ASIP can be divided into two parts: (i) logic which is able to execute some well-known instruction set and (ii) specific logic, which can be configurable per application, that is accessed via newly introduced instructions [52]. The specific logic can be then placed in a dedicated component (e.g., ASIC) or in the programmable field (such as FPGA). As can be seen in Fig. 2.2, the splitting of the microprocessor components into these two parts provides a good trade-off between the flexibility of a general-purpose microprocessor and the ASIC's performance and low power consumption.

Because of the above-mentioned properties, ASIPs provide an attractive approach in a growing number areas of embedded systems, for example, as an alternative to hardware accelerators for video coding [59] or signal processing [120].

2.4 Modern Hardware/Software Co-Design

As was discussed in the previous sections, the current microprocessor design cycle strives to find the most suitable microprocessor (often in the form of an ASIP) for the target application within a short time and at a low cost. Due to this time-to-market pressure and short product life-cycle, a rapid exploration and evaluation of candidate architectures is an essential need. Hardware description languages (HDLs), such as VHDL or Verilog, are commonly used for hardware design, modeling, and simulation. However, a microprocessor specified only in HDL does not include all necessary information about assembler syntax, binary encoding of instructions, etc. This is the reason why specially crafted *architecture description languages* ADLs were introduced [92].

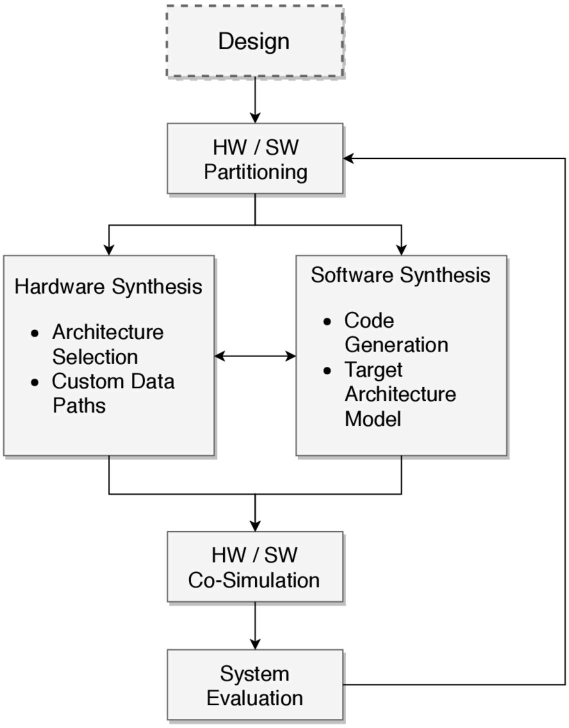


Figure 2.3: A generic hardware/software co-design methodology. Source: [140].

An ADL together with a microprocessor integrated development environment (IDE) and an appropriate tool-set helps the designer to quickly find a microprocessor that optimally splits computation tasks between hardware and software. ADLs are used to specify processor and memory architectures and to automatically generate a software toolkit including compiler, simulator, assembler, profiler, and debugger. Moreover, there are ADLs that can describe microprocessors on several levels of abstraction. With such an ADL, it is then possible to start writing the target (application) programs even before the low-level (RTL) description of the processor exists, because much simpler high-level (ISA) description often suffices to generate compilers, debuggers and simulators.

Fig. 2.3 shows a common exploration co-design flow [140]. Tasks computed by the system are partitioned between hardware and software. The application programs are compiled and simulated, and the feedback is used to modify the ADL specification with the goal of finding the best possible architecture for the given set of application programs under various design constraints such as area, power, and performance. Because of the short time that is typically allowed for design and implementation, bugs can be introduced in the microprocessor, and thus the candidate designs have to be verified whether they still comply with the original specification. The required time savings are then accomplished by automation of these tasks that would otherwise have to be done manually (such as the tool-chain and/or the HDL representation generation).

Since ADLs play a key role in the modern hardware/software co-design, the next chapter describes and classifies them in a more detail together with their accompanying tools.

Chapter 3

Architecture Description Languages

This chapter describes the expressive power of the contemporary ADLs together with microprocessor development frameworks that are based on them. Further, the chapter also points out possible verification options offered by the frameworks. Please note that the following list intentionally does not represent an exhausting overview of the ADLs and frameworks, but it should give the reader an idea about the environment in which the proposed verification methods are supposed to be integrated. Moreover, since the verification techniques proposed in the thesis aim to be automated as much as possible, it is also important to observe which information is usually part of the microprocessor descriptions and what kind of information would have to be provided externally.

As it is discussed in [92, 67, 93], hardware ADLs can be divided into three categories: (i) *structure-oriented*, (ii) *instruction-set-oriented*, and (iii) *mixed*. The level of abstraction in structure-oriented ADLs is close to the RTL. Such a description typically misses high-level information. Therefore, extraction of, e.g., an assembly language is a quite hard task. On the opposite side, instruction-set-oriented ADLs are close to the ISA level. They lack cycle-accurate information, and thus they usually cannot be used for hardware synthesis. They are mainly manufactured for use in retargetable compilers which are compilers/decompilers that are designed to be relatively easy to modify and to generate/decompile code for various instruction-set architectures. Mixed ADLs try to overlap the gap between the two former approaches by adding the missing pieces of information.

3.1 Structure-Oriented ADLs

The structural ADLs capture the structure in terms of architectural components and their connectivity. Structural ADLs enable flexible and precise architecture descriptions. The same description can be used for hardware synthesis, test generation, simulation, and compilation. However, it is difficult to extract the instruction set without restrictions on a description style. Therefore, the structural ADLs traditionally find their use more for hardware generation than in compilers [92]. In this Section, MIMOLA [80] ADL is briefly described.

3.1.1 MIMOLA

The machine-independent microprogramming language (MIMOLA) is one of the first languages specifically designed for synthesis and not just for the hardware simulation. This

Figure 3.1 A MIMOLA example showing the description of a multifunctional ALU module. Source: [80].

```
MODULE ALU(IN operation: (1:0);
           IN a: (31:0);
           IN b: (31:0);
           OUT result: (31:0);)
CONBEGIN
    result <- CASE operation OF
        0: a + b;
        1: a - b;
        2: a AND b;
        3: b;
    END;
CONEND;
```

approach avoided time-consuming considerations caused by differences between synthesis and simulation semantics (i.e., checking whether the simulated design is within a synthesizable subset). The ADL-driven synthesis used in *MIMOLA Software System* (MSS) was among the first approaches of its kind.

The major advantage of MIMOLA is that the same description can be used for synthesis, simulation, test generation, and compilation. A toolchain including a hardware synthesizer, a code generator, a self-test program compiler, a functional simulator, and an RTL simulator were all developed based on the MIMOLA language [80]. The description of the microprocessor in MIMOLA ADL consists of the following three parts: (i) the algorithm to be compiled (application program), (ii) the target processor model, and (iii) the additional linkage and transformation rules.

The algorithmic part of a processor description in MIMOLA is an extension of PASCAL. Unlike other high-level languages (e.g., C or PASCAL), it allows references to physical registers and memories. It also allows usage of hardware components in the form of procedure calls. For example, if the processor description contains a component named `ALU` (arithmetical-logical unit), programmers can write segments like `result := ALU(op, a, b)` to get the result of the mathematical operation given by the operation `op` which is performed by the multifunctional `ALU` component.

The target processor model is then described using modules and connections. Modules describe the behavior of hardware components. In MIMOLA, each module is specified by its port interface and its behavior. Similarly to VHDL, several predefined, primitive operators exist. Example 3.1 shows the description of a multifunctional ALU module. In the example, the `CONBEGIN`/`CONEND` construct denotes a set of concurrent assignments. Within the assignment block, a conditional assignment to output port `result` is specified, which depends on the two-bit control input `operation`. The microprocessor structure is then formed by connecting ports of module instances. For example, a MIMOLA description shown in Fig. 3.2 connects two modules: (i) the arithmetic-logic unit `ALU` and (ii) the accumulator `ACC`.

Finally, the linkage information is used by the compiler to locate important modules such as program counter and instruction memory. The code segment which is shown in Fig. 3.3 specifies the program counter and instruction memory locations.

Figure 3.2 MIMOLA description connecting two modules. Source: [80].

```
CONNECTIONS ALU.result -> ACC.input
            ACC.output -> ALU.a
```

Figure 3.3 MIMOLA linkage segment specifies the program counter and instruction memory locations. Source: [80].

```
LOCATION_FOR_PROGRAM_COUNTER PC;
LOCATION_FOR_INSTRUCTIONS INSTR_MEMORY[0..1023];
```

From the verification point of view, the MSS tools rely solely on functional verification techniques based on simulation which are more deeply described in Chapter 5.

3.2 Instruction-Set-Oriented ADLs

The problem of the structure-oriented ADLS with the extraction of the instruction set can be avoided by abstracting behavioral information away from the structural details. Instruction-set-oriented (sometimes also named *behavioral*) ADLs explicitly specify the instruction semantics and ignore detailed hardware structures. This typically leads to a situation when there is a correspondence between instruction-set-oriented ADLs and the instruction set reference manual.

Typically, the instruction-set-oriented languages describe the microprocessor's instruction set in a hierarchical way using, for instance, attribute grammars [106]. This property simplifies the instruction set description by sharing the common components between operations. However, the capabilities of these models are limited due to the lack of detailed pipeline and timing information. Thus, it is not possible to generate cycle-accurate simulators without certain assumptions regarding control behavior. Due to the lack of structural details in instruction-set-oriented ADLS, it is also not possible to perform any resource-based scheduling [92, 67]. Furthermore, without the ability to capture the low-level information, it is also very difficult to deploy verification techniques that are based on a gradual refinement of microprocessor description.

In this section, we will describe two instruction-set-oriented ADLs: ISDL [54] and TIE [117].

3.2.1 ISDL

The *Instruction Set Description Language* (ISDL) [54] was designed to be an ADL for compiler retargetability, specially focused on microprocessors with *very large instruction words* (VLIWs). ISDL is a purely instruction-set-oriented language based on an attributed grammar which is primarily used to describe the instruction set of processor architectures. Thus, without additional assumptions, the ISDL tools (such as GenSim simulator generator) are not capable of extracting the correct behavior for pipelined architectures with complex execution schemes that include, for instance, cancellation of partially executed instructions (pipeline clearing), or multi-cycle instructions of variable length [53].

Figure 3.4 Example of an instruction set description in ISDL.

```
Section Format
Main = OPCODE[8];

Section Global_Definitions
//    Assembly Token Type Value
Token X[0..1]   XR     ival { yylval.ival = yytext[1] - ’0’; };
Token Y[0..1]   YR     ival { yylval.ival = yytext[1] - ’0’; };
Token ACC       AR     ival { };

//            Type Assembly Action
Non_Terminal ival XYSRC:    XR { $$ = 2 * XR; } |
                            YR { $$ = 2 * YR + 1; };
Non_Terminal ival ACC:      AR { $$ = 1; }

Section Storage
Register X0   = 0x8; Register X1   = 0x8; Register Y0   = 0x8;
Register Y1   = 0x8; Register ACC = 0x8

Section Assembly
  Field Main:
    // Assembly    // Binary
    ADD XYSRC, ACC { Main.OPCODE = 0x01 | (ACC<<3) | (XYSRC<<4); }
                   { ACC <- ACC + XYSRC; }  // RTL Operation
                   { cycle = 2; size = 1; } // Costs
                   { latency = 1; }         // Timing

Section Constraints
~(REP *) & ([1] ADD *, *)
```

ISDL description of the microprocessor consists of mainly five sections: (i) instruction word format, (ii) global definitions, (iii) storage resources, (iv) assembly syntax and constraints, and (v) an optimization information section.

The instruction word is separated into multiple fields each containing one or more sub-fields. The bitwidth of each sub-field is also provided. The instruction word is assembled by concatenating all the sub-fields in the order specified in this section. Fig. 3.4 shows an example of the format section for a simple instruction with just one field `Main` with a single sub-field `OPCODE`. The total length of the instruction word in the example is 8 bits.

Next, Fig. 3.4 also demonstrates ISDL's global definition section. Here, primitive and complex operands of the microprocessor's assembly language are defined. Each operand definition consists of the keyword `Token`, the syntax of the token as it appears in assembly, a symbolic name for the token, the type of value returned by the token, and a piece of Lex [20] dependent code that returns the appropriate token value [54]. For instance, in Fig. 3.4, the first token has a symbolic name `XR` whose value is an integer. The assembly syntax allowed is either `X0`, or `X1`, and the values returned are 0 or 1 respectively.

The complex operands are then defined via non-terminals which have several purposes. First, syntactically unrelated tokens can be grouped together into a non-terminal for convenience. For instance, if there is a large number of possible alternatives in an instruction (e.g., several addressing modes), they can be factored out to a non-terminal. Next, non-terminals can also define new grammar rules, not necessarily related to any instruction. Finally, the action portion of non-terminals allows the inclusion of arbitrary C code to be executed along with every rule. The non-terminal definitions consist of the keyword `Non_Terminal`, the type of the returned value, a symbolic name as it appears in the assembly, and an action that describes the possible token or non-terminal combinations and the return value associated with each. For example, in Fig. 3.4, the non-terminal with the symbolic name `XYSRC` returns value 1 and 3 for registers `Y0` and `Y1`, respectively.

The storage section lists all storage resources visible to the programmer. It lists the names and sizes of the memory, the register files, and the special registers. This section is used by the compiler to determine the available resources and how they should be used.

The assembly syntax section is then split into subsections (per each field defined in format section) corresponding to the separate operations that can be performed in parallel within a single instruction. An instance of the assembly section shown in Fig. 3.4 as well. One can see, that each operation consists of assembly mnemonic, a binary representation of instruction, the effect of the operation on storages, operations costs (such as execution time and code size), and timing information (e.g., because of pipelining).

The assembly syntax section describes a number of fields that can be generally executed in parallel. However, there are certain combinations of operations that may not be executable by the hardware. The constraints section is used to make these combinations visible to the compiler so that the compiler can avoid generating such illegal operation combinations. The constraints are described as a set of Boolean rules, all of which must be satisfied for an instruction to be valid. Fig. 3.4 contains an example that shows how to describe the constraint that the instruction `ADD` cannot directly follow instruction `REP`. The `[1]` indicates a time shift of one instruction fetch for the `REP` instruction. The "`~`" is a symbol for NOT and "`&`" is for logical AND.

3.2.2 TIE

The *Tensilica Instruction Extension* (TIE) [117] is an ADL language aimed at customization of the functionality of RISC Xtensa processors [30] within Tensilica Software Development Toolkit (SDK) [28]. The customization is given by defining custom execution units, register files, I/O interfaces, load/store instructions, and multi-issue instructions which are synthesized into configurable hardware components. The TIE language syntax is a mixture of the Verilog hardware description and the C programming language. A designer does need to worry about pipelining, control/bypass logic, and interfacing to other processor modules as the instruction extensions are integrated directly into the processor pipeline by the SDK. In other words, the TIE language is used only for adding instruction extensions and datapaths to a processor pipeline as it is not a general-purpose hardware design language.

The TIE language optimizes computational strength of the processor in the following ways. One can (i) create new instructions to increase processor performance and efficiency. This is achieved by defining the exact data width needed for the application instead of using an implicit standard size transfer bandwidth, or by merging serial operations into a single instruction that can be issued back-to-back to achieve single cycle throughput. Further, one may also (ii) utilize data-level parallelism by creating *single instruction mul-*

Figure 3.5 Definition of the TIE instruction `addshift`. Source: [29].

```
operation addshift {out AR avg, in AR A, in AR B} {}
{
    assign avg = (A + B) >> 1;
}
```

Figure 3.6 Optimization of computation using a custom instruction. Source: [29].

```
// For-loop in the C language
for (unsigned int i = 0; i < N; i++) {
    c[i] = (a[i] + b[i]) / 2; // <<< target to optimize
}

// Compiled assembly:
// * without optimizations                * with optimization
...                                       ...
add.n a9,a11,a10                          addshift a12,a10,a8
srli  a9,a9,1                             ...
...
```

tiple data (SIMD) operations, and perform the same operation across multiple elements. Next, (iii) instruction-level parallelism can be used as well by creating multi-operation VLIW instructions with variable slot widths. Finally, (iv) data bandwidth connecting RTL blocks, memories, or other processors can be increased without going through the system bus, reducing I/O bottlenecks and improving data throughput.

In order to demonstrate the use of the TIE language, assume the code shown in Fig. 3.5 that describes a new instruction named `addshift`. Fig. 3.6 then shows the practical use of the first of the above-mentioned optimization approaches, that is, the merge of several instructions increases performance by combining multiple operations into a single instruction. The C code in Fig. 3.6 contains a for-loop with an inner-loop code of `c[i] = (a[i] + b[i]) / 2`. Compiling this code on the Xtensa processor without any custom instructions results in two sequentially executed instructions. The first operation, `add.n`, calculates the two's complement 32-bit sum. The second operation, `srli`, shifts the contents by a constant amount encoded in the instruction word right (inserting zeros on the left). Each iteration of the for-loop executes in two cycles. With the TIE language, we can merge the two operations into a single new operation called `addshift` that performs both the add and shift operations at the same time. Now, compiling the for-loop using the new fused operation, the assembly code shows the fusion operation `addshift` that executes only in a single cycle. The TIE compiler automatically generates an opcode for the `addshift` operation, and all software tools are automatically updated to understand the function and timing of the newly added `addshift` operation.

As we have seen, the TIE language allows the designer to perform limited microprocessor customizations by utilizing configurable hardware components. In such a way, for instance, a new instruction with special semantics can be defined. However, the TIE language is not capable to perform any general structural changes as the processor architecture is implicitly bound to the one used in Xtensa processor families.

Figure 3.7 Example of instruction description in the nML language.

```
op alu_instruction(operation:alu_operation, src:src_t, dst:dst_t)
{
    action {
        op_src = src;
        op_dst = dst;
        operation.action;
        dst = op_dst;
    }
    syntax : operation" "dst","src;
    image  : operation::dst::src;
}
op alu_operation = add | sub;
op add()
{
    action {
        op_dst = op_dst + op_src;
    }
    syntax : "add";
    image  : 0xA;
}
```

3.3 Mixed ADLs

The mixed ADLs capture both, the structure and behavior of the architecture. This section further focuses on two examples of the mixed ADLs: nML and CodAL.

3.3.1 nML

The *nML* language [46, 79, 99, 109] is a high-level definition language originally designed for instruction set descriptions. Thus, it offers the abstraction level comparable to the programmer's manual of a given processor. The main idea behind the design of the nML language builds on the fact that several instructions may share common properties. Ideal nML descriptions are compact and simple if the shared properties are properly re-used. A hierarchical scheme is used to describe instruction sets. The instructions are the topmost elements in the hierarchy. The intermediate elements of the hierarchy are the so-called *partial instructions* (PIs). The relationship between elements can be established using `AND` and `OR` composition rules. The `AND`-rule groups several PIs into a larger PI while the `OR`-rule enumerates a set of alternatives for one PI. Therefore, instruction definitions in nML can be in the form of an and/or tree where each possible derivation of the tree corresponds to an actual instruction. In Fig. 3.7, the definition of `alu_instruction` joins three PIs with the `AND`-rule: `alu_operation`, `src_t`, and `dst_t`. The first PI, `alu_operation`, uses the `OR`-rule to describe the valid options for ALU actions, that is, `add` or `sub`. The number of all possible derivations of `alu_instruction` is given by the product of the size of `alu_operation`, `src_t`, and `dst_t`. The shared behavior of all these options is defined in the `action` attribute of `alu_instruction`. Each option for `alu_operation` should have its `action` attribute defined as its specific behavior, which is referred by the `operation.action`

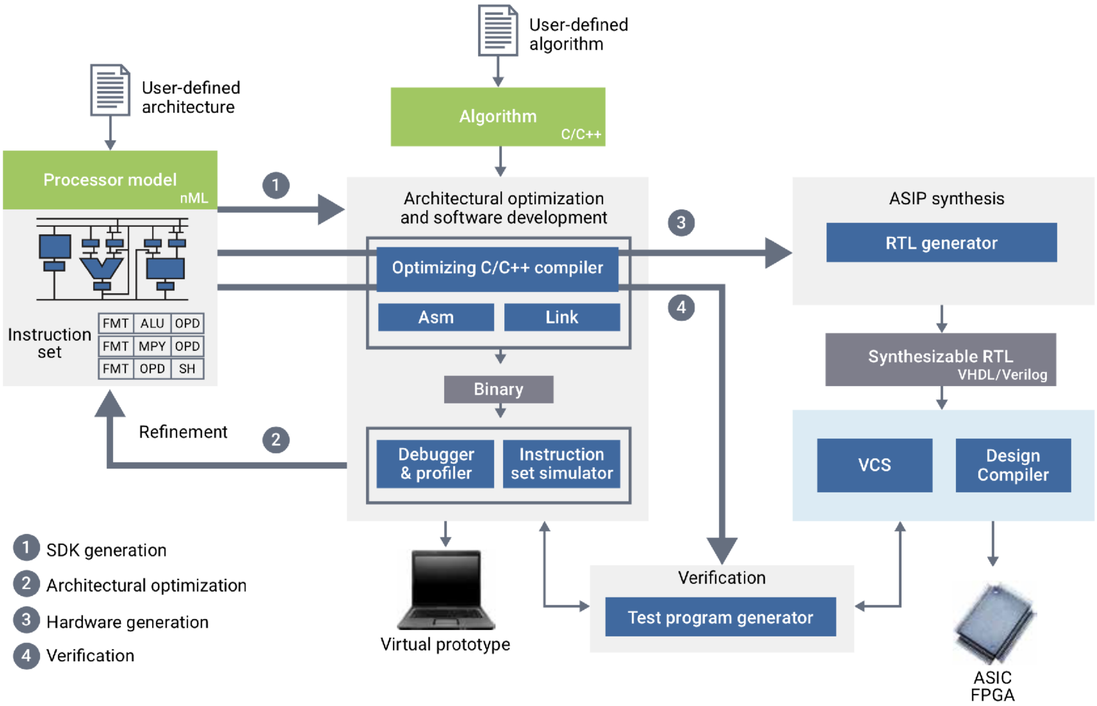


Figure 3.8: An overview of an ASIP Designer tool flow. Source: [125].

command. In the example, the action description is given for `add` operation. Binary and assembly syntax can also be specified in the same hierarchical manner using `image` and `syntax` attributes.

The nML language is also capable of capturing the structural information of the microprocessor. The nML language supports three types of storages: (i) RAM, (ii) register, and (iii) transitory storage. While the RAM and register storages are visible directly to the instruction set, the transitory storage refers to machine states that are retained only for the limited number of cycles, for instance, values on buses and latches. Computations have no delay in the nML timing model — only storage units have one. Instruction delay slots are modeled by introducing storage units as pipeline registers. The results of the computation are then propagated through the registers according to the description in the behavioral specification.

The nML models constraints between operations by enumerating all the valid instruction combinations, and thus such an enumeration may render nML descriptions which are very long. More complicated constraints, which often appear in DSPs associated with irregular instruction-level parallelism, or in VLIW processors with multiple issue slots, are hard to model with nML. For example, nML cannot model the constraint that instruction I_1 cannot directly follow instruction I_0 [92, 58].

The nML language has been used by several HDL code generators such as CBC [46], Structural Sim-HS [11], and Chess [79]. An example of the instruction set simulators that build on the nML language are then Sigh/Sim [47], Behavioral Sim-HS [11], and Checkers [51, 125]. The Behavioral and Structural Sim-HS are together provided within *Sim-HS* framework allowing the transformation of microprocessor description to the corresponding Verilog models that are suitable for simulation and synthesis, respectively. However, be-

Figure 3.9 Example of resource description in the CodAL language.

```
// Program Counter
program_counter bit[8] pc;
// General Purpose 4b Registers - r0..r3
arch register bit[8] regs[4] { .dataport = {2, 1} };
// Program Memory
memory bit[8] prog_mem {
    .dataport = {1, 0}, .lau = 8, .endianess = big,
    .size = 256, .flags = {r, x}, .latency = {0, 1}
};
// Memory Mapping for Program Memory
memorymapping defaultmap { 0..255 = prog_mem[7..0]; };
// Wires
signal bit[1] jmp_en;
signal bit[2] opcode, srcA, srcB, dst, alu_op;
signal bit[4] imm, addr;
signal bit[8] mem, regA, regB;
```

cause of the lack of low-level information, a more optimized (e.g., deeply pipelined) synthesizable output cannot be generated. The problem with the lack of the low-level information in the nML language was addressed by several language vendor-specific extensions allowing more precise modeling of pipelines and VLIW instructions. These extensions push the nML language more towards the group of mixed ADLs.

One of the significantly extended variants of the nML language is now adopted by *ASIP Designer* [125] which builds on an updated version of the previously mentioned Chess/Checkers [79, 51] environments. The nML version used in ASIP Designer provides options to expose the exact processor's resource and pipeline utilization. This accurate structural and timing description stands at the basis of the simulation and hardware generation techniques used in the ASIP Designer tool suite (schematically visualized in Fig. 3.8). The nML hazard rules provide efficient solutions for pipeline conflicts, either by stalling or forwarding, and their compact notation gives the designer full control over handling of the pipeline hazards. The generated pipeline control logic that avoids the hazards is supposed to be correct *by construction*. The designer can then tweak the hardware-software trade-off while being relieved from the detailed hardware implementation of pipeline interlocking and forwarding paths. The ASIP Designer can also co-operate with functional verification tools, for instance, VCS [126], static analyzers, e.g., SpyGlass Lint [127], and formal verifiers such as VC Formal [128].

3.3.2 CodAL

CodAL is a language used by Codasip [1] which is an environment aiming at rapid processor development. In Codasip, each processor is described by two CodAL models, the instruction-accurate (IA) model, and the cycle-accurate (CA) model. The IA model describes the syntax and semantics of the instructions and their functional behavior without any micro-architectural details. On the other hand, the CA model then describes micro-architectural details such as pipelines, decoding, timing, etc.

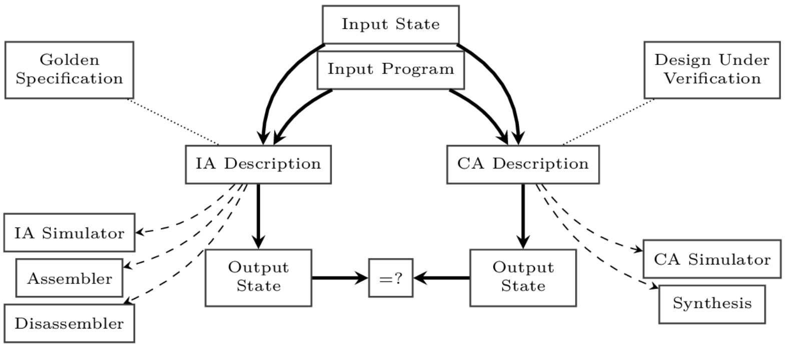


Figure 3.11: Processor design flow in Codasip.

The CodAL descriptions are composed of two main types of definitions: (i) resources, (ii) instructions and events. The resource description captures hardware elements of a given processor. These may involve the definition of registers, memories, and system buses together with their mappings. Further, the resource description can also include other elements such as signals (wires) or pipelines. Fig. 3.9 demonstrates resources commonly present in almost every processor. The example contains a program counter `pc` (8bit), a memory resource `prog_mem` to store the program code (256 x 8bit), and default mapping of the program memory to the processor address space. Further, it also contains a definition of architecturally visible register file `regs` (4 x 8bit) and definition of globally accessible signals (i.e., equivalents of wires in HDLs).

Next, Fig. 3.10 shows an example of an instruction and event description at the IA level. This part contains a definition of the instruction set `instr_set` accompanied by the description of operand `r` which represent access to the previously defined `regs` resource. Similarly, as in the case of nML, the binary and assembly syntax, as well as semantic actions, can also be specified in a hierarchical manner using `assembler`, `binary`, `semantics` sections. The example also includes special events that must be involved in each processor description — namely, (i) the `reset` event that describes start up state of the processor, (ii) the `halt` event describing shutdown actions of the processor, and (iii) the `main` event describing an assembly grammar entry point and actions that the processor should do at every clock cycle. The main event defines the top-level element of the processor's instruction set in `start` section. Moreover, the `decoders` section holds information about decoding instructions using the instruction decoder. Finally, the `semantics` section describes computation done in each clock cycle. As can be seen in this IA example, the whole processing of each instruction is done in just one cycle. However, for CA models, the main event would typically contain activations of pipeline stages, interrupt checking, etc.

From the IA and CA CodAL models, Codasip tools can automatically generate SDK tools (assembler, disassembler, linker, C-compiler, simulators, profilers, debuggers) [64, 130, 111, 110, 112]. Fig. 3.11 depicts the common processor design flow in Codasip. Typically, the IA model is available significantly sooner than the CA one. This model allows the compiler tool-chain and the IA simulator to be generated. These tools then give programmers an opportunity to start early with writing programs for the instruction set given by the IA model. When the development of the CA model is finished and ready for hardware synthesis, the IA model can serve as the so-called *golden specification* for processor verifica-

Figure 3.10 Example of instruction-accurate description of instructions and events in the CodAL architecture description language.

```
element r represents regs {
    assembler   { "r" ~ idx=unsigned };
    binary      { idx=0b[2] };
    return      { idx; };
}

element instr_add {
    use r as dst, srcA, srcB;
    assembler   { "ADD" dst "," srcA "," srcB };
    binary      { 0x1:2 dst     srcA     srcB };
    semantics   {
        alu_op = ALU_ADD;
        jmp_en = FALSE;
    };
}

set instr_set = instr_add /* ... */;

event main {
    use instructions;
    start           { { instr_set;         } };
    decoders (pc)   { { instr_set(opcode); } };
    semantics       {
        // Fetch instruction
        mem       = prog_mem[pc];
        // Split instruction into opcode and operand parts
        opcode    = (mem >> 6) & 0x3;
        srcA      = (mem >> 4) & 0x3;
        srcB      = (mem >> 2) & 0x3;
        dst       = (mem)      & 0x3;
        // ...
        // Get data from registers
        regA   = regs[srcA];
        regB   = regs[srcB];
        // Perform write-back
        switch (alu_op) {
            case ALU_ADD: regs[dst] = regA + regB; break;
            // ...
        }
    };
}

event reset { semantics { pc = 0x00; }; }

event halt { }
```

tion. As it is discussed in [142], to assure mutual equivalency between IA and CA models, Codasip uses an UVM-based functional verification.

Chapter 4

Introduction to Selected Areas of Formal Verification

In this chapter, we will introduce basic notions of formal verification methods and concepts used later in the thesis. We recall that verification is a process that checks whether a system is correct with respect to a provided specification. As opposed to testing and bug-hunting techniques which aim at detection of flaws against the specification, the goal of formal verification is to formally (mathematically) prove that the system is indeed correct. That is, if no issue is found by a formal method, it is guaranteed that the system conforms to the given specification. Ideally, a formal approach should be sound and complete which means that an error is reported if and only if there is a real error in a system, otherwise the system is said to be correct. However, meeting these properties can be costly (or impossible to achieve) and, therefore, to provide efficiency and automation, completeness and/or soundness are sometimes compromised which leads to error detection methods with formal roots.

In the rest of this chapter, we will formally introduce concepts that will be used throughout the thesis, in particular, model checking, static analysis, and SAT/SMT solvers.

4.1 Preliminaries

Throughout this thesis, the standard notion of *formal languages* is used according to their definition as it is given in [77, 60, 89].

Definition 1. An *alphabet* is defined as a non-empty finite set of *symbols*.

Definition 2. A *word* over an alphabet Σ is recursively defined as follows:

- the empty word ε is a word over the alphabet Σ,
- if x is a word over Σ and $a \in \Sigma$, then xa is also a word over Σ.

We denote the set of all words over an alphabet Σ as Σ^*. By *concatenation* one can always combine two words x, y over Σ to form a new word xy.

Definition 3. A *formal language* L is defined as any subset of Σ^*. Next, given formal languages L_1 and L_2 over Σ, we define *concatenation* $L_1 \cdot L_2$ of formal languages as the set $\{xy \mid x \in L_1 \wedge y \in L_2\}$. Moreover, given a formal language L, we define the *iteration* L^*, resp. the *positive iteration* L^+, of the language L as follows:

- $L^0 := \{\varepsilon\}$,
- $L^n := L \cdot L^{n-1}$ for $n \geq 1$,
- $L^* := \bigcup\limits_{n \geq 0} L^n$,
- $L^+ := \bigcup\limits_{n \geq 1} L^n$.

Further, let us define a significant class of the formal languages known for its many practical applications.

Definition 4. A *regular set* over an alphabet Σ is recursively defined as follows:

- $\emptyset$ is a regular set over Σ,
- $\{\varepsilon\}$ is a regular set over Σ,
- for all $a \in \Sigma$, $\{a\}$ is a regular set over Σ,
- if P and Q are regular sets over Σ, then $P \cup Q$, $P \cdot Q$, P^* are also regular sets over Σ.

The class of regular sets is thus the smallest language class that contains $\emptyset$, $\{\varepsilon\}$, $\{a\}$ for all symbols $a \in \Sigma$, and it is closed with respect to union, concatenation, and iteration.

4.2 Model Checking

Model checking [9] is an algorithmic approach of checking whether a given system satisfies a given property through a systematic exploration of the state space of the system. Compared to other formal approaches (such as static analysis or theorem proving), model checkers are (usually) highly automated (for a closed system), fairly general, and capable of providing counter-examples. Often, a CEGAR loop [39] is supported allowing for automated refinement of the used abstraction in order to exclude spurious counter-examples. One of the major disadvantages of model checkers is the so-called space-explosion problem which needs to be typically mitigated by efficient storage techniques (such as BDDs [26, 22]), automata (as in [17, 16]), state-space reductions (for example, the so-called partial order reduction [40]), or (more recently) the integration of SAT solvers in model checking engines as in, for instance, IC3/PDR [18, 45]. Another significant disadvantage is that a closed system is required, i.e., the verified system must be joined with a model of its environment which may require a lot of non-trivial labour.

The following sections briefly describe relevant model checking concepts that are later used in this thesis.

4.2.1 Transition Systems

In this section, a notion of *transition systems* is defined in the same way as it described in [9]. A transition system is a mathematical structure consisting of two parts, (i) a set of configurations and (ii) a binary relation on this set.

Definition 5. A *transition system* T is a pair of the form $T = (C, \hookrightarrow)$ where C is a set of *configurations* and $\hookrightarrow \subseteq C \times C$ is a binary *transition relation*.

The purpose of transition systems is to describe behaviors which we define as certain sequences of configurations.

Definition 6. A transition system $T = (C, \hookrightarrow)$ generates a set $S(T)$ of *sequences* defined as follows: (i) the finite sequence $c_0, \ldots, c_n$ (for $n \geq 0$) belongs to $S(T)$ if $c_0 \in C$ and $c_i \hookrightarrow c_{i+1}$ for all $0 \leq i < n$, (ii) the infinite sequence $c_0, \ldots, c_n, \ldots$ belongs to $S(T)$ if $c_0 \in C$ and $c_i \hookrightarrow c_{i+1}$ for all $0 \leq i$.

In most applications of transition systems, we are only interested in configurations of the transition system that are reachable from given initial configurations.

Definition 7. Given a transition system $T = (C, \hookrightarrow)$ and a set $I \subseteq C$ of *initial configurations*, we say that a configuration $c_n \in C$, $n \geq 0$, is *reachable* if there exists a sequence $c_0, \ldots, c_n \in S(T)$ such that $c_0 \in I$.

4.2.2 Parameterized Systems

In this thesis, we will work with a common notion (used, e.g., in [41, 102, 3]) of a *parameterized system* operating on a linear topology where processes may perform local transitions or universally/existentially guarded transitions.

Definition 8. A parameterized system is a pair $P = (Q, \Delta)$ where Q is a finite set of local states of a process and Δ is a set of transition rules over Q. A transition rule is either local or global. A *local transition rule* is of the form $q \to q' \in \Delta$, $q, q' \in Q$. A *global transition rule* is then of the form $\mathbb{Q}_\circ \colon G \models q \to q' \in \Delta$ where $\mathbb{Q} \in \{\forall, \exists\}$, $\circ \in \{\leftarrow, \rightarrow, \leftrightarrow\}$, $G \subseteq Q$, and $q, q' \in Q$ with a part "$\mathbb{Q}_\circ \colon G$" being referred as *transition guard.* The global rule can be applied only if its transition guard is satisfied. For example, the meaning of the guard $\exists_\leftrightarrow \colon G$ is "for each state $g \in Q$ from the set G, there should be at least one process in the linear topology including the current one so that the process is in the state g". Formally, the guard $\exists_\leftrightarrow \colon G$ is satisfied in the configuration $q_1 \ldots q_i \ldots q_n$ by the i-th process iff $\forall q \in G\ \exists 1 \leq j \leq n \colon q_j = q$. Similarly, the meaning of the guard $\exists_\leftarrow \colon G$ is "for each state q from the set G, there should be at least one process to the left of the current one so that the process is in the state q". Formally, the guard $\exists_\leftarrow \colon G$ is satisfied in the configuration $q_1 \ldots q_i \ldots q_n$ by the i-th process iff $\forall q \in G\ \exists 1 \leq j < i \colon q_j = q$. The meaning of the other guards is defined analogically.

A parameterized system $P = (Q, \Delta)$ induces an infinite transition system $T = (C, \hookrightarrow)$ whose configurations C are finite non-empty words over Q, i.e., elements from the set Q^+. If we use $c[i]$ to denote the state of the ith process within the configuration $c \in C$, the transition relation $\hookrightarrow$ then contains a transition $c \hookrightarrow c'$ with $c[i] = s$, $c'[i] = s'$, $c[j] = c'[j]$ for all $j \colon j \neq i$ iff either (i) Δ contains a local rule $s \to s'$, or (ii) Δ contains a global rule $\mathbb{Q}_\circ \colon G \models s \to s'$, and one of the following conditions is satisfied:

- $\mathbb{Q} = \exists \wedge \circ = \leftrightarrow$ and $\forall q \in G \colon\ \exists 1 \leq j \leq |c| \colon c[j] = q$,
- $\mathbb{Q} = \exists \wedge \circ = \leftarrow$ and $\forall q \in G \colon\ \exists 1 \leq j < i \colon c[j] = q$,
- $\mathbb{Q} = \exists \wedge \circ = \rightarrow$ and $\forall q \in G \colon\ \exists i < j \leq |c| \colon c[j] = q$,
- $\mathbb{Q} = \forall \wedge \circ = \leftrightarrow$ and $\forall 1 \leq j \leq |c| \colon c[j] \in G$,
- $\mathbb{Q} = \forall \wedge \circ = \leftarrow$ and $\forall 1 \leq j < i \colon c[j] \in G$,

- $\mathbb{Q} = \forall \wedge \circ = \dashrightarrow$ and $\forall i < j \leq |c| \colon c[j] \in G$.

An instance of the reachability problem is defined by a parameterized system $P = (Q, \Delta)$, a regular set $I \subseteq Q^+$ of initial configurations, and a set $\mathit{Bad} \subseteq Q^+$ of bad configurations. In particular, we will define Bad as the upward closure of a finite set $B \subseteq Q^+$ of minimal bad configurations. This is, $\mathit{Bad} = \{c \in Q^+ \mid \exists b \in B \colon b \sqsubseteq c\}$ 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 of the transition system $T = (C, \hookrightarrow)$. We say that the system P is safe wrt I and Bad iff no bad configuration is reachable, i.e., $R \cap \mathit{Bad} = \emptyset$.

4.2.3 Regular Model Checking

Regular model checking (RMC), firstly described in [71] and [141], is a uniform framework for analyzing various classes of parameterized and infinite-state systems. The regular model checking framework [4] represents a transition system as follows:

- A configuration (state) of the system is a word over an alphabet Σ.
- The set of initial configurations is a regular set over Σ.
- The transition relation is a regularity-preserving relation[1] on Σ, often (but not always) required to be regular and length-preserving. It is typically represented by a finite-state transducer over $(\Sigma \times \Sigma)$, which accepts all words $(a_1, b_1) \cdots (a_n, b_n)$ such that $(a_1 \cdots a_n, b_1 \cdots b_n)$ is in the transition relation.[2]

More formally, a length-preserving finite-state transducer T over Σ is a tuple (Q, s, δ, F) where Q is the set of states, $s \in Q$ is the initial state, $\delta \colon (Q \times \Sigma \times \Sigma \times Q$ is the transition function, and $F \subseteq Q$ is the set of accepting states. A transducer configuration is a pair (q, w) where $q \in Q$, $w \in (\Sigma \times \Sigma)^*$. Given transducer configurations (q_1, aw) and (q_2, w), $a \in \Sigma \times \Sigma$, we say that the transducer makes a transition from (q_1, aw) to (q_2, w) denoted $(q_1, aw) \vdash (q_2, w)$, iff $q_2 \in \delta(q_1, a)$. The language of T is the language $\{w \in (\Sigma \times \Sigma)^* \mid (s, w) \vdash^* (f, \varepsilon) \wedge f \in F\}$ where $\vdash^*$ is the transitive closure of the relation $\vdash$ defined in the standard way. We use $L(T)$ to denote the language of T. The transducer T induces a regular relation R on words over Σ. More precisely, for words $x = a_1 \cdots a_n$ and $y = b_1 \cdots b_n \in \Sigma^*$, we have $(x, y) \in R$ if $(a_1, b_1) \cdots (a_n, b_n) \in L(T)$. The idea is that R is used to represent the transition relation on the configurations of the system (each of which is a word in Σ).

When using RMC, a safety verification task is formulated as follows: Given a regular set I of initial configurations, a regularity-preserving relation $R \subseteq \Sigma^* \times \Sigma^*$, and a regular set of bad configurations $B \subseteq \Sigma^*$, is it the case that $R^*(I) \cap B = \emptyset$? Due to the undecidability issues, the question may not be solvable in general. It is solvable for length-preserving systems, but even there one may hit a problem in the form of state explosion. Moreover, note that even in length-preservation case, $R^*(I)$ cannot be computed by simple iterative computation of $R^n(I)$ where $n \geq 0$. Therefore, an accelerated computation of $R^*(I)$ is required. Here, an application of abstraction on the involved automata, leading to abstract RMC [17] (ARMC), has shown as particularly successful.

[1] A relation $\varrho \subseteq \Sigma^* \times \Sigma^*$ is regularity preserving iff $\varrho(L) \in \mathcal{L}_3$ for every $L \in \mathcal{L}_3$.

[2] Sometimes, the transition relation is given as a union of a finite number of relations, each of which is called an action.

As it is shown, for instance, in [4], one of typical applications of RMC is verification of parameterized systems with linear or ring-formed topologies (where each component is finite-state). Within this thesis, namely in Chapter 9, ARMC method of [17] is used for showing that potential pipeline hazards may indeed occur in certain interleavings of instructions.

4.3 Static Analysis

Static analysis tries to avoid direct execution of the system being examined and, instead, it analyses and gathers approximate (often conservative) information about the system from its source code. Therefore, it may produce many false alarms. From the point of view of this thesis, the most important form of static analysis is the so-called data-flow analysis that is described in the next section.

4.3.1 Data-Flow Analysis

Data-flow analysis (DFA) is a technique for gathering information about the possible set of values calculated at various points in a computer program or circuit. The information gathered is often used by compilers when optimizing the given program or circuit. An example of a DFA is the computation of reaching definitions in compilers.

As an input, the DFA typically expects a *flow graph* G describing a given program (then G typically has the form of the so-called *control flow graph*) or a circuit (where G can have the form of a block schema). The flow graph can be often represented by a tuple (B, E, L) where B is a finite set of blocks, $E \subseteq B \times B$ is a finite set of oriented edges, and L is a labeling function[3]. A simple way to perform DFA is to deploy the so-called *monotonic DFA framework* which, for each block of the flow graph, sets up data-flow equations over data-flow domains having the form of a complete lattice. The equations are then solved by repetitive local calculation of output from inputs at each node until the whole system stabilizes, i.e., it reaches a fixpoint. This general DFA framework-based approach was firstly introduced in [73].

Given a flow graph (B, E, L), an instance of the DFA framework can be more formally described using a quintuple $(V, \sqcap, F, b_0, v_0)$ where V is a set describing possible flow values, $\sqcap\colon V \times V \to V$ is a meet operator (describing how are values originating from multiple locations joined together), F is a set of block monotone transfer functions $f_b\colon V \to V$ for each block $b \in B$ (describing the effect of passing through a block), $b_0 \in B$ is a boundary block, and $v_0 \in V$ is a boundary value. Moreover, it is expected that a pair $(V, \sqcap)$ forms a complete lattice, and thus V contains the bottom element $\bot$ and the top element $\top$. Finally, F must include the identity function, it must be closed under the function composition, and the used lattice should not contain infinite descending chains.

The most common (naive) solution for solving a DFA framework instance is given in Alg. 1. Here, the output states $Out(b)$ for each block b are computed by applying the transfer functions on the input states $In(b)$. From these, the input states are updated by applying the meet operation. The latter two steps are repeated until we reach the fixpoint, that is, the situation in which the output states do not change anymore. After reaching the fixpoint, the output and input states of the blocks can be used to derive properties of the program or circuit at the block boundaries.

[3] A concrete form of the labeling function usually depends on the purpose of data-flow analysis.

Algorithm 1 Round-robin iterative DFA algorithm.

```
$Out(b_0) := v_0$
for $b \in B \setminus \{b_0\}$ do
    $Out(b) := \top$
end for
while $Out(b)$ has changed for some $b \in B$ do
    for $b \in B$ do
        $Pred := \{b' \in B \mid (b', b) \in E\}$
        $In(b) := \sqcap_{p \in Pred} Out(p)$
        $Out(b) := f_b(In(b))$
    end for
end while
```

In Chapter 9 of the thesis, data-flow analysis of this type is utilized to detect potential hazards in a microprocessor's pipeline control logic.

4.4 SAT and SMT Solvers

The Boolean satisfiability (SAT) problem is the problem of determining whether there exists an interpretation that satisfies a given Boolean formula. In other words, we ask whether the variables of a given Boolean formula can be consistently replaced by the values *True* or *False* in such a way that the formula evaluates to *True*. Similarly, the satisfiability modulo theories (SMT) problem is a decision problem for first-order logical formulas with respect to combinations of background theories expressed in the classical first-order logic with equality. Examples of such theories are the theory of real numbers, the theory of integers, and the theories of various data structures such as lists, arrays, bit vectors, etc. SAT and SMT solving has found many applications in verification (e.g., within predicate abstraction or invariant checking), test generation, hardware synthesis, error trace minimization, and artificial intelligence [18, 45].

In this thesis, SAT and SMT solvers are utilized in several cases. For instance, in Chapters 7 and 8, the GlueMinisat [101] SAT solver is used as an external SAT solver for the Cadence SMV [87] tool. In another case (in Chapter 9), the Z3 [100] SMT solver is utilized for validation of the consistency of a processor pipeline.

4.4.1 SAT Solvers

The SAT problem, which asks whether a given propositional formula is satisfiable, is the first problem which has been proven to be NP-complete. Normally, we consider a propositional formula to be given in the conjunctive normal form (CNF), i.e., as a conjunction of clauses where a clause is a disjunction of literals, and a literal is a (possibly negated) propositional symbol. Stated formally, let P be a finite set of propositional symbols. If $p \in P$, then p is an atom, and p and $\neg p$ are literals of P. A clause is a disjunction of literals $\ell_1 \vee \ldots \vee \ell_n$. A CNF formula is a conjunction of one or more clauses $C_1 \wedge \ldots \wedge C_n$. Most contemporary SAT-solvers build on variants of the classical Davis-Putnam-Longemann-Loveland (DPLL) procedure [44] extended to the so-called conflict-driven clause-learning approach (CDCL) [85, 12], which we will describe in terms of an abstract CDCL system.

Abstract CDCL Algorithm

An abstract CDCL system is a pair $(S, \to)$ where S is a set of states of the system and $\to \subseteq S \times S$ is its set of transitions modeling progress of the algorithm. Most states are of the form $M \parallel F$ where:

- M is a sequence of annotated literals denoting a partial truth assignment, and
- F is the CNF formula being checked, represented as a set of clauses.

The initial state is $\emptyset \parallel F$, where F is to be checked for satisfiability. The final state is either:

- the special fail state *fail* if F is unsatisfiable, or
- $M \parallel G$ where G is a CNF formula equisatisfiable with the original formula F and M satisfies G.

We further write $F \models C$ to mean that, for every truth assignment v, $v(F) = \mathit{True}$ (i.e., F holds in valuation v) implies $v(C) = \mathit{True}$.

In what follows, we will describe transitions between states of the abstract system CDCL system $(S, \to)$.

- *Pure Literal*

$$M \parallel F \to M\ell \parallel F \quad \textbf{if} \quad \begin{cases} \ell \text{ occurs in some clause of } F, \\ \neg\ell \text{ occurs in no clause of } F, \text{ and} \\ \ell \text{ is undefined in } M. \end{cases}$$

- *Decide*

$$M \parallel F \to M\ell^d \parallel F \quad \textbf{if} \quad \begin{cases} \ell \text{ or } \neg\ell \text{ occurs in a clause of } F, \text{ and} \\ \ell \text{ is undefined in } M. \end{cases}$$

- *Unit Propagate*

$$M \parallel F, C \vee \ell \to M\ell \parallel F, C \vee \ell \quad \textbf{if} \quad \begin{cases} M \models \neg C, \text{ and} \\ \ell \text{ is undefined in } M. \end{cases}$$

- *Fail*

$$M \parallel F, C \to \mathit{fail} \quad \textbf{if} \quad \begin{cases} M \models \neg C, \text{ and} \\ M \text{ contains no decision literals.} \end{cases}$$

- *Back Jump*

$$M\ell^d N \parallel F, C \to M\ell' \parallel F, C \quad \textbf{if} \quad \begin{cases} M\ell^d N \models \neg C \text{ and there is some} \\ \text{clause } C' \vee \ell' \text{ such that:} \\ \quad F, C \models C' \vee \ell', \\ \quad M \models \neg C', \\ \quad \ell' \text{ is undefined in } M, \text{ and} \\ \quad \ell' \text{ or } \neg\ell' \text{ occurs in } F \text{ or in } M\ell^d N. \end{cases}$$

- *Learn*

$$M \parallel F \to M \parallel F, C \quad \textbf{if} \quad \begin{cases} \text{all atoms in } C \text{ occur in } F, \text{ and} \\ F \models C. \end{cases}$$

- *Forget*
  $M \parallel F, C \rightarrow M\ell \parallel F \quad \textbf{if} \quad M \models C$
- *Restart*
  $M \parallel F \rightarrow \emptyset \parallel F$

The given formula is satisfiable if neither *Pure Literal*, *Unit Propagate*, *Back Jump*, nor *Decide* is applicable and the system is not in the *fail* state. In particular, the truth assignment M in the final state is an example of a satisfying assignment for the input formula. Moreover, the rules are not applied in a completely random order. The priorities for applying the rules are as follows: (i) If *Fail* or *Back Jump* are applicable, they are applied. Otherwise, (ii) *Unit Propagate* and *Pure Literal* are applied if possible. (iii) Only if no other rule can be applied, *Decide*, *Learn*, *Forget*, or *Restart* is used. The main motivation is quite straightforward — reducing the amount of guessing as much as possible. The use of *Decide*, *Learn*, *Forget*, and *Restart* rules is then subject to heuristics. These heuristics may vary solver to solver and are one of the main subjects of the on-going research (e.g., [82, 72]). Modern SAT solvers are able to deal with real-life SAT problem instances containing millions of variables and clauses.

4.4.2 SMT Solvers

The satisfiability modulo theories (SMT) problem is a decision problem for first-order logical formulas with respect to combinations of background theories expressed in the classical first-order logic with equality. An SMT instance is a formula in first-order logic where some function and predicate symbols have additional interpretations and SMT is the problem of determining whether such a formula is satisfiable. Example predicates involve linear inequalities (e.g., $4x + 2y \geq z$), equalities involving uninterpreted terms and function symbols (e.g., $f(f(x, y), z) = f(x, z)$ where f is some unspecified binary function), or bit-vector arithmetic with equalities (e.g., $u \oplus (CAFE)_{16} = w \ll x$ where $\oplus$ and $\ll$ denote the "*xor*" and "*left shift*" bit operations, respectively). Formulae with atoms from a specific theory are decided using their respective decision procedures. Then, approaches for combining such procedures (e.g., the Nelson-Oppen procedure [103]) are used for mixed formulae (where some variables are used in atoms of several different theories).

Early attempts to solve SMT instances involved translating SMT instances to Boolean SAT instances. For example, a 32-bit integer variable would be encoded by 32 variables, each representing one bit with the appropriate ordering, and word-level operations would be replaced by lower-level logic operations on the bits. However, this loss of the high-level semantics of the underlying theories means that the Boolean SAT solver has to work much harder than necessary as it must (re-)discover trivial theory facts (such as commutativity for the bit-vector $\oplus$ operation). This observation led to the development of the so-called *lazy* SMT approaches where SMT solvers tightly integrate the Boolean reasoning of a CDCL-style search with theory-specific solvers that handle conjunctions of predicates from a given theory [10].

Chapter 5

Hardware Verification Techniques

This chapter discusses contemporary hardware verification techniques with a specific focus on the ones used during the development of pipelined microprocessors. The chapter is organized as follows. The first two sections describe some of the state-of-the-art approaches for automatic verification of hardware using functional verification and formal methods which are related to the aim of the thesis. The last section is then dedicated to various ways of large memory modeling that represents another important research topic as it can boost the performance of the two former approaches.

Lots of work has been invested in the area of formal and functional verification of hardware. Unfortunately, according to financial reports of major hardware developers, functional verification was preferred to formal approaches in the previous decades.[1] This can be explained by the fact that formal methods were usually time-consuming and more difficult to deploy. Yet, in the last years, with the great advances in computational power of modern processors and advances in research, formal methods are becoming more popular as well.[2]

5.1 Functional Verification of Hardware

Although this topic is not the focus of the thesis, functional verification is one of the most popular techniques for verification of hardware. Therefore, it should be mentioned, at least briefly, so a more complete view of topics related to the thesis is provided to the reader. The functional verification typically generates a set of constrained and/or random test vectors and compares the behavior of the system for these vectors with the behavior specified by a reference model. In order to get a high level of coverage of the system's state space, it is required to (i) discover a way to generate input vectors that cover critical parts of the state space, and/or (ii) increase the number of tested vectors. Coverage (e.g., code coverage, functional coverage, path coverage) dynamically measures the completeness of state-space exploration and allows the verification engineer to improve quality of input test vectors, usually by adding constraints, to achieve an even higher level of coverage. Full automation of the process can be achieved, for example, by an intelligent program that controls coverage results and chooses parameters of a new test vector to reach better coverage. Such an approach is called a *coverage-driven verification*.

[1] Source: Gary Smith EDA, Oct 2010.

[2] Source: Gary Smith EDA, Oct 2017.

The VCS [126] tool for functional verification is used by many major corporations in a commercial sphere. VCS speeds up the verification process by running several tasks in parallel on machines with multiple cores. A special proprietary technology for generating expressions named Echo [126] is used for automatic creation of stimuli to efficiently cover the state space specified by the user who typically adds constraining formulae to the code. The expressions are generated by constraint solvers that find an appropriate solution to the supplied constraints while minimizing conflicts between them. The VCS tool uses a *uniform coverage database* for storing coverage statistics which can eliminate redundant execution of certain test vectors for designs that were only partially modified (e.g., by finding identical parts of the designs).

In [142], authors describe a functional verification approach applied when checking the implementation of RISC-V processor [115] designed in Codasip framework [1]. The verification is based on the RTL simulation (running in Veloce emulator [90]) and the *universal verification methodology* (UVM) [5] which is a standardized methodology for verifying integrated circuit designs. The approach leverages the fact that, in Codasip, models of the processor can be described at various level of detail, that is, typically instruction- and cycle-accurate as we have shown in Section 3.3.2. To keep up the pace with the RTL emulation, a fast software simulator generated from the instruction-accurate level description is taken as a golden model for the verification task. The UVM is then used for orchestration of loading and execution of the test bench stimuli into both runtime environments as well as for asserting the equality of the obtained results (e.g., contents of register files).

The ArchC [114, 6] framework provides a co-simulation tool allowing a designer to verify conformance of two different models of the architecture. The ArchC verification approach is based on a transaction verification methodology which tracks down every update to storage devices of both models, marking them with timestamps to show when they happened. By comparing the sequence of transactions generated throughout the execution, the ArchC verifier can tell whether both models are consistent. A deficiency of the method is the maximum frequency of the simulation which is claimed to be in the order of units of megahertz for a MIPS processor. Such a frequency may not be sufficient for applications that need to communicate using high-speed interfaces.

In HAVEN [123], the issue with the slow speed of simulation is resolved by utilizing the inherent parallelism of a hardware system to accelerate its functional verification. The verified system together with several necessary components of the verification environment is moved to a field-programmable gate array (FPGA). The frequency achieved by the acceleration is approx. 125 MHz which is significantly higher than the frequency of emulation-based solutions available at a comparable price. The current disadvantage of the technique may be a lack of ability to automatically drive the generation of test vectors to target coverage holes given by continuously measured coverage.

Another tool for functional verification is ZamiaCAD [129]. It is a modular and extensible platform with IDE for hardware design. The main advantage of this platform is its ability to automatically locate design flaws in microprocessor designs at RTL. As an input, the user has to provide a set of independent tests where both failing and passing tests are present. The error localization is done by statistical simulation [83] which is refined using dynamic and static slicing [76, 137]. Besides this feature, ZamiaCAD also offers the ability to highlight results computed by static analysis directly in HDL representation including the cone of influence, dead code, etc.

5.2 Formal Verification of Pipelined Microprocessors

In this section, we would like to describe formal verification techniques with a high degree of automation used during microprocessor design. When concentrating on verification of microprocessors, the approach of theorem proving (cf., e.g., [70, 118, 61]) is often considered. There are multiple successful industrial applications of theorem proving, including, e.g., a proof of correctness of the floating-point arithmetic of the Intel Itanium processor [57] or the fully verified design of the VAMP microprocessor, which was verified using the PVS theorem prover [13]. However, theorem proving typically requires a significant level of expertise and user intervention. A typical microprocessor verification cost using theorem proving is counted in person-years.

Because this thesis aims at the maximal automation of the proposed techniques, we will concentrate more on automated techniques. An approach inspired by theorem proving is the approach of automatic generation of properties satisfied by a given design (cf., e.g., [48, 116, 94, 43, 38]). This approach is based on automatic learning of dependencies or properties from simulation traces or data-flow graphs. Unfortunately, the approach is primarily suited for an initial understanding of the design since it lacks the ability to completely verify the whole microprocessor design. More automation is also offered by the approach of model checking based on a systematic exploration of the state space of the verified system. The approach of bounded model checking (BMC) [14], exploring the state space of a verified system up to certain depth only, and related approaches such as IPC [105] have become very popular in practice, leveraging the recent advances in automatic decision procedures, especially, SAT and/or SMT solvers [134, 132, 133, 45].

Majority of the work on automated formal verification of pipelined microprocessors can be separated into two main branches: (i) correspondence checking between various abstraction levels of implementation and (ii) verification of the microprocessor with respect to generic properties of pipelined microprocessors. These two branches are often supplemented by (iii) methods looking for undesirable patterns in the microprocessor implementations. Each of these topics is more discussed in the following subsections.

5.2.1 Correspondence Checking

Despite the formal methods of correspondence checking have a history dating back over decades [62], one of the key ideas used in correspondence checking among the ISA and RTL implementations is described in [27]. Typically, the most challenging part of the ISA-RTL correspondence checking is to find an abstraction function α_{ISA} that maps states of the RTL-level states to ISA level such that the α_{ISA} mapping is maintained in each cycle of the RTL level operation. The key contribution of [27] is showing that the abstraction function α_{ISA} could be computed automatically by symbolically simulating the microprocessor as it clears out instructions out of the pipeline (typically, by inserting `NOP` instructions into the pipeline). Indeed, most pipelined processor designs already have a mechanism for clearing instructions, because this is required to bring the pipeline to an idle state when dealing with exceptional conditions, such as halting or interrupt handling.

For a single-pipelined microprocessor, the following verification task, schematically depicted in Fig. 5.1, can be used for checking the ISA-RTL equivalence. Given the function α_{ISA} the task consists of:

1. choosing an arbitrary *legal* starting RTL state s_{RTL},

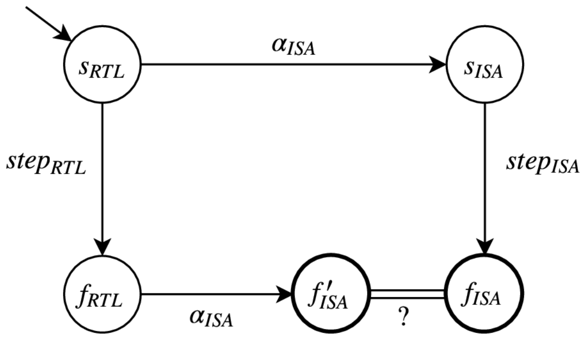


Figure 5.1: Correspondence checking approach between ISA and RTL processor descriptions as it is proposed in [27].

2. symbolically computing the corresponding ISA state s_{ISA} by finishing partially executed instructions in the pipeline, i.e., $s_{ISA} := \alpha_{ISA}(s_{RTL})$,
3. obtaining an ISA state f_{ISA} by executing the instruction in the ISA model, i.e., $f_{ISA} := step_{ISA}(s_{ISA})$,
4. getting an RTL state f_{RTL} by running the instruction for a normal pipeline cycle in the RTL model, that is, $f_{RTL} := step_{RTL}(s_{RTL})$,
5. computing the corresponding ISA state f'_{ISA} after making the normal cycle in the RTL model, i.e., $f'_{ISA} := \alpha_{ISA}(f_{RTL})$,
6. comparing the programmer-visible parts of the designs, that is, checking whether $f_{ISA} = f'_{ISA}$.

The original approach [27] utilizes 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 [24], EUFM is extended by positive equality of uninterpreted functions (PEUF) which greatly reduces the time needed for verification. The works [136, 135, 56] 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. Moreover, specifying an arbitrary legal starting RTL state is a hard problem and requires significant user intervention, e.g., by writing assertions related to each microprocessor signal. The difficulty of identifying such assertions can be seen, for instance, in a recent work [23] aiming at verification of microprocessors using the above-described technique where non-trivial invariants related to pipeline control signals had to be added explicitly.

A correspondence checking method is also proposed in [75]. The main idea of the approach is based on proving equivalence of data-flow graphs (DFGs) that are extracted from instruction-accurate and cycle-accurate models by unrolling the transition relation for the needed number of time frames. The method benefits from reducing sizes of DFGs by finding *potentially equivalent pairs* (PEPs) and proving their equivalence. Therefore, the size of a DFG to be analyzed is much smaller. The method can be divided into the following steps:

1. Detection of PEPs by computing values of each node in both instruction- and cycle-accurate DFGs for some random test pattern placed to the graph inputs. Any pair of nodes that have the same simulation values are considered to be a PEP.

2. Prove the PEP equivalence using model checking. The model checking is invoked for each of the PEPs.

3. Merge equivalent PEPs and continue with Step 1 until graph outputs are proven equivalent. If the PEP nodes are shown to be not equivalent, then a counter-example trace is used to prune the set of PEPs.

To achieve a better performance, additional techniques such as constant propagation over the DFGs or graph rewriting rules are used.

Another, yet similar approach to correspondence checking of the control of a microprocessor is described in [78]. The work proposes a method of automatic formal verification of a pipelined implementation against its ISA specification by using IPC [105] that all assertions of all instructions are satisfied and to prove the 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 manual user intervention.

Checking of the pipeline control of a microprocessor is also addressed in [81]. The paper presents a formal verification technique called *unpipelining*. At first, the unpipelining technique analyzes the pipeline structure of a design. The analysis works with a graph of the structure of the pipeline control where it identifies and classifies (by using pattern-matching) all the control logic into three classes that deal with the basic pipeline hazards, i.e., stalling, clearing, and bypassing. Using the results of this analysis, the method automatically reverse-engineers a pipeline through a series of transformations called *pipeline deconstruction*. Each application of the pipeline deconstruction shortens the pipeline by merging its last two stages into a single stage. If all the deconstruction transformations are successful, the model is transformed into a functionally equivalent unpipelined design. This equivalent design of the RTL specification can then be checked for correspondence with the ISA description. The main deficiency of the method is that it cannot be used for designs that implement, for instance, delayed branches or branch prediction.

Compared to the above approaches, the approach of correspondence checking presented in Chapter 8 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 fully formal verification, it can find bugs not found by functional verification.

5.2.2 Checking of Generic Properties of Pipelined Microprocessor

Instead of concentrating on proving the full ISA-RTL correspondence which, as we have seen, could be a rather complicated task, the approaches listed in this section aim at automated verification wrt one or more specific properties that any correct pipelined microprocessor should satisfy.

The approach proposed in [69] introduces the so-called self-consistency check that compares results of executions of an instruction in two scenarios wrt a property given by the user. For example, for a property concerning data hazards, the approach works with (i) executions of an instruction enclosed by the finite number of random instructions within the pipeline and (ii) executions of the same instruction surrounded by `NOP` instructions only.

The main drawback of this approach is that a user has to list all valid instructions and their possible combinations. Further, the conformance established by the approach is valid only up to the given number of instructions.

In [2], 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 [78]. 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 [105] is used to check several properties implying correctness of the pipeline behavior. Again, both of the above methods require significant manual user intervention.

The works [98, 95, 96] propose general properties of the correct behavior of a typical single-pipelined implementation of a microprocessor. For instance, the work [98] includes definition of a rule that prevents an undesirable duplication of an instruction within the pipeline. These properties together with an ADL description of a processor are then converted to a BMC problem to find possible counterexamples [97].

In contrast with the above approaches, the approach for showing an absence of problems caused by pipeline hazards proposed in Chapter 9 is almost fully automated—the only step required from the user is to identify the architectural resources (such as registers and memory ports) and the program counter.

5.2.3 Looking for Undesirable Design Patterns

Searching for specific design patterns that could cause unwanted behavior of the designed system (e.g., proper dealing with high impedance values in HDL languages) could be a rather simple but very efficient way to find some types of bugs. Spyglass Lint [127] is a pattern-based static RTL checker delivered with the ASIP Designer framework [125]. It contains a set of customizable rules which are aimed to help with revealing flaws in early phases of the microprocessor development. Certainly, such rules only approximate reality and can produce many false alarms. However, this information can be used to improve the performance of other (more sophisticated) tools (e.g., [126, 75]) by providing useful information about the verified system. A somewhat similar approach is also offered by the Sigasi framework. In [122], the authors state that the framework is, for example, capable of detecting signals and variables that are never read/written, dead states in state machines, or `case` statements that do not cover all choices.

As one can see, static analysis of a hardware system can be used as an entry point for more advanced techniques by providing hints that can, for instance, narrow the state space explored by a model checker used for subsequent detailed analysis of the given system.

5.3 Large Memory Abstraction

Numerous works have focused on memory abstraction, notably within the area of formal verification. Designs with large embedded memories are quite common and have many applications. However, these embedded memories add further complexity to formal verification tasks due to an exponential increase in the state space with each additional memory bit. With explicit modeling of large embedded memories, the search space frequently becomes prohibitively large to analyze. Therefore, it is important to use abstract models of such memories.

Theories for reasoning about arrays [86, 104] are often used as a formal basis in current approaches for memory abstraction, especially the work on an extensional theory of arrays [124]. Intuitively, this theory formalizes the idea that two arrays are equivalent if they have the same value at each index. An example of such an approach has been presented in [50]. This work specializes in reasoning about safety properties of systems with arrays. In the work, an automatic algorithm for constructing abstractions of memories is presented. The algorithm computes the smallest sound and complete abstraction of the given memory.

In [19], the authors introduce a theory of arrays with quantifiers which is an extension of [124]. Moreover, they define the so-called *array property fragment* for which the authors supplement a decision procedure for satisfiability. A modification of the decision procedure for purposes of correspondence checking is proposed in [74] and implemented in [75].

Another method for large memory modeling is described in [131]. The memory state is represented by an ordered set containing triplets composed of (i) an expression denoting the set of conditions for which the triplet is defined, (ii) an address expression denoting a memory location, and (iii) a data expression denoting the contents of this location. For this set, a special implementation of write and read operations wrt the above-described representation of the memory is defined. The abstracted memory interacts with the rest of the circuit using standard *enable*, *address*, and *data* signals. The size of the set is proportional to the number of memory accesses. Further, in [25], the same author extends the approach in a way that it can be used for correspondence checking by applying the so-called shadowing technique for read operations. The technique is used on all read operations when the second of the two verified models is symbolically executed. In contrast with the original read operation, the modified one delegates computation of the return value to the memory used in the first model if a requested address has no record in the above-defined set. Such an approach ensures (otherwise missing) consistency of read operations of both verified models.

The work [63] formally specifies and verifies a model of a large memory that supports efficient simulation. The model is tailored for Intel x86 implementations only in order to offer a good trade-off between the speed of simulation and the needed computational resources.

A common disadvantage of [50, 74, 131, 25] is the fact that they omit support for addressing different sizes of data which is considered, e.g., in [63]. On the other hand, in [63], the authors assume starting from the nullified state of the memory, not from a random state.

Some of the other proposed works describe a smarter encoding of formulas including memories into CNF [84, 49]. In the thesis, the problems linked to CNF transformation are not discussed, however, the ideas in [84, 49] can be potentially applied to it. An example of a tool based on the method coupled with CNF is the *Bit Analysis Tool* [84] (BAT) which automatically builds abstraction for memories over bounded time intervals. As an input, the BAT uses custom LISP-based language. The version of the verified system with abstracted memories is created in the following steps:

1. The design to be verified is simplified through pre-defined rewrite rules applied on the level of terms of the BAT language.

2. An *equality test relation* that relates memories that are directly compared for equality is built over the set of memory variables.

3. The transitive closure of the test relation is computed. Such a closure is an equivalence relation.

4. An *address set* is computed for each of the equivalence classes. The address set contains only addresses that are relevant for a given class.

5. For all addresses in address set, a shorter bit vector for addressing the abstract memories is created. The size of the vector is proportional to the number of memory accesses.

6. The behavior of memories is changed to be compatible with the new addressing style.

7. Original memories and addresses are replaced with their abstract counterparts.

A description of a system together with the checked properties is then efficiently transformed into a CNF formula. Similarly to previous approaches, there is no support for addressing different sizes of data.

In Chapter 7, we propose another approach to generate abstractions of memories which support addressing of arbitrary addressable units, such as bytes and words (unlike [50, 74, 131, 25]), with multiple read and write ports (in contrast with [50, 74]), and it allows the memory to start from a random initial state (not available in [63]). Our algorithm is also not bound to any specific verification technique (unlike [84, 49]).

Chapter 6

Goals of the Thesis

The general idea of the thesis is to design new hardware verification techniques optimized for use in the process of hardware/software co-design. The key idea is to improve and/or develop verification techniques with an emphasis on (i) maximal amount of automation, (ii) efficiency, and (iii) ability to deliver continuous feedback about the verification process. The proposed techniques should be in particular applicable to the class of ASIPs that are broadly used in light-weight embedded devices with the following properties:

- 32bit architecture,
- in-order execution of instructions,
- memories with multiple read/write ports,
- I/O communication through buses, and
- ability to handle interrupts.

The first goal of the thesis is to develop formal methods for checking *correspondence* of designs on various levels of abstraction. This goal can be narrowed down as follows:

- The proposed formal technique should be able to verify correspondence between RTL and ISA specifications of a processor.
- The technique should be scalable for use in parallel processing.
- The method should deliver (at least partial) results in the order of minutes.
- The approach should be able to cope with the complex issues brought by the presence of large memories in designs.

The above-specified first goal is addressed in Chapter 8 which introduces a new algorithm for verifying correspondence between the RTL and ISA microprocessor specifications with a high degree of automation together with a new method for modeling large memories and register files described in Chapter 7.

The second goal of the thesis is to develop new methods for checking correctness of various functional parts of a microprocessor, especially those associated with the pipeline control. This goal can be more expanded as follows:

- The proposed formal technique should be able to work on a low-level RTL specification of microprocessors with a single pipeline.

- The technique should be able to benefit from parallel processing.
- The method should be able to split the verification task into smaller parts that can be processed separately and thus deliver results in a reasonable time (in the order of minutes).
- The efficiency of the proposed method should not downgrade significantly for microprocessors with wide data-paths.

Concerning this topic, in Chapter 9, we propose an approach for detection of problems caused by data and control hazards in pipelined microprocessor designs.

Chapter 7

Large Memory Abstraction

This chapter describes a technique for automatic generation of abstract models of memories that can be used for efficient formal verification of hardware designs. Our approach is able to handle addressing of different sizes of data, such as quad words, double words, words, or bytes, at the same time. The technique is also applicable to memories with multiple read and write ports, memories with read and write operations with zero- or single-clock delay, and it allows the memory to start with a random initial state allowing one to formally verify the given design for all initial contents of the memory. Our abstraction allows large register-files and memories to be represented in a way that dramatically reduces the state space to be explored during formal verification of microprocessor designs as witnessed by our experiments.

7.1 Introduction

As we have already said, the complexity of the verification process of microprocessor designs is usually significantly influenced by the presence and size of the memories used in the design because of an exponential increase in the size of the state space of the given system with each additional memory bit. Therefore the so-called *efficient memory modeling* (EMM) techniques that try to avoid explicit modeling of the memories are being developed.

In this chapter, we present an approach to automatic generation of abstract memory models whose basic idea comes from the fact that formal verification often suffices with exploring a limited number of accesses to the available memory, and it is thus possible to reduce the number of values that are to be recorded to those that are actually stored in the memory (abstracting away the random contents stored at unused memory locations). Expanding the basic idea, we propose an approach that allows one to represent memories with various advanced features, such as different kinds of endianness (big or little), read and write delays, multiple read and write ports, and different sizes of addressable units (e.g., bytes, words, double words). As far as we know, the ability to handle all of the above mentioned features differentiates our approach from the currently used ones. Moreover, our technique is applicable in environments requiring a very high level of automation (e.g., processor development frameworks), and it is suitable for formal verification approaches that aim at verifying a given design for an arbitrary initial contents of the memory. Further, our abstract memory models can be used within formal verification in a quite efficient way as proved by our experiments.

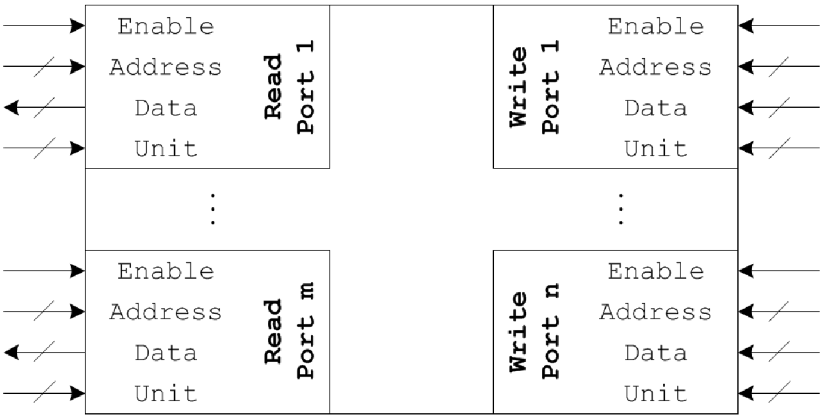


Figure 7.1: Memory interface.

The following sections provide a description of our technique of automated memory abstraction that was originally published in [32]. As we have already said, its basic idea is to record only those values in the memory that are actually used (abstracting away the random contents stored at unused memory locations).

7.2 Memories To Be Abstracted

In our approach, we view a memory as an item of the verified design with the interface depicted in Fig. 7.1. The interface consists of (possibly multiple) read and write ports. Each port is equipped with `Enable`, `Address`, `Data`, and `Unit` signals. When the `Enable` signal is down, the value of the `Data` signal of a read port is undefined. When dealing with a write port, no value is stored into the memory through this port. On the other hand, when the `Enable` signal is up, the memory returns/stores data from/into the cell associated with the value of the `Address` signal. In the special case when multiple ports are enabled for writing into the same memory cell, the result depends on the implementation of the memory. We support two variants: (i) either a prioritized port is selected or (ii) an undefined (random) value is stored to the multiply addressed memory cell.

The size of the addressed unit can be modified by the `Unit` signal. When the size of the accessed unit is smaller than the size of the greatest addressable unit, the most significant bits of the `Data` signal are filled up with zeros. It is also assumed that the size of any addressable unit is divisible by the size of the least addressable unit, and thus for the `Data` signal it is sufficient to transfer the size of the addressed unit expressed as a multiple of the least addressable unit only (instead of the actual number of bits of the unit). Finally, if the memory allows addressing of a single kind of units only, then the `Unit` signal can be omitted.

7.3 Abstraction of the Considered Memories

Our abstraction preserves the memory interface, and hence concrete memories can be easily substituted with their abstract counterparts. We will first describe the basic principle of our abstraction on memories with a single addressable unit only. An extension of the approach for multiple addressable units will be discussed later. Moreover, we assume reading with no

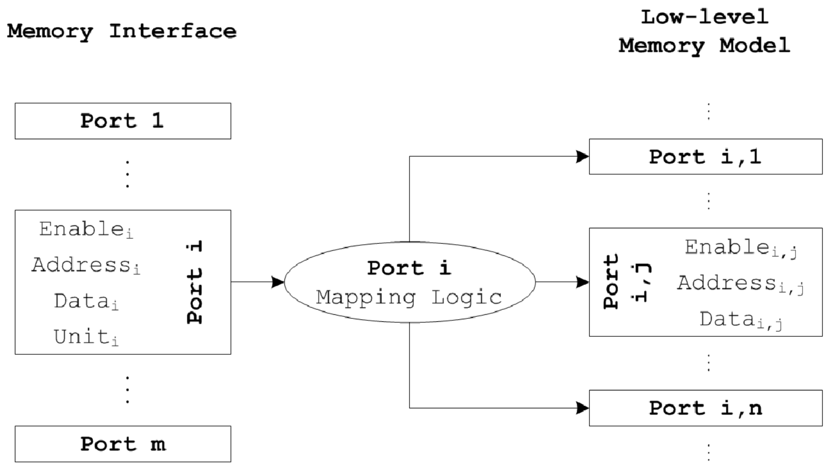


Figure 7.2: Memory mapping.

delay and writing with a delay of one cycle. An extension to other timings will be described in Section 7.5.

The abstract memory effectively remembers only the memory cells which have been accessed. Internally, the memory is implemented as a table consisting of some number d of couples of variables storing corresponding pairs of addresses and values (a, v). When using bounded model checking (BMC) as the verification technique, the needed number d of address-value pairs can be easily determined from the depth k of BMC as the following holds $d = k * (m + n)$ where m and n denote the number of read and write ports, respectively. For unbounded verification, the number d can be iteratively incremented until it is sufficient. The incrementation is finite since the number of memory cells is finite. The memory also remembers which of the pairs are in use by tracking the number $r \in \{0, \ldots, d\}$ of couples that were accessed (and hence the number of the rows of the table used so far).

When the memory is accessed for reading, the remembered address-value pairs (a_1, v_1), $\ldots$, (a_r, v_r) that are in use are searched first. If a location a_{rd} that is being read has been accessed earlier, then the value v_i associated with the appropriate address $a_i = a_{rd}$ is simply returned. On the other hand, if a location that has never been accessed is being read, a corresponding pair is not found in the table, and a new couple (a_{rd}, v_{rd}) is allocated. Its address part a_{rd} will store the particular address that is accessed while the value v_{rd} is initialized as unconstrained. However, the variable representing the value v_{rd} associated with the accessed location a_{rd} is kept constant in the future (unless there occurs a write operation to the a_{rd} address). This ensures that subsequent reads from a_{rd} return the same value. In the case of writing, the address a_{wr} and value v_{wr} are both known. When writing to a location that has not been accessed yet, a new address-value pair (a_{wr}, v_{wr}) is allocated in order to memoize the given memory access. Otherwise, a value v_i associated with the given address $a_{wr} = a_i$ is replaced by v_{wr}.

7.4 Dealing with Differently Sized Data

To support different sizes of addressable data (including reading/writing data smaller than the contents of a single memory cell of the modeled memory), we split our abstract memory into a low-level memory model and a set of functions mapping accesses to ports of the modeled memory to ports of the low-level memory. The idea of this approach is shown in Fig. 7.2 and further discussed below.

The low-level memory consists of cells whose size equals the size of the least addressable unit of the modeled memory, and therefore, for low-level memory, the `Unit` signal can be omitted. In the low-level memory, values of units that are larger than the least addressable unit are stored on succeeding addresses. In order to allow reading/writing the allowed addressable units (including the greatest one) in one cycle, the number of read and write ports of the low-level memory is appropriately increased. The resulting number of ports of the low-level memory is equal to $m * n$ where m is the number of interface ports and n is the number of distinct addressable units. The latter can be expressed as the quotient of bit-widths of the greatest (w_{gau}) and the least (w_{lau}) addressable unit. In other words, for each port of the memory interface there are n corresponding ports of the low-level memory model. Therefore, we use double indices for the low-level memory ports in our further description.

In particular, let $enable_i$, $data_i$, $address_i$, and $unit_i$ be values of signals of the port i of the memory interface, and let $enable_{i,j}$, $data_{i,j}$, and $address_{i,j}$ have the analogical meaning for the low-level memory port i, j. Then, the value of the $enable_{i,j} \in \mathbb{B}$ signal can be computed as $enable_i \wedge unit_i \geq j$ where $enable_i \in \mathbb{B}$ and $1 \leq unit_i \leq n$. This means that the required number of low-level memory ports are activated only. Next, the value of $address_{i,j}$ can be expressed as $address_i + j - 1$ for the little endian version of the memory and $address_i + unit_i - j$ for the big endian version, respectively. These expressions follow from the fact that larger units of the original memory are stored as multiple smallest addressable units stored at succeeding addresses in the low-level memory.

Further, for transfers of data, separate mappings for read ports and write ports must defined. In the case of a write port, the data flow into the low-level memory, and the value of the $data_{i,j}$ signal can be computed as $slice(data_i, unit_i * w_{lau} - 1, (unit_i - 1) * w_{lau})$ where $slice$ is a function extracting the part of the first argument (on the bit level) that lies within the range given by the second and third arguments (with the bit indices being zero-based). Finally, for a read port, for which data flow from the low-level memory, the value of the $data_i$ signal can be expressed as $concat(ite(enable_{i,n} \vee \neg enable_{i,1}, data_{i,n}, 0), ..., ite(enable_{i,2} \vee \neg enable_{i,1}, data_{i,2}, 0), data_{i,1})$ where $concat$ is a bit concatenation and ite ("if-then-else") is the selection operator. Thus, the data value is composed from several ports of the low-level memory, and the most significant bits are zero-filled when the read unit is smaller than the greatest one. Note that according to the semantics of the `Enable` and `Data` signals (described in Section 7.2), in the case when $enable_{i,1}$ is false (i.e., no unit is read), the value of the $data_i$ signal is undefined.

7.5 Further Extensions of the Abstract Memory Model

To broaden the range of memories that we can abstract, we further added support for more memory timing options, in particular for the one-cycle-delay reading and the zero-delay writing. The former can be achieved by simply connecting a unit buffer to the data signal of the memory interface. For the latter case, a special attention must be paid to the situation when both read and write operations over the same address are zero-delayed. In such a situation, it is required to append an additional logic that ensures that written data are propagated with zero delay to a given read port.

Moreover, for a practical deployment in correspondence checking, our model has also been extended by applying the shadowing technique described in [25]. In particular, during correspondence checking, both models are executed in a sequence. The shadowing technique deals with potential inconsistencies that can arise when both models read from the same uninitialized memory cell—indeed, in this case, a random value is to be returned, but the same one in both models. To ensure this when the shadowing is used, the return value of the read operation is obtained from the memory in the design executed first whenever the value is not available in the second design.

7.6 Implementation and Experiments

The memory abstraction that we generate in the above described way can be encoded in any language for which the user can provide templates specifying (i) how to express declarations of state and nonstate variables, (ii) how to encode propositional logic expressions over state and nonstate variables, (iii) and how to define initial and next states of state variables. We currently developed these templates for the Cadence SMV language [87].

In order to prove usefulness of the described abstraction technique, we used our abstract memory generator within the approach proposed in [31] (further described in Chapter 8) for checking correspondence between the ISA and RTL level descriptions of microprocessors, which we applied to several embedded microprocessors. Briefly, in the approach of [31], the ISA specification and VHDL model of a processor are automatically translated into behavioral models described in the language of a model checker (the Cadence SMV language in our case). These models are then equipped with an environment model, including architectural registers and memories, which can be abstracted using the technique proposed in this chapter. All these models are composed together, and BMC is used to check whether 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. An experimental version of the described approach was integrated into the Codasip IDE [1] processor development framework.

Our approach was tested on the following processors: *TinyCPU* is a small 8-bit test processor with 4 general-purpose registers and 3 instructions that we developed mainly for testing new verification approaches. *SPP8* is an 8-bit ipcore with 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 allowing one, e.g., to load/store both bytes and words from/to the memory. *Codea2* is a 16-bit processor with 4 pipeline stages partially based on the MSP430 microcontroller developed by Texas Instruments [42]. The processor is dedicated for signal processing applications. It is equipped with 16 general-purpose registers, 15 special registers, a flag register, and an instruction set consisting of 41 instructions, where each may use up to 4 available addressing modes. Our experiments were evaluated for two modifications of the processor—using memory with and without multiple addressable units.

Our experiments were run on a PC with Intel Core i7-3770K @3.50GHz and 32 GB RAM using Cadence SMV (the build from 05-25-11) and GlueMinisat (version 2.2.5) [101] as an external SAT solver. The results can be seen in Table 7.1. The first three columns give a name of verified processor, a size of its register file, and a size of the memory. The next columns give the results obtained from the verification—in particular, the average time needed for verification of a single instruction with the abstraction applied or not-applied in different combinations on the register file and the memory. In the first case, both the register file and the memory were modeled explicitly which, for larger designs such as Codea2, led to out-of-memory errors (“o.o.m.”). Next, the abstraction was only used for

Table 7.1: Verification results.

Processor	Reg. File Size	Memory Size	Explicit Memory	Abs. Reg. File	Abs. Memory	All Abs.
TinyCPU	4 x 8bit	-	0.151 s	0.41 s	-	-
SPP8	16 x 8bit	256 x 8bit	5.06 s	1.11 s	3.66 s	0.452 s
SPP16	16 x 16bit	2048 x 8bit	266 s	92.2 s	1.23 s	0.822 s
Codea2_single	32 x 16bit	32768 x 16bit	o.o.m.	o.o.m.	4.30 s	4.44 s
Codea2_mult	32 x 16bit	65536 x 8bit	o.o.m.	o.o.m.	4.75 s	4.89 s

single	Single addressable unit used	o.o.m.	Out of memory error occurred
mult	Multiple addressable units used		

register files. Even though better results were obtained this way for the SPP8 and SPP16 processor designs, the verification still ran out of system resources for Codea2 because of the explicitly modeled memory. In the last two cases when either only memories or both memories and register files of the verified processors were abstracted, verification was able to finish even for larger designs. We explain the 10 % deterioration between verification times for the Codea2 processor with and without presence of multiple addressable units by the complexity of the additional logic.

Finally, we note that for very small memories and memories with many possible accesses (caused by, e.g., a higher verification depth during BMC), the overhead brought by the abstraction can result in worse verification times as can be seen in the case of the register file of the TinyCPU and Codea2 processors. Moreover, for SPP8, where only a few instructions directly access the memory, and thus only a few instructions influence the average verification times, the overhead caused by the abstraction introduces worse than expected average verification time when abstracting the memory only. In practice, we deal with this problem by defining heuristics that computes whether or not it is better to use the explicit or the abstract description of a given memory.

7.7 Conclusion

We have presented an approach of memory abstraction that utilizes the fact that formal verification often suffices with exploring a limited number of accesses to the available memory, and it is thus possible to reduce the number of values that are to be recorded to those that are actually stored in the memory. Our approach allows one to abstract memories with various advanced features, such as different kinds of endianness, read and write delays, multiple read and write ports, and different sizes of addressable units. The technique is fully automated and suitable for usage within processor development frameworks where it can bring a significant improvement in verification times.

Chapter 8

RTL-ISA Correspondence Checking

In [31], we proposed an automated approach built on a formal basis and intended to be used within an automated microprocessor design framework 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 the 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.

Compared to the techniques proposed, e.g., in [27, 69], the approach presented in this chapter does not provide full formal verification since (i) it uses BMC, (ii) it does not consider any mutual influence among the instructions, and (iii) it may limit the bit-width of input data in some cases. Hence, it may under-approximate the behavior of the verified designs. 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.

An experimental version of the approach has been implemented within the Codasip IDE [1] and successfully tested in several case studies. The experiments included a non-trivial 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 single-pipelined 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).

Section 8.1 of this chapter provides a background on the expected design flow for which our approach is optimized. The main idea of the proposed method is then described in Section 8.2. Sections 8.3, 8.4, 8.5, and 8.6 provide more details about the way we model processors and about the actual verification process. Potential parallelization options of the proposed method are given in Section 8.7. Experiments are discussed in Section 8.8. Section 8.9 concludes the chapter.

8.1 Background: Expected Design Flow

Our work was originally motivated by a request to provide some support for verification on a formal basis for the Codasip IDE [1] described in Section 3.3.2, 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).

Our method uses both the IA and CA descriptions given in CodAL ADL 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. The above can be obtained by automatically generated extractor of instruction semantics for the target compiler [64, 130]. 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 the case of processors with multicycle instructions, we need to know the maximum number of cycles each instruction needs to complete its execution.

For our approach, as stated above, 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. 8.1. 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 a static single assignment (SSA) format with a few simple optimizations.

8.2 The Main Idea of the Proposed RTL-ISA Correspondence Checking

We concentrate on checking a 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, branch prediction, caches, etc. We, however, believe that such an approach is still useful, especially when combined with other techniques (such as the one discussed in Chapter 9).

The proposed method uses the bounded model checking as an automated reasoning engine. A typical approach to use the (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

Figure 8.1 A description of the `add` instruction in CodAL.

```
element reg represents regs {
    use imm4 as num;
    assembler { "r" ~ num };
    binary { num };
    return { num; };
}
element add {
    assembler { "ADD" };
    binary { OP_ADD:4 };
    return { OP_ADD; };
}
set opc = add, /* ... */;
element instr_alu {
    use reg as { dst, sA, sB };
    use opc;
    assembler { opc dst "," sA "," sB };
    binary    { opc dst     sA     sB };
    semantics {
        switch (opc) {
            case OP_ADD:
                regs[dst] = regs[sA] + regs[sB];
                cf = func_add_carry(regs[sA], regs[sB]);
                break;
            /* ... */
        }
    };
}
```

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 I/O ports. All these models are composed together, and BMC is used to check whether 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 [27], 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 in 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.

8.3 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 the *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. 8.2, 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 usage 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 [87]. 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. 8.2) by the top-level model discussed below.

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

Figure 8.2 The output from the Codasip semantics extractor for the `add` instruction.

```
 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]
```

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 the certain SMV operation), some are replaced by multiple SMV operations. For an illustration of the translation, see Fig. 8.3 which shows the result of translating the `add` instruction. Note, e.g., the extraction of operands from the fetch vector (lines 12-14 in Fig. 8.3) and the translation of the `carry_add` operation (line 8 in Fig. 8.2) using the operations plus and bit extraction (lines 25, 26 in Fig. 8.3).

8.4 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 one of the instruction formats (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. 8.2) 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,

Figure 8.3 Instruction semantics translated to SMV.

```
-- Variant instr__add__reg__reg__reg__
-- Definitions
reg0 : array 3..0 of boolean;
reg1 : array 3..0 of boolean;
reg2 : array 3..0 of boolean;
_tmp0 : array 15..0 of boolean;
_tmp1 : array 15..0 of boolean;
_tmp2 : array 15..0 of boolean;
_tmp3 : boolean;
_tr_tmp0 : array 16..0 of boolean;
-- Transitions
reg0[3..0] := fetch[11..8];
reg1[3..0] := fetch[7..4];
reg2[3..0] := fetch[3..0];
regs_re0 := 1;
regs_ra0 := reg1;
_tmp0 := regs_q0;
regs_re1 := 1;
regs_ra1 := reg2;
_tmp1 := regs_q1;
_tmp2 := (_tmp0 + _tmp1);
regs_we0 := 1;
regs_wa0 := reg0;
regs_d0 := _tmp2;
_tr_tmp0 := (_tmp0 + _tmp1);
_tmp3 := _tr_tmp0[16];
cf_we := 1;
cf_d := _tmp3;
```

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.

8.5 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 described in Chapter 7. 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. As it is shown in Chapter 7, the abstraction could sometimes use more bits than the actual implementation. Hence, a decision whether or not to use the abstraction is done based on the knowledge of the number of state variables that are to be used in each of the cases.

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

8.6 Data-Domain Reduction

Another technique that we use to cope with the state space explosion problem is data-domain reduction, which we apply to reduce the influence of the many different random values stored in data memories, register files, and immediate operands of the fetch vector on a rapid increase in the size of the state space. 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 or one.

We use two types of data-domain reductions each of which comes from stressing different aspects of the operations over bit-vectors: The first one concentrates on flaws in incorrectly implemented basic effects of operations (including, e.g., situations when the implementation performs a completely different operation than intended, it incorrectly loads operands from a fetch vector, and the like). The second one then concentrates on flaws in instruction side effects (e.g., in setting the carry-flag after a successful completion of an arithmetic operation). We implement the data-domain reductions by preserving high and low values of operands only—we call these reductions as *high* and *low* reductions, respectively. The high reduction transforms all bit-vectors being used as operands such that the least significant bits are set to one, while the low reduction sets the most significant bits to zero. The idea behind this is that a flaw in the implementation of the basic effect of an operation will be revealed even for small values of operands, and a flaw in the implementation of side effects will be revealed by high values of operands. The ratio of the number of random bits (i.e., those whose randomness is preserved) and reduced bits (set to zero or one) is defined by a *reduction factor*. For example, the factor of 1/4 of the low reduction means that every bit-vector which is used as an input of an instruction is transformed such that 3/4 of most significant bits of the bit-vector are set to zero and 1/4 of the least significant bits are left random.

We apply our data-domain reduction on output data from data memories, data from register files, as well as immediate operands of the fetch vector. We do not consider addresses because the bit-width of addresses has an insignificant influence on the size of the state space since we cope with it using abstracted memories. We implement the reduction technique such that all outputs of data memories and register files are masked with a predefined bit-vector representing the required data-domain reduction. When using the low reduction, the output from a memory or a register file is AND-masked with a bit-vector with zero's in the most significant bits. On the other hand, when using the high reduction, the output is OR-masked with a bit-vector with ones in the least significant bits. Similarly, the same masking is performed on each immediate operand of the fetch vector resulting in the so-called reduced fetch vector.

8.7 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 the case of a flawed RTL design, the encountered problem can be presented to the developers in terms of the original variables.

Table 8.1: Verification results.

Processor /	No. of	No. of instr.	Proved	Partially	Part. proved		Part. proved		Part. proved		Avg.
time limit	instructions	formats	no reduction	proved	1/2 reduction		1/4 reduction		1/8 reduction		time
					high	low	high	low	high	low	
SPP8 / 10 s	9	9	9	-	-	-	-	-	-	-	0.43 s
SPP16 / 10 s	11	11	11	-	-	-	-	-	-	-	0.89 s
Codea2 / 850 s	41	319	213	106	49	77	43	28	14	1	2.50 s
additive	5	73	46	27	19	22	4	5	4	-	2.51 s
multiplicative	3	54	3	51	14	39	33	12	4	-	2.60 s
logic	8	96	89	7	7	7	-	-	-	-	2.47 s
move	9	50	49	1	1	1	-	-	-	-	2.44 s
jump	7	13	13	-	-	-	-	-	-	-	2.33 s
memory	5	12	10	2	2	2	-	-	-	-	2.51 s
other	4	21	3	18	6	6	6	11	6	1	2.57 s
Codea2 / 2400 s	41	319	277	42	24	42	18	-	-	-	3.45 s
additive	5	73	73	-	-	-	-	-	-	-	2.84 s
multiplicative	3	54	12	42	24	42	18	-	-	-	6.69 s
logic	8	96	96	-	-	-	-	-	-	-	2.72 s
move	9	50	50	-	-	-	-	-	-	-	2.82 s
jump	7	13	13	-	-	-	-	-	-	-	2.71 s
memory	5	12	12	-	-	-	-	-	-	-	2.85 s
other	4	21	21	-	-	-	-	-	-	-	2.80 s

In fact, we do not generate a single BMC problem for each format of each instruction, but seven of them to be solved in parallel. These seven problems differ in the data-domain reduction used, in particular: no reduction, 1/2 low and high reductions, 1/4 low and high reductions, and 1/8 low and high reductions. 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.

8.8 Experiments

We have implemented the above described method in a prototype tool and tested it on the processors which we have described in Section 7.6 of Chapter 7. Our experiments were run on a PC with Intel Core i7-3770K @3.50GHz and 16 GB RAM using Cadence SMV (build from 05-25-11) and GlueMinisat (version 2.2.5) [101] as an external SAT solver. The results can be seen in Table 8.1. The first three columns give the processor being verified, the number of instructions in its instruction set, and the number of formats of all instructions (IFs), which gives the number of the (parallelized) BMC problems to be solved. The next columns give 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 with at least some data-reduction, and numbers of IFs successfully verified for the different concrete data-reductions. Finally, the column "Avg. time" denotes the average time needed for verification of a single instruction format.

The time limit for verification was set to 10 s for SPP8 and SPP16. For SPP16, 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 0.7 s per instruction format. The average time needed for SAT solving was 0.19 s per instruction format. Pushing the time 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: 850 s and 2400 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: 2.21 s per instruction format on average). The latter limit leaves more time for SAT solving (0.87 s in contrast of 0.29 s per instruction format on average). As can be observed, with more time dedicated to SAT solving, more instruction formats get verified with a less aggressive reduction factor. Further, one can notice that within the smaller time limit of 850 s, every instruction format was proved at least for the reduction factor of 1/8 (for a 16-bit processor, this means that 2 bits of the register file and memory were left random). Within the time limit of 1000 s (not listed in the table), each instruction format was verified at least for the 1/4 reduction. Finally, multiplication instructions (42 instruction formats) were the only ones that were too complex to be proved fully even within the extended time limit of 2400 s.

Next, to demonstrate an ability of the proposed data-domain reductions to rapidly detect errors, we also ran a series of experiments on some flawed designs of the Codea2 microprocessor. The results are shown in Table 8.2. The first column denotes the type of flaw, while the next columns provide the average time (in seconds) per instruction format needed to detect a flaw of the given type with a particular level of reduction.

Table 8.2: Detection of flaws using data-domain reductions.

Flaw type		1/2		1/4		1/8	
	none	high	low	high	low	high	low
add. cf	2.89	2.73	2.67	2.66	2.62	2.62	2.59
mult. high	3.36	3.15	-	3.03	-	2.97	-
load byte	2.91	-	2.74	-	2.30	-	2.25

The first type of flaws (named “add. cf”) represents errors that we actually found during verification of a development version of Codea2. All of them were confirmed as real errors by the processor development team and subsequently corrected. The errors were discovered in three instructions. Each of them was related to setting the carry flag during arithmetic instructions. Although one could expect that flaws related to the carry flag should be detected only when no reduction or the high reduction is used, in our case, they were detected even with the low reduction. This is due to the different ways how the verified ISA and RTL models initialize the value of the carry flag—in the RTL model, it is always nullified, while the ISA model leaves it in the previous state.

The further two types of flaws (“mult. high” and “load byte”) were artificially injected into the design. However, we tried to inject errors that are likely to appear during processor development. In the first case (“mult. high”), the most significant bits of the result of multiplication are wrongly propagated (some are set to zero). In this case, the error can be detected with the high or no reduction only. Using the high reduction is by approximately 10 % faster than in the case with no reduction. In the last case (“load byte”), a wrong `AND`-mask (`0xF` instead of `0xFF`) is applied on the value fetched from a data memory. Since the bit-mask affects the least significant bits only, the error is detected only when the low version of the data-domain reduction is used. The speed-up is comparable to the case described above. Moreover, the described speed-up can, in fact, be also seen during verification of flawless instruction formats. This produces an improvement in the overall verification time in the order of minutes for microprocessors of size comparable to Codea2, and therefore we can conclude that verification with data-domain reductions can be advantageously used to quickly scan a design for presence of errors.

8.9 Conclusion

In this chapter, 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 and for utilizing parallelization in order to increase usefulness of the results that can be obtained in the given time.

A potential future work may include adding support for designs with multiple pipelines. Another considerable topic is also an experimental evaluation of suitability of another back-end verification procedures (e.g., SMT solving instead of SAT) and representations with which these procedures work (e.g., *and-inverter graph* format [15] vs. DIMACS). Finally, one can also utilize recent advances in model-checking techniques that are not based on BMC, such as IC3/PDR [18, 45], and use them for adding better support for multi-cycle instructions.

Chapter 9

Analysis of Pipeline Hazards

In this chapter, we present an automated approach that combines static analysis of data paths, SMT solving, and formal verification of parametric systems in order to discover flaws caused by improperly handled data and control hazards. The chapter unifies and better formalizes our previous works on read-after-write [34, 35], write-after-read, and write-after-write hazards [36, 37] and also adds support to handle control hazards. The approach has been implemented in a tool called Hades using which we have obtained promising experimental results. The contents of the chapter is currently under submission to a journal.

Plan of the Chapter Section 9.1 defines the needed notions. In Section 9.2, we sketch the main idea of the proposed approach. Sections 9.3 and 9.4 discuss pre-processing tasks that are needed before the core steps of our verification approach are applied. These core steps are then described in Section 9.5. Section 9.6 presents an experimental evaluation of the proposed approach. Finally, Section 9.7 concludes the chapter.

9.1 Preliminaries

We now introduce various basic notions that we will build on in the rest of the chapter.

9.1.1 Processor Structure Graphs

In what follows, we expect a processor to be described in the form of a so-called *processor structure graph* (PSG) which can be represented by a tuple $G = (V, E, s, t, \omega)$. Here, V is a finite set that is the union $V_s \cup V_f$ of a set V_s of *storages* and a set V_f of *Boolean circuits*, $V_s \cap V_f = \emptyset$. We distinguish two types of storages: namely, *architectural storages* V_a and *pipeline registers* V_p such that $V_s = V_a \cup V_p$ and $V_a \cap V_p = \emptyset$. We expect all storages to have a unit write and zero read delay. Longer access times (e.g., for memory ports) can be modeled by introducing sequentially connected registers emulating the required delay. Boolean circuits represent common combinational logic circuits. For the rest of the chapter, it is sufficient to distinguish these circuits into *multiplexers* V_{mx} and all other circuits V_g, referred to as *generic circuits* further on. Hence, we let $V_f = V_{mx} \cup V_g$ while requiring $V_{mx} \cap V_g = \emptyset$.

For registers, we use a well-known notation to characterize their connections: namely, we use `d` to denote the data-in, `q` data-out, `rst` reset, and `en` write-enable connections. For multiplexers, we denote by `sel` the inbound connection that is the selector which selects one of the input cases $\mathtt{c}_i$ to be transferred from the input to the output of the multiplexer, which is again denoted as $\mathtt{q}$. We denote input connections of generic Boolean circuits as generic inputs $\mathtt{a}_i$. Let $\mathbb{T} = \{\mathtt{d}, \mathtt{q}, \mathtt{rst}, \mathtt{en}, \mathtt{sel}\} \cup \{\mathtt{a}_i, \mathtt{c}_i \mid i \in \mathbb{N}\}$ be the set of all connection types.

Next, we use E to denote a finite set of *transfer edges*. Note that we do not define the set of edges as $E \subseteq V \times V$ since we sometimes need more edges between two nodes. Instead, we simply require that E is a finite set of some abstract edges, and we assign each edge with its source, target, and type. Namely, we use $s : E \to V \times \mathbb{T}$ to assign to each edge its source vertex and its connection type, and $t : E \to V \times \mathbb{T}$ to assign 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:

- For each storage $v_s \in V_s$, there is exactly one inbound data-in edge $e_d \in E$ such that $t(e_d) = (v_s, \mathtt{d})$.
- For each storage $v_s \in V_s$, there are arbitrarily many outbound data-out edges $e_q^i \in E$ such that $s(e_q^i) = (v_s, \mathtt{q})$ where $0 \leq i < n$ for some $n \in \mathbb{N}$.
- For each storage $v_s \in V_s$, there is exactly one inbound clear edge $e_{rst} \in E$, also denoted as the synchronous reset edge, such that $t(e_{rst}) = (v_s, \mathtt{rst})$.
- For each storage $v_s \in V_s$, there is exactly one inbound enable edge $e_{en} \in E$ such that $t(e_{en}) = (v_s, \mathtt{en})$.
- For each circuit $v_g \in V_g$ implementing a Boolean function $g(a_0, \ldots, a_{n-1})$, there is exactly one inbound edge for each argument of g such that $t(e_{a_i}) = (v_g, \mathtt{a}_i)$ for all $0 \leq i < n$ where $n \in \mathbb{N}$. (For $n = 0$, we get a constant function without parameters.)
- Every multiplexer $v_{mx} \in V_{mx}$ that implements a case selection function $\mathit{switch}(\mathit{sel}, \mathit{case}_0, \ldots, \mathit{case}_{n-1})$ has exactly one inbound edge for each of its arguments such that $t(e_{sel}) = (v_{mx}, \mathtt{sel})$ and $t(e_{case_i}) = (v_{mx}, \mathtt{c}_i)$ for all $0 \leq i < n$ where $n \geq 2$.
- For each circuit $v_f \in V_f$, there are arbitrarily many outbound result edges $e_q^i \in E$ such that $s(e_q^i) = (v_f, \mathtt{q})$ where $0 \leq i < n$ for some $n \in \mathbb{N}^+$.
- There are no other types of edges other than the ones described above.
- There is no cycle in the graph consisting of 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 $t(e) = (v, \mathtt{c})$, $v \in V$, $\mathtt{c} \in \mathbb{T}$, can be encoded using the expression $v.\mathtt{c}$. Finally, the function $\omega \colon E \to \mathbb{N}^+$ represents a mapping that assigns some bit-width to all edges of the PSG. The mapping can be naturally expanded to be defined over storages too—namely, we let $\omega(v_s) = \omega(v_s.\mathtt{d})$ for all $v_s \in V_s$. Additionally, it must also hold that $\omega(e_{out}) = \omega(v_s.\mathtt{d})$ for any $(v_s, e_{out}) \in V_s \times \{e \in E \mid s(e) = (v_s, \mathtt{q})\}$.

Since we propose the notion of PSGs to be as simple as possible, it does not take into account *memories* and *memory ports*. Instead, it contains architectural registers, which can be used to represent particular memory cells. In the chapter, we assume that a memory is modeled using a finite number of architectural storages representing the cells of the memory. Memory ports are then modeled using additional logic circuits that select the appropriate memory cell using its address. In particular, for a memory with n addressable units, there

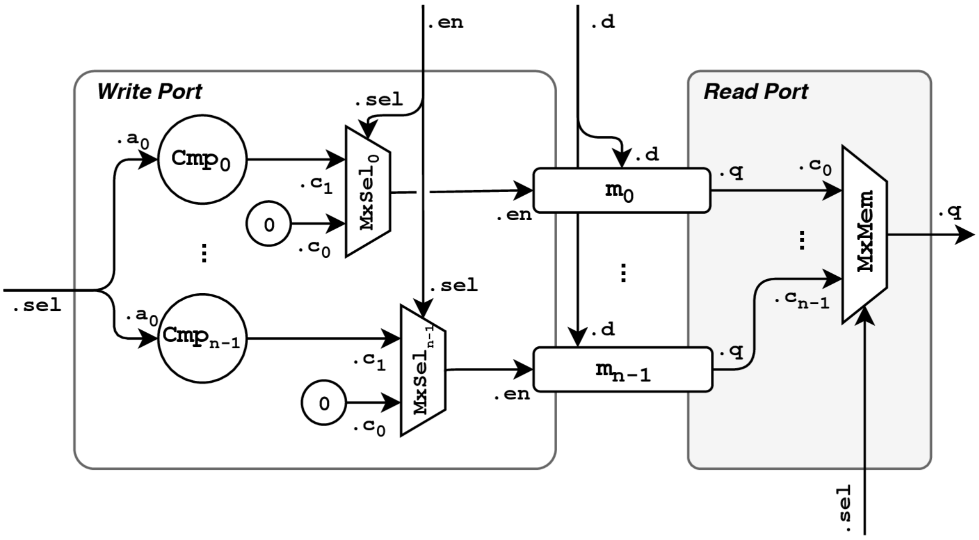


Figure 9.1: A schematic of a write and a read memory port.

are architectural registers $m_0, \ldots, m_{n-1} \in V_a$. A read memory port of such a memory is modeled using a single multiplexer circuit $v_{read} \in V_{mx}$ connected to each of the registers representing memory units—for each $m_i, 0 \leq i < n$, there is an edge $e = v_{read}.\mathtt{c}_i$ connecting a multiplexer case with the corresponding memory unit $s(e) = (m_i, \mathtt{q})$. The selector edge $v_{read}.\mathtt{sel}$ then represents a memory address and $v_{read}.\mathtt{q}$ represents the data-out connection of the memory port. A write memory port is modeled by n circuits used to enable writing to a given memory-cell $m_i, 0 \leq i < n$. Each of these circuits implements a Boolean function $(sel = i) \wedge en$, $0 \leq i < n$, where sel represents a memory port address and en enables writing to the memory. A schematic of a write and a read memory port is depicted in Fig. 9.1.

9.1.2 Transition Systems Induced by PSGs

Let $\mathbb{B} = \{0, 1\}$ be the set of Boolean values, and let $\mathbb{B}^n$ denote the set of bit-vectors of size $n \geq 1$. A PSG $G = (V, E, s, t, \omega)$ induces a (finite) *transition system* $(C, \hookrightarrow)$ where $C = \bigotimes_{v \in V_s} \mathbb{B}^{\omega(v)}$ is the set of configurations of G and $\hookrightarrow \subseteq C \times C$ is its transition relation (defined later in this section). We use $c[v_s]$ to denote the bit-vector value of the register $v_s \in V_s$ in a configuration $c \in C$. We abuse the notation and write $c[e]$ to denote the value transferred over an edge $e \in E$ in the configuration c as well. Given an edge $e \in E$ such that $s(e) = (v_f, \mathtt{q})$ where $v_f \in V_f$ is a circuit computing a function $fn(a_0, \ldots, a_{n-1})$, $n \in \mathbb{N}$, the value of $c[e]$ can be recursively expressed as $c[e] = fn(c[e_{a_0}], \ldots, c[e_{a_{n-1}}])$ where $e_{a_i} \in E$, $0 \leq i < n$, corresponds to the edge of the i-th parameter of the function fn. In the case that an edge $e \in E$ is an outbound edge of a storage $v_s \in V_s$, i.e., $s(e) = (v_s, \mathtt{q})$, we let $c[e] = c[v_s]$.

For each storage $v_s \in V_s$ of a bit-width m, $m \geq 1$, we assume the standard *next-state function* $f_{v_s}^{next} \colon \mathbb{B}^{(2 \cdot m + 2)} \to \mathbb{B}^m$ where the storage v_s is written a value transferred over the $v_s.\mathtt{d}$ edge iff the $v_s.\mathtt{rst}$ edge transfers "0" and $v_s.\mathtt{en}$ transfers "1" in the given configuration. Next, the value of the storage v_s is nullified if the $v_s.\mathtt{rst}$ edge transfers "1". In the following, we will refer to such a transition as storage *clearing*. Finally, the storage v_s keeps the same value if both $v_s.\mathtt{en}$ and $v_s.\mathtt{rst}$ transfer the value of "0". This will be referred as storage *stalling* in the following explanation. When put together, the next state function $f_{v_s}^{next}$ can be formally defined as follows:

$$f_{v_s}^{next}(curr, new, en, rst) := \begin{cases} curr & en = 0 \land rst = 0, \\ new & en = 1 \land rst = 0, \\ 0 & \text{otherwise.} \end{cases}$$

Then, the relation $\hookrightarrow$ contains a transition $c \hookrightarrow c'$ iff $c'[v_s] = f_{v_s}^{next}(c[v_s],\ c[v_s.\mathtt{d}],\ c[v_s.\mathtt{en}],\ c[v_s.\mathtt{rst}])$ for all $v_s \in V_s$.

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 $\langle v_1, e_1, v_2, \ldots, e_{k-1}, v_k \rangle$ where $v_2, ..., v_{k-1} \in V$, $e_1, ..., e_{k-1} \in E$, and every two subsequent vertices are incident with the edge listed 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 $1 \leq i, j \leq k$.

Since our approach builds on analysing conditions that hold in certain stages of the 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 edge $e \in E$ transfers some value $b \in \mathbb{B}^{\omega(e)}$. By $\mathbb{E}$, we denote the set of all such edge conditions. For each multiplexer $v_{mx} \in V_{mx}$, we define a mapping $\sigma_{v_{mx}} \colon E \to \mathbb{E}$ that captures the edge condition that must hold over the multiplexer's selector edge $v_{mx}.\mathtt{sel}$ for the data on the i-th inbound-case edge $v_{mx}.\mathtt{c}_i$ to be propagated to the multiplexer's outbound edge $v_{mx}.\mathtt{q}$. In particular,

$$\sigma_{v_{mx}}(v_{mx}.\mathtt{c}_i) := v_{mx}.\mathtt{sel} \rightsquigarrow bin_{\omega(v_{mx}.\mathtt{sel})}(i)$$

where $bin_n \colon \mathbb{Z} \to \mathbb{B}^n$ is the standard two's complement encoding of a decimal value on n bits. Further, we define a mapping $\gamma \colon \mathbb{E} \to 2^C$ that assigns each edge condition $(e \rightsquigarrow b) \in \mathbb{E}$ the set of configurations from C in which the edge e transfers the value b, i.e., $\gamma(e \rightsquigarrow b) := \{c \in C \mid c[e] = b\}$. Given a set $K \subseteq \mathbb{E}$, we also use the point-wise extension $\gamma(K) := \bigcap_{k \in K} \gamma(k)$ of γ.

9.1.3 Data and Control Hazards

Hazards in the instruction pipeline of central processing units (CPUs) are problems caused by inadequate synchronisation of earlier and later instructions running concurrently through the pipeline that may cause potential corruption of the data used by the instructions, with some result of the computation that referred to such data eventually propagated to a programmer-visible storage [108]. Three common types of hazards are data hazards, control hazards, and structural hazards. In this thesis, we will further focus on the first two types of the hazards and on CPU designs that do not use out-of-order execution. We will now give informal definitions of each of the considered hazard types, which we will later formalize in Section 9.4.

Definition 9. A *read-after-write* (RAW) data hazard is a scenario in which a later-started instruction uses data supposed to be produced by an earlier-started instruction, but the earlier instruction has not yet managed to proceed far enough in the pipeline to write the data into the storage used by the later instruction. The later instruction then stores a potentially wrong result of its execution, obtained by dealing with obsolete data, into some programmer-visible storage.

Definition 10. A *write-after-read* (WAR) data hazard is a scenario in which some data that should be used by an earlier-started instruction are overwritten by a later-started instruction before the earlier instruction manages to read the data. The earlier instruction then stores a potentially wrong result of its execution, obtained by dealing with data seemingly coming from the future, into some programmer-visible storage.

Definition 11. A *write-after-write* (WAW) data hazard is a scenario in which an earlier-started instruction overwrites the result of a later-started instruction that is stored in some programmer-visible storage, which then ends up containing obsolete data.

Definition 12. A *control* (CTL) hazard is a scenario where an earlier-started control-flow instruction changes the flow of the control, but some later, speculatively-started instruction manages to store some data into a programmer-visible storage.

In in-order execution designs, the above specified hazards are eliminated by *pipeline stalling* and/or *operand forwarding*. For pipeline stalling, it is necessary for a processor to be equipped with a control logic that determines whether a hazard could/will occur. If such a situation is detected, the control logic inserts no-operation (`NOP`) instruction, sometimes called *bubble*, into the pipeline. Therefore, before the later instruction from the pair of instructions which would cause the hazard executes, the earlier one will have sufficient time to proceed far enough in the pipeline so that the hazard does not happen.

In the case of operand forwarding, additional (redundant) data-paths are introduced into a processor design. These data-paths are aimed to provide an option to propagate partially computed data[1] from an earlier instruction to a later one in order to minimize the number of `NOP` instructions that would otherwise have to be inserted using the above mentioned stalling technique.

9.2 The Proposed Approach to Hazard Detection

Our approach for verifying that the pipeline logic prevents hazards consists of the following steps: (i) a simple data-flow analysis intended to distinguish particular stages of the pipeline, (ii) a consistency check to make sure that the flow logic guarantees an in-order execution of instructions through the identified pipeline stages, (iii) a static analysis deriving constraints over data-paths of instructions that can potentially cause a pipeline hazard, (iv) generation of a parametric system modelling mutual interactions between potentially conflicting instructions allowed by the derived constraints, and (v) an analysis of the constructed parametric system to see whether the identified interactions may lead to a hazard.

We assume the processor under verification to be represented using a PSG, which can be easily obtained from a description of the processor on the register transfer level (RTL) written in common hardware description languages, such as VHDL or Verilog.

Example 1. Throughout the following sections, we will be illustrating the different steps of our approach on a running example depicted in Fig. 9.2. The figure shows a PSG describing a part of a simple microprocessor with an accumulator architecture with the following architectural storages: X (a memory index register), A (an accumulator), PC (the program counter), $Prog_i$ (program memory cells), and Mem_j (data memory cells) where $0 \leq i \leq \ell$, $0 \leq j \leq k$ and k, resp. ℓ, are the sizes of the memories. The depicted part of the CPU is used when executing arithmetic and load/store instructions. In order

[1] The data that have not been written to its final storage.

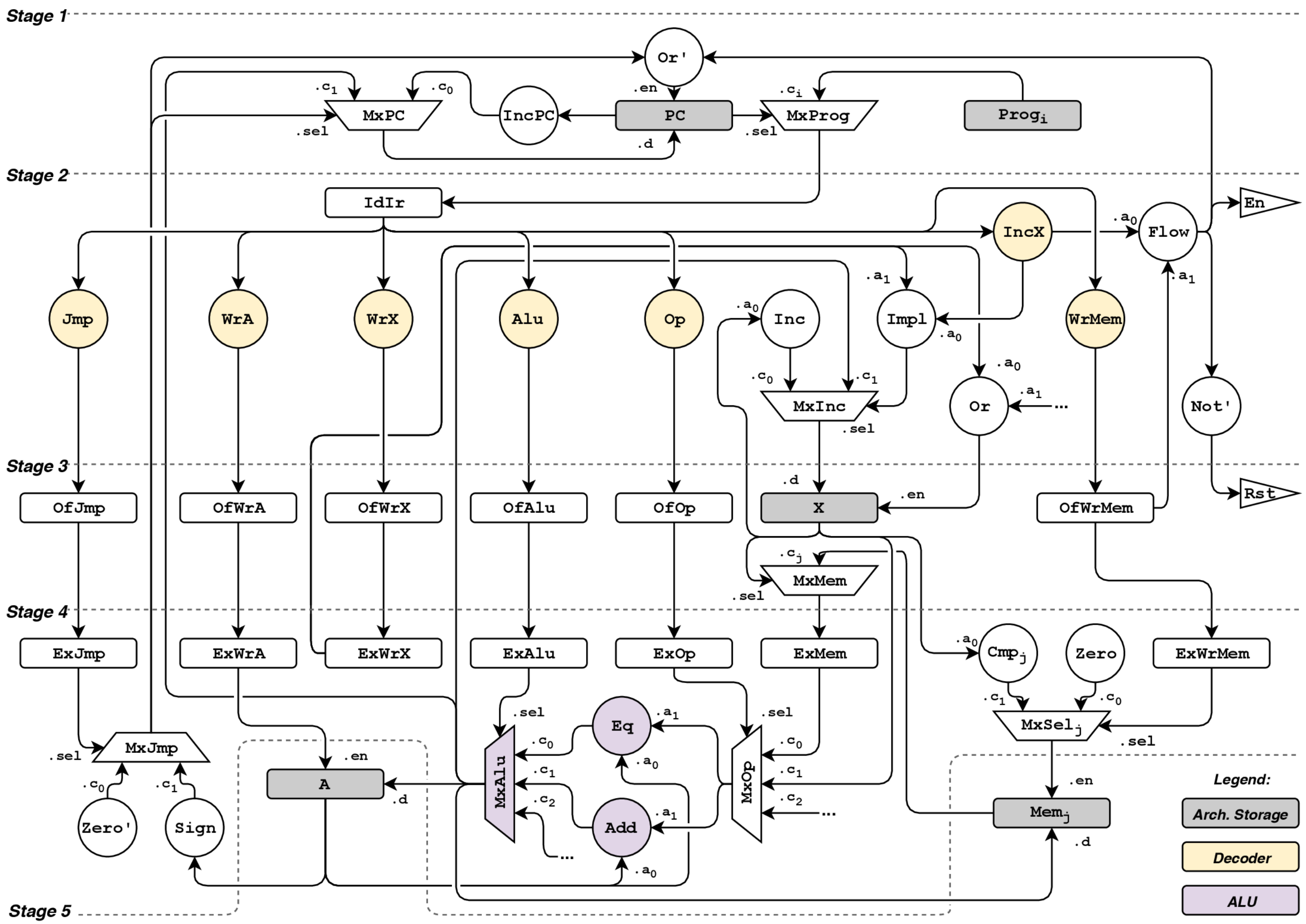


Figure 9.2: A processor structure graph of a part of a CPU with an accumulator architecture.

to keep the PSG easily readable, types of connections are shown for architectural storages and case-`c` edges of multiplexers only. Also, since enable (i.e., "`en`") and clear (i.e., "`rst`") connections for pipeline registers[2] are common for each stage, they are left out up to the ones that are required in the further explanation.

In the CPU, the computation starts in Stage 1 by using the content of the program counter PC to address the ith cell of the program memory $Prog_i$. An instruction fetched from the program memory cell is stored into the storage $IdIr$ that represents the so-called fetch register. The fetched instruction word in $IdIr$ is then decoded by an instruction decoder in Stage 2. Boolean circuits that belong to the decoder are shown in yellow. Next, an address stored in the index register is used to fetch data from the jth cell of the data memory Mem_j in Stage 3. Optionally, the index register can be *auto-incremented*. The auto-incrementation logic is a feature allowing for an early incrementation of the value of a register for memory addressing just before or right after it is read. We then speak about the so-called pre-/post-increment, respectively. The auto-incrementation feature usually brings a more efficient execution of sequences of instructions accessing the processor's memory (for instance, when computing over long arrays or other juxtaposed data). This speed up results from removing a need of otherwise required pipeline stalls, but it also introduces potential WAW and WAR hazards that must be handled properly. Finally, in Stage 4, the decoded opcode part of the instruction is used to determine the type of an ALU operation (with the ALU itself colored in purple) and to select destination storages by setting their enable connection "`en`" to logical "1".

The Boolean circuit *Flow* in Fig. 9.2 represents the flow logic of the second pipeline stage. This logic is responsible for dealing with WAR hazards on the index register X. The flow logic implements the function

$$Flow(IncX, OfWrMem) := \neg IncX \lor \neg OfWrMem.$$

In case a later instruction wants to perform an auto-increment of the index register X while an earlier instruction is going to use the content of X for a memory write, the flow logic uses the enable "`en`" and clear "`rst`" signals of pipeline registers to insert a pipeline bubble between the instructions into Stage 3. ◁

9.3 Preprocessing a Processor Structure Graph

This section describes the first two steps of the proposed approach: namely, the data-flow analysis identifying pipeline stages and the pipeline consistency check ensuring a proper in-order execution of instructions within the pipeline.

9.3.1 Data-Flow Analysis Discovering Pipeline Stages

The input of the proposed verification method consists of a PSG and a list of its architectural registers, including the program counter. On this input, the method starts by a simple data flow analysis whose goal is to compute the number of pipeline stages. We then map storages, logic functions, and edges of the PSG into the pipeline stages. We define a *pipeline stage* as the sub-graph of the PSG that is responsible for executing a single-cycle step of an instruction. The pipeline stage that an edge or a vertex (representing a storage or circuit) of a PSG belongs to is given by the minimum number of cycles needed to propagate data

[2] For a full list of pipeline registers, see Table 9.1 in Section 9.3.1.

Table 9.1: Storages of the CPU from Fig. 9.2 and the corresponding pipeline stages.

Storage	Stage	Write stages	Read stages	Pivot
	φ	φ^{wr}	φ^{rd}	
PC	1	$\{1,2,3,4\}$	$\{1,2\}$	–
$Prog_i$	1	$\emptyset$	$\{2\}$	–
X	3	$\{2,3,4\}$	$\{3,4,5\}$	–
A	5	$\{4\}$	$\{1,2,3,4,5\}$	–
Mem_j	5	$\{4\}$	$\{4\}$	–
$IdIr$	2	$\{1,2,3,4\}$	$\{1,2,3\}$	✓
$OfJmp$	3	$\{2,3,4\}$	$\{1,2,3,4\}$	✓
$OfWrA$	3	$\{2,3,4\}$	$\{4\}$	×
$OfWrX$	3	$\{2,3,4\}$	$\{1,2,3,4\}$	✓
$OfAlu$	3	$\{2,3,4\}$	$\{1,2,3,4\}$	✓
$OfOp$	3	$\{2,3,4\}$	$\{1,2,3,4\}$	✓
$OfWrMem$	3	$\{2,3,4\}$	$\{1,2,3,4\}$	✓
$ExJmp$	4	$\{3,4\}$	$\{1,2,3,4\}$	✓
$ExWrA$	4	$\{3,4\}$	$\{5\}$	×
$ExWrX$	4	$\{3,4\}$	$\{1,2,3\}$	✓
$ExAlu$	4	$\{3,4\}$	$\{1,2,3,5\}$	✓
$ExOp$	4	$\{3,4\}$	$\{1,2,3,5\}$	✓
$ExMem$	4	$\{3,4\}$	$\{3,5\}$	✓
$ExWrMem$	4	$\{3,4\}$	$\{1,2,3,5\}$	✓

from the input of the program counter to the edge or the output of the given vertex, respectively. Hence, as a particular case, the program counter itself belongs to Stage 1.

The 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. If several stage values are propagated to a single vertex or edge, the minimum is taken. Whenever a propagated stage value passes a storage, it is incremented by one. This analysis gives us a mapping $\varphi\colon V \cup E \to \mathbb{S}$, $\mathbb{S} = \{1, \dots, n\}$, $n \geq 1$, which maps graph's vertices and edges to pipeline stages.

Subsequently, we derive the so-called *write stage* mapping $\varphi^{\mathrm{wr}}\colon V \cup E \to 2^{\mathbb{S}}$ that maps each vertex or edge to the set of stages that directly influence its value. Namely, we include into $\varphi^{\mathrm{wr}}(x)$ the stage of every pipeline storage $v_p \in V_p$ from which there is a path to x that does not pass through any further storage from V_p. Likewise, we derive the *read stage* mapping $\varphi^{\mathrm{rd}}\colon V \cup E \to 2^{\mathbb{S}}$ for each vertex or edge that describes which stages are directly influenced by its value. In particular, we include into $\varphi^{\mathrm{rd}}(x)$ the stage of every pipeline storage $v_p \in V_p$ to which there is a path from x that does not pass through any other storage from V_p.

Pipeline stages of the storages from the PSG of Fig. 9.2 and the corresponding read and write stages, computed as described above, are shown in Table 9.1. (The notion of pivots will be introduced later on.)

9.3.2 Pipeline Consistency Checking

The second step of our approach 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. To check the above, we verify whether the flow logic obeys a set of rules which express how the control connections ($\texttt{en}, \texttt{rst}$) of storages in adjacent pipeline stages should be set. In particular, we use a strengthened variant of the rules proposed in [98]. The rules have been strengthened since (as we will see later on) our approach builds on an assumption that, if some pipeline stage is stalled, then all predecessor stages have to be stalled as well. This means that our approach rules out some extreme ways of pipeline implementation allowed by the original rules. An example of such a situation is an optimization of the execution during stage stalling when an instruction preceded by a series of `NOP` instructions is allowed to proceed to the next stage in order to increase the throughput.

For the following, assume a transition system $(C, \hookrightarrow)$ induced by the PSG being verified. We introduce mappings st, $rst\colon V_p \to 2^C$ defined as

$$\begin{aligned} st(v_p) &:= \gamma(\{v_p.\texttt{en} \rightsquigarrow 0, v_p.\texttt{rst} \rightsquigarrow 0\}), \\ rst(v_p) &:= \gamma(v_p.\texttt{rst} \rightsquigarrow 1). \end{aligned}$$

Intuitively, for any storage $v_p \in V_p$, $st(v_p)$ and $rst(v_p)$ are the sets of configurations in which v_p is stalled or cleared, respectively. The pipeline consistency rules that we check are then the following:

- *Rule 1*: If some pipeline register of a stage $s \in \mathbb{S}$ is stalled, then all pipeline storages of the Stage s have to be stalled, i.e., for all $v_p, v_p' \in V_p$:

  $$\varphi(v_p) = \varphi(v_p') \Rightarrow st(v_p) \subseteq st(v_p').$$

  The rule follows the idea that an instruction carried by a pipeline stage cannot be fragmented. The rule also reflects one of the fundamental assumptions about pipelined execution from [98]: namely, at any given time, an instruction is always in a single pipeline stage only. As a corollary, by simply swapping v_p and v_p', one can derive a stronger statement $\varphi(v_p) = \varphi(v_p') \Rightarrow st(v_p) = st(v_p')$.

- *Rule 2*: If some pipeline register in a Stage $s \in \mathbb{S} \setminus \{\max(\mathbb{S})\}$ is stalled, then all pipeline storages of the Stage $s+1$ have to be stalled or cleared, i.e., for all $v_p, v_p' \in V_p$:

  $$\varphi(v_p) = \varphi(v_p') - 1 \Rightarrow st(v_p) \subseteq st(v_p') \cup rst(v_p').$$

  This rule is a rephrased version of Equation (15) from [98] and prevents duplication of an instruction.

- *Rule 3*: If some pipeline register in a Stage $s \in \mathbb{S} \setminus \{1\}$ is stalled, then all pipeline storages of the Stage $s-1$ have to be stalled, i.e., for all $v_p, v_p' \in V_p$:

  $$\varphi(v_p) = \varphi(v_p') + 1 \Rightarrow st(v_p) \subseteq st(v_p').$$

This rule is a rephrased version of Equation 16 from [98] and prevents an instruction to be lost.

- *Rule 4*: If some pipeline register in a Stage $s \in \mathbb{S}$ is cleared, then all pipeline storages of the Stage s have to be cleared, i.e., for all $v_p, v'_p \in V_p$:
$$\varphi(v_p) = \varphi(v'_p) \Rightarrow rst(v_p) \subseteq rst(v'_p).$$
Similarly to Rule 1, this rule prevents fragmentation of an instruction and it is a part of the basic assumptions about pipelined execution mentioned in [98].

We check the above rules using an SMT solver [21, 100] for the bit-vector logic. To convert the rules into the bit-vector logic, we first define an operator * that maps edges of a PSG to variables of the bit-vector logic (BVL) such that $e_1^\star = e_2^\star \Leftrightarrow s(e_1) = s(e_2)$ for each $e_1, e_2 \in E$. Intuitively, edges with the same source must have the same value. Then, for any $e \in E$, we define a BVL formula $\psi(e)$ that encodes how the value transmitted over e is computed from values stored in storages. The formula $\psi(e)$ is recursively defined as

$$\psi(e) := \begin{cases} e^\star = g(e_1^\star, ..., e_m^\star) \wedge \bigwedge_{i=1}^{m} \psi(e_i) & s(e) = (v, \mathtt{q}) \wedge v \in V_f, \\ true & \text{otherwise} \end{cases}$$

where g denotes the Boolean function computed by the circuit $v \in V_f$.

Now, the inclusion test $st(v_p) \subseteq st(v'_p)$ from Rule 1 can be reduced to checking validity of the following formula:

$$\begin{aligned} \Phi(v_p) := (\ &(\psi(v_p.\mathtt{en}) \wedge \psi(v_p.\mathtt{rst}) \wedge \psi(v'_p.\mathtt{en}) \wedge \psi(v'_p.\mathtt{rst})) \\ &\Rightarrow (\ (v_p.\mathtt{en}^\star = 0 \wedge v_p.\mathtt{rst}^\star = 0) \Rightarrow \\ &\qquad (v'_p.\mathtt{en}^\star = 0 \wedge v'_p.\mathtt{rst}^\star = 0)\ )\ ). \end{aligned}$$

Intuitively, $\Phi(v_p)$ says that if the values of $v_p.\mathtt{en}$, $v_p.\mathtt{rst}$, $v'_p.\mathtt{en}$, and $v'_p.\mathtt{rst}$ are computed according to the given flow logic, then if v_p is stalled, v'_p is stalled too. Instead of checking validity of $\Phi(v_p)$, one can check unsatisfiability of the negation of the formula, i.e., $\neg sat(\neg\Phi(v_p))$. Moreover, as $\neg\Phi(v_p) = \psi(v_p.\mathtt{en}) \wedge \psi(v_p.\mathtt{rst}) \wedge \psi(v'_p.\mathtt{en}) \wedge \psi(v'_p.\mathtt{rst}) \wedge v_p.\mathtt{en}^\star = 0 \wedge v_p.\mathtt{rst}^\star = 0 \wedge (v'_p.\mathtt{en}^\star = 1 \vee v'_p.\mathtt{rst}^\star = 1)$, the check $\neg sat(\neg\Phi(v_p))$ can be replaced by the following two simpler checks:[3]

$$\neg sat \begin{pmatrix} \psi(v_p.\mathtt{en}) & \wedge & v_p.\mathtt{en}^\star & = 0 & \wedge \\ \psi(v_p.\mathtt{rst}) & \wedge & v_p.\mathtt{rst}^\star & = 0 & \wedge \\ \psi(v'_p.\mathtt{en}) & \wedge & v'_p.\mathtt{en}^\star & = 1 & \end{pmatrix} \tag{9.1}$$

$$\neg sat \begin{pmatrix} \psi(v_p.\mathtt{en}) & \wedge & v_p.\mathtt{en}^\star & = 0 & \wedge \\ \psi(v_p.\mathtt{rst}) & \wedge & v_p.\mathtt{rst}^\star & = 0 & \wedge \\ \psi(v'_p.\mathtt{rst}) & \wedge & v'_p.\mathtt{rst}^\star & = 1 & \end{pmatrix} \tag{9.2}$$

Hence, Rule 1 can be checked by applying the checks from Equations 9.1 and 9.2 to all $v_p, v'_p \in V_p$ such that $\varphi(v_p) = \varphi(v'_p)$.

Rules 2–4 can be checked in a very similar way as Rule 1.

[3] Note that, in Equation 9.1, we may remove the $\psi(v'_p.\mathtt{rst})$ conjunct since the constraint $v'_p.\mathtt{rst}^\star = 1$ is not present, and likewise with $\psi(v'_p.\mathtt{en})$ in Equation 9.2.

9.4 Static Detection of Potential Pipeline Hazards

According to Definitions 9–12, a pipeline hazard (of any of the discussed kinds) occurs when two instructions access the same architectural storage and at least one of the accesses is a write. We will further use the term *spoiler* whenever referring to the writing instruction causing the hazard. The other involved instruction will then be called a *victim* instruction. Finally, we will speak about a *hazard case* when referring to the pair formed by a spoiler and a victim instruction.

In this section, we will first focus on identifying a finite set of hazard cases potentially causing hazards in a given processor. For that, we will use a static hazard analysis examining the PSG and pipeline stage mappings φ, φ^{wr}, φ^{rd} determined by the data-flow analysis from Section 9.3.1. In order to be able to describe a spoiler-victim pair forming a hazard case, we will introduce several auxiliary concepts with the so-called *minimal transfer execution* and *maximal store execution* being the most important ones.

We begin by introducing a notion representing a generic concept of a data transfer between two vertices within a given PSG. Naturally, each such transfer must conform to the φ^{wr} and φ^{rd} mappings. We first formalize the notion of data transfers in a broader form in Definition 13, which is narrowed later on in Definition 14. In particular, Definition 13 is broader in the sense that it may describe data transfers that can only be achieved when multiple instructions are involved and some of the instructions pass the data back to lower stages of the pipeline where they are processed by instruction(s) that entered the pipeline later. This would mean that a spoiler itself (and likewise a victim) could consist of multiple instructions. Dealing with such situations is, of course, interesting, but we will restrict ourselves to the case of the spoiler and victim being single instructions each, generating the so-called forward executions (Definition 14).

Definition 13. Given a walk $\pi = \langle p_1, p_2, \ldots, p_k \rangle$ for some $k \geq 3$ in a PSG G, p_1, $p_3, \ldots, p_k \in V$, p_2, $p_4, \ldots, p_{k-1} \in E$, an *execution plan* is any valuation $\tau\colon \{1, \ldots, k\} \to \mathbb{S}$ s.t. $p_i \in V_s \Rightarrow \tau(i) - 1 \in \varphi^{\mathrm{wr}}(p_i)$ for all $1 < i \leq k$.

Intuitively, an execution plan gives a sequence of stages in which particular vertices are written during a data transfer. Hence, taking into account the unit delay of writing, the value written to a vertex p_i is obtained from a value computed in the stage $\tau(i) - 1$ (with the first element of the walk being, of course, special and excluded from this requirement). An *execution walk* is then any walk in G with an execution plan. We define an *execution* as a pair (π, τ) consisting of an execution walk π and an execution plan τ. We denote the set of all such pairs as $\mathbb{X}$. In the following explanation, we will also use shortcuts τ^{fst} and τ^{lst} in order to refer to the valuation of the first and last element of the walk π, respectively, i.e., $\tau^{\mathrm{fst}} = \tau(1)$ and $\tau^{\mathrm{lst}} = \tau(k)$.

Example 2. Consider the PSG G depicted in Fig. 9.2. A pair (π_1, τ_1) s.t. $\pi_1 = \langle X$, $\mathit{MxMem}.\mathtt{sel}$, MxMem, $\mathit{ExMem}.\mathtt{d}$, ExMem, $\mathit{MxOp}.\mathtt{c}_0$, MxOp, $\mathit{Eq}.\mathtt{a}_0$, Eq, $\mathit{MxAlu}.\mathtt{c}_0$, MxAlu, $A.\mathtt{d}$, $A\rangle$ and $\tau_1 = \{1^{X} \mapsto 3$, $2^{\mathit{MxMem}.\mathtt{sel}} \mapsto 3$, $3^{\mathit{MxMem}} \mapsto 3$, $4^{\mathit{ExMem}.\mathtt{d}} \mapsto 3$, $5^{\mathit{ExMem}} \mapsto 4$, $6^{\mathit{MxOp}.\mathtt{c}_0} \mapsto 4$, $7^{\mathit{MxOp}} \mapsto 4$, $8^{\mathit{Eq}.\mathtt{a}_0} \mapsto 4$, $9^{\mathit{Eq}} \mapsto 4$, $10^{\mathit{MxAlu}.\mathtt{c}_0} \mapsto 4$, $11^{\mathit{MxAlu}} \mapsto 4$, $12^{A.\mathtt{d}} \mapsto 4$, $13^{A} \mapsto 5\}$ is an execution in G describing one of the possible data transfers from the storage X to the storage A. Note that we indexed the left-hand sides of the mappings by the corresponding storages to make the mappings more readable.

Another example of an execution is a pair (π_2, τ_2) where $\pi_2 = \langle \mathit{ExJmp}$, $\mathit{MxJmp}.\mathtt{sel}$, MxJmp, $\mathit{MxPC}.\mathtt{sel}$, MxPC, $\mathit{PC}.\mathtt{d}$, PC, $\mathit{MxProg}.\mathtt{sel}$, MxProg, $\mathit{IdIr}.\mathtt{d}$, $\mathit{IdIr}\rangle$ and $\tau_2 =$

$\{1^{ExJmp} \mapsto 4,\ 2^{MxJmp.\mathtt{sel}} \mapsto 4,\ 3^{MxJmp} \mapsto 4,\ 4^{MxPC.\mathtt{sel}} \mapsto 4,\ 5^{MxPC} \mapsto 4,\ 6^{PC.\mathtt{d}} \mapsto 4,\ 7^{PC} \mapsto 5,\ 8^{MxProg.\mathtt{sel}} \mapsto 1,\ 9^{MxProg} \mapsto 1,\ 10^{IdIr.\mathtt{d}} \mapsto 1,\ 11^{IdIr} \mapsto 2\}$. ◁

To narrow our selection only to executions that are feasible by a single instruction, one needs to only think of executions tied with execution plans where stages form a non-decreasing sequence. Intuitively, a single instruction in the pipeline can only move forward or stay in the same stage. This leads us to the definition given next.

Definition 14. A *forward execution* is a special type of execution $(\langle p_1, p_2, \ldots, p_k\rangle, \tau) \in \mathbb{X}$, $k \geq 3$, where the following restrictions hold: (i) $p_i \in V_s \Rightarrow \tau(p_i) = \tau(p_{i-1}) + 1$ and (ii) $p_i \in V_f \cup E \Rightarrow \tau(p_i) = \tau(p_{i-1})$ for all $1 < i \leq k$.

Clearly, if any of the conditions (i) or (ii) is not met, there could not be any single instruction capable of a data transfer described by the execution.

Example 3. Consider the executions from Example 2. The execution (π_1, τ_1) is a forward execution while (π_2, τ_2) is not since $\tau_2(8^{MxProg.\mathtt{sel}}) \neq \tau_2(7^{PC})$. ◁

For further explanation, it is important to be able to identify a storage from which the transferred data can be passed to another (later) instruction. Such an action occurs only if there exists a path leading from a storage in a higher stage to a storage that belongs to a lower one. This is formalized in the next definition.

Definition 15. A pipeline storage $v \in V_p$ is a *pivot* if there exist a stage $s_r \in \varphi^{\mathrm{rd}}(v)$ s.t. $s_r \leq \varphi(v)$.

We also need to establish a notion of a stage that can be cleared without the previous stage being stalled. Such a stage can be used to nullify the state of a partially executed instruction.

Definition 16. A stage $s \in \mathbb{S}$ is *independently clearable* if there exist pipeline storages $v_p, v_p' \in V_p$ s.t. $\varphi(v_p) = s = \varphi(v_p') + 1$ and $\mathit{rst}(v_p) \cap \overline{\mathit{st}(v_p')} \neq \emptyset$ where *st* and *rst* are the mappings defined in Section 9.3.2.

We decide whether a stage satisfies the above given constrains for being independently clearable in a similar way to Rules 1–4. More precisely, an SMT solver performs the following check in this case:

$$\mathit{sat}\begin{pmatrix} \psi(v_p.\mathtt{rst}) & \wedge & \psi(v_p'.\mathtt{en}) & \wedge & \psi(v_p'.\mathtt{rst}) \\ v_p.\mathtt{rst}^\star = 1 & \wedge & (v_p'.\mathtt{en}^\star = 1 & \vee & v_p'.\mathtt{rst}^\star = 1) \end{pmatrix} \tag{9.3}$$

The above check can be further decomposed into two simpler checks while it suffices that at least one is satisfiable:

$$\mathit{sat}\begin{pmatrix} \psi(v_p.\mathtt{rst}) & \wedge & v_p.\mathtt{rst}^\star & = 1 \\ \psi(v_p'.\mathtt{en}) & \wedge & v_p'.\mathtt{en}^\star & = 1 \end{pmatrix} \tag{9.4}$$

$$\mathit{sat}\begin{pmatrix} \psi(v_p.\mathtt{rst}) & \wedge & v_p.\mathtt{rst}^\star & = 1 \\ \psi(v_p'.\mathtt{rst}) & \wedge & v_p'.\mathtt{rst}^\star & = 1 \end{pmatrix} \tag{9.5}$$

In the next step, we define an execution that can be performed by a single instruction and which *may* influence the value stored in some storage.

Definition 17. A *store execution* is a forward execution $(\langle v_1, e_1, \dots, e_{k-1}, v_k\rangle, \tau)$ for some $k > 0$, $v_k \in V_s$ so that $v_2, \dots, v_{k-1} \notin V_s$. We also define a *maximal store execution* as a store execution that is not a suffix of any other store execution.

As a final step, we define an execution that can be performed by a single instruction and which *may* influence the data stored in an architectural storage $v_a \in V_a$ by reading some data from a (potentially different) storage $v \in V_s$ and transferring them to the storage v_a.

Definition 18. A *transfer execution* is a forward execution $(\langle v_1, e_1, \dots, e_{k-1}, v_k\rangle, \tau)$ for some $k > 0$, $v_k \in V_s$ that satisfies the following two properties: (i) The storage v_k satisfies one of the following: (a) it is an architectural storage $v_k \in V_a$, (b) it is a pipeline storage $v_k \in V_p$ s.t. $t(e_{k-1}) = (v_k, \texttt{rst})$ and $\varphi(v_k)$ is an independently clearable stage, or (c) the storage $v_k \in V_p$ is a pivot s.t. $t(e_{k-1}) = (v_k, \texttt{d})$. (ii) Moreover, $t(e_i) \notin V_p \times \{\texttt{en}, \texttt{rst}\}$ for all $1 \leq i < k$. We also define a *minimal transfer execution* as a transfer execution that does not contain any prefix that is a transfer execution.

Condition (i-a) is straightforward as the execution affects the architectural storage directly in this case. Clearing the target pipeline register $v_k \in V_p$ in an independently clearable stage as described in Condition (i-b) causes cancellation of any partially executed instruction in Stage $\varphi(v_k)$. Such an event may indirectly influence any architectural storage $v_a \in V_a$ that belongs to a stage $s \geq \varphi(v_k)$. Similarly, concerning Condition (i-c), if the target pipeline register $v_k \in V_p$ is a pivot, the value read from it—by a later instruction—may also indirectly influence any architectural storage that the later instruction writes to. Next, as described by Condition (ii), the transfer execution must not traverse through enable connections of pipeline registers. Such executions cannot influence the value of any architectural storage. Their only impact can be that they stall a stage. This also holds for reset connections of pipeline storages in a stage that is not independently clearable—in this case, an instruction cannot be lost since the previous stage is always stalled. In such a case, the pipeline consistency given by Rules 1–4 from Section 9.3.2 assures correct preservation of all partially executed instructions.

An incorrectly handled pipeline hazard manifests upon the first write of improper data into some architectural storage of the design. Therefore, it suffices to further deal with the minimal transfer executions only. We can now formalize the notion of hazard cases in a unified way for all the different kinds of hazards (restricted to the case when the spoiler and victim consist of single instructions) as follows. In particular, we represent a hazard case as a tuple $(\chi_{sp}, \chi_{vi}) \in \mathbb{X}^2$ where χ_{sp} and χ_{vi} are spoiler and victim executions appropriate for the concerned kind of hazard. More rigorous descriptions of each considered type of hazard cases are given in the following definitions.

Definition 19. A *RAW hazard case* is a tuple $(\chi_{sp}, \chi_{vi}) \in \mathbb{X}^2$ consisting of a maximal store execution $\chi_{sp} = (\langle v_1^{sp}, e_1^{sp}, \dots, e_{k-1} = v_{k-1}^{sp}.\texttt{d}, v_k^{sp} = v\rangle, \tau_{sp})$, $v_1^{sp} \in V_s$, of a spoiler instruction and a minimal transfer execution $\chi_{vi} = (\langle v_1^{vi} = v, e_1^{vi}, \dots, v_\ell^{vi}\rangle, \tau_{vi})$ of a victim instruction where $v \in V_a \setminus \{v_{pc}\}$, $k, \ell > 1$, and data in the architectural storage v can be read by the victim instruction before they are written by the spoiler, i.e., $\tau_{vi}^{\text{fst}} < \tau_{sp}^{\text{lst}}$.

Definition 20. A *WAR hazard case* is a tuple $(\chi_{sp}, \chi_{vi}) \in \mathbb{X}^2$ consisting of a maximal store execution $\chi_{sp} = (\langle v_1^{sp}, e_1^{sp}, \dots, e_{k-1} = v_{k-1}^{sp}.\texttt{d}, v_k^{sp} = v\rangle, \tau_{sp})$, $v_1^{sp} \in V_s$, of a spoiler instruction and a minimal transfer execution $\chi_{vi} = (\langle v_1^{vi} = v, e_1^{vi}, \dots, v_\ell^{vi}\rangle, \tau_{vi})$ of a victim instruction where $v \in V_a \setminus \{v_{pc}\}$, $k, \ell > 1$, and data in the architectural storage v can be written by the spoiler before they are read by the victim, i.e., $\tau_{sp}^{\text{lst}} < \tau_{vi}^{\text{fst}}$.

Definition 21. A *WAW hazard case* is a tuple $(\chi_{sp}, \chi_{vi}) \in \mathbb{X}^2$ consisting of a maximal store execution $\chi_{sp} = (\langle v_1^{sp}, e_1^{sp}, \ldots, e_{k-1}^{sp} = v.\mathtt{d}, v_k^{sp} = v\rangle, \tau_{sp})$, $v_1^{sp} \in V_s$, of a spoiler instruction and a maximal store execution $\chi_{vi} = (\langle v_1^{vi}, e_1^{vi}, \ldots, e_{\ell-1}^{vi} = v.\mathtt{d}, v_\ell^{vi} = v\rangle, \tau_{vi})$, $v_1^{vi} \in V_s$, of a victim instruction where $v \in V_a \setminus \{v_{pc}\}$, $k, \ell > 1$, and data into the architectural storage v can be written from two different stages. In the following, without a loss of generality (since the conflicting instructions can always be swapped), we will assume the spoiler to perform a write operation in an earlier stage, i.e., $\tau_{sp}^{\text{lst}} < \tau_{vi}^{\text{lst}}$.

One can observe that there is no need to include any minimal transfer execution in the case of WAW hazard since an error that is caused by the hazard is manifested instantly by writing an incorrect value to the storage v.

Definition 22. A *CTL hazard case* is a tuple $(\chi_{sp}, \chi_{vi}) \in \mathbb{X}^2$ consisting of a maximal store execution $\chi_{sp} = (\langle v_1^{sp}, e_1^{sp}, \ldots, e_{k-1} = v_{k-1}^{sp}.\mathtt{d}, v_k^{sp} = v_{pc}\rangle, \tau_{sp})$, $v_1^{sp} \in V_s$, of a spoiler instruction and a minimal transfer execution $\chi_{vi} = (\langle v_1^{vi} = v_{pc}, e_1^{vi}, \ldots, v_\ell^{vi}\rangle, \tau_{vi})$ of a victim instruction where $k, \ell > 1$, $v_{pc} \neq v_\ell^{vi}$ and the program counter $v_{pc} \in V_a$ is written with data originating from a source other than auto-increment logic, which we consider to appear in Stage 1. Therefore, the spoiler must always write from a stage other than the first one, i.e., $\tau_{sp}^{\text{lst}} > 2$.

Note that, since the definition of a particular hazard case speaks about storages, their access stages, and the path along which the problematic data are transferred, it is not defined for a single concrete instruction only but for an entire class of instructions that conform to the criteria given by the hazard case. Further, note that the cases when $\tau_{sp}^{\text{lst}} = \tau_{vi}^{\text{fst}}$ for RAW, WAR, and CTL hazards as well as the cases when $\tau_{sp}^{\text{lst}} = \tau_{vi}^{\text{lst}}$ for WAW hazards are not covered by the above definitions. This is because our approach assumes correct execution of isolated instructions, which rules such cases out. Such correctness can be checked separately using, e.g., methods described in [27, 31].

In order to generate the set $\mathbb{H}$ of hazard cases, we proceed as follows. First, using results of the data-flow analysis from Section 9.3.1, we find all storages $v_a \in V_a$ for which there is a risk that some hazard situation may be initiated between stages $s_1, s_2 \in \mathbb{S}$. The conditions that must hold for s_1, s_2 differ for different hazard cases. For instance, for RAW hazards, we need the following conditions to hold: $s_1 - 1 \in \varphi^{\text{wr}}(v_a)$, $s_2 + 1 \in \varphi^{\text{rd}}(v_a)$, and $s_2 < s_1$. The condition $s_2 < s_1$ reflects the fact that the needed data are read from v_a before they are written into v_a. The rest of the condition reflects that it must be possible to write to v_a in stage s_1 and read in stage s_2, i.e., it must have a predecessor storage in stage $s_1 - 1$ and a successor storage in stage $s_2 + 1$. The subtraction/addition of 1 is applied due to the unit write delay that happens between the data are read from the previous storage and written to v_a and then between reading the data from v_a and writing them to the successor storage. For other kinds of hazards, the conditions are derived from the kind of hazard analogously as for RAW hazards as shown later on. Second, we find all maximal store executions that terminate in the storage v_a. Finally, we generate all minimal transfer executions originating from the v_a vertex of the given PSG G.[4]

The procedure for generating the set $\mathbb{H}$ is shown in Alg. 2. The procedure first constructs auxiliary sets A_{RAW}, A_{WAR}, and A_{CTL} strictly following the constraints given by RAW, WAR, and CTL hazard cases (see Definitions 19, 20, and 22). The sets A_{RAW}, A_{WAR}, and A_{CTL} consist of quintuples characterising suspected hazards. They include the architectural storage v_a on which the hazard happens, the target storage v_t through which

[4] This step is not necessary in the case of WAW hazard as the error caused by the hazard is immediate.

the hazard manifests, and three stages: namely, stages s_1 and s_2 in which the conflicting read/write operations on v_a happen, and stage s_3 in which the hazard gets manifested. For WAW hazards, the procedure later on proceeds similarly, but there is no v_t and s_3 needed since the hazard manifests immediately upon the second write operation (Definition 21). The auxiliary sets are then used for finding maximal store and minimal transfer executions in the PSG. A standard breadth-first search algorithm during which constraints from Definitions 13–18 are checked on-the-fly can be used to obtain the minimal transfer executions in G for the suspected hazards. Similarly, the procedure may deploy the depth-first search algorithm while checking constraints from Definitions 13, 14, and 17 in order to find the maximal store executions.

Algorithm 2 Procedure computing a set of hazard cases $\mathbb{H}$.

```
Require: A PSG $G = (V, E, s, t, \omega)$, a set $V_a \subseteq V$ of architectural storages, a program counter $v_{pc} \in V_a$, a set $V_p \subseteq V$ of pipeline registers, $V_a \cap V_p = \emptyset$, a set $V_{pivot} \subseteq V_p$ of pivots, and a set $S_{ic} \subseteq \mathbb{S}$ of independently clearable stages.
Ensure: A set $\mathbb{H} \subseteq \mathbb{X} \times \mathbb{X}$ of hazard cases in the CPU encoded by $G$.
$V_t := V_a \cup V_{pivot} \cup \{v \in V_p \mid \varphi(v) \in S_{ic}\}$
Let $\mathbb{A}$ denote $V_a \times \mathbb{N} \times \mathbb{N} \times V_t \times \mathbb{N}$
$A_{RAW} := \{(v_a, s_1, s_2, v_t, s_3) \in \mathbb{A} \mid s_1 - 1 \in \varphi^{\mathrm{wr}}(v_a) \wedge s_2 + 1 \in \varphi^{\mathrm{rd}}(v_a) \wedge \mathbf{s_2} < \mathbf{s_1} \wedge s_3 - 1 \in \varphi^{\mathrm{wr}}(v_t) \wedge s_2 \leq s_3\}$
$A_{WAR} := \{(v_a, s_1, s_2, v_t, s_3) \in \mathbb{A} \mid s_1 - 1 \in \varphi^{\mathrm{wr}}(v_a) \wedge s_2 + 1 \in \varphi^{\mathrm{rd}}(v_a) \wedge \mathbf{s_1} < \mathbf{s_2} \wedge s_3 - 1 \in \varphi^{\mathrm{wr}}(v_t) \wedge s_2 \leq s_3\}$
$A_{CTL} := \{(v_{pc}, s_1, 1, v_t, s_3) \in \mathbb{A} \mid s_1 - 1 \in \varphi^{\mathrm{wr}}(v_a) \wedge 2 \in \varphi^{\mathrm{rd}}(v_{pc}) \wedge \mathbf{s_1} > \mathbf{2} \wedge s_3 - 1 \in \varphi^{\mathrm{wr}}(v_t) \wedge v_{pc} \neq v_t \wedge s_3 > 1\}$
$A := A_{RAW} \cup A_{WAR} \cup A_{CTL}$
$\mathbb{H} := \emptyset$
for $(v_a, s_1, s_2, v_t, s_3) \in A$ do
    $X_{mse} := \{\ (\pi, \tau) \in \mathbb{X} \mid \pi = \langle v_1, e_1, \ldots, e_{k-1}, v_k \rangle \wedge (\{v_k\} \times \mathbb{N} \times \mathbb{N} \times V_t \times \mathbb{N}) \cap A \neq \emptyset \wedge t(e_{k-1}) = (v_k, \mathtt{d}) \wedge v_2, \ldots, v_{k-1} \notin (V_a \cup V_p) \wedge \tau^{\mathrm{lst}} = s_1 \wedge (\pi, \tau)$ is a maximal store execution $\}$
    $X_{mte} := \{\ (\pi, \tau) \in \mathbb{X} \mid \pi = \langle v_1, e_1, \ldots, e_{k-1}, v_k \rangle \wedge (\{v_1\} \times \mathbb{N} \times \mathbb{N} \times \{v_k\} \times \mathbb{N}) \cap A \neq \emptyset \wedge \tau^{\mathrm{fst}} = s_2 \wedge \tau^{\mathrm{lst}} = s_3 \wedge (\pi, \tau)$ is a minimal transfer execution $\}$
    $\mathbb{H} := \mathbb{H} \cup (X_{mse} \times X_{mte})$
end for
Let $\mathbb{A}'$ denote $V_a \times \mathbb{N} \times \mathbb{N}$
$A_{WAW} := \{(v_a, s_1, s_2) \in \mathbb{A}' \mid s_1 - 1, s_2 - 1 \in \varphi^{\mathrm{wr}}(v_a) \wedge \mathbf{s_2} < \mathbf{s_1}\}$
for $(v_a, s_1, s_2) \in A_{WAW}$ do
    $X^1_{mse} := \{\ (\pi, \tau) \in \mathbb{X} \mid \pi = \langle v_1, e_1, \ldots, e_{k-1}, v_k \rangle \wedge (\{v_k\} \times \mathbb{N} \times \mathbb{N}) \cap A_{WAW} \neq \emptyset \wedge t(e_{k-1}) = (v_k, \mathtt{d}) \wedge v_2, \ldots, v_{k-1} \notin (V_a \cup V_p) \wedge \tau^{\mathrm{lst}} = s_2 \wedge (\pi, \tau)$ is a maximal store execution $\}$
    $X^2_{mse} := \{\ (\pi, \tau) \in \mathbb{X} \mid \pi = \langle v_1, e_1, \ldots, e_{k-1}, v_k \rangle \wedge (\{v_k\} \times \mathbb{N} \times \mathbb{N}) \cap A_{WAW} \neq \emptyset \wedge t(e_{k-1}) = (v_k, \mathtt{d}) \wedge v_2, \ldots, v_{k-1} \notin (V_a \cup V_p) \wedge \tau^{\mathrm{lst}} = s_1 \wedge (\pi, \tau)$ is a maximal store execution $\}$
    $\mathbb{H} := \mathbb{H} \cup (X^1_{mse} \times X^2_{mse})$
end for
return $\mathbb{H}$
```

Example 4. Consider the PSG from Fig. 9.2 and the mappings shown in Table 9.1. One can see that there is a potential WAR hazard on the index register $X \in V_a$ because, for example, it can be written in Stage 3 ($\varphi^{\mathrm{wr}}(X) = \{2, 3, 4\}$) and read by Stage 5 ($\varphi^{\mathrm{rd}}(X) = \{3, 4, 5\}$). By Definition 20, to form a WAR hazard, the PSG must contain (i) a maximal store execution of a spoiler instruction $(\pi_{sp}, \tau_{sp}) \in \mathbb{X}$ ending in X and (ii) a minimal transfer execution $(\pi_{vi}, \tau_{vi}) \in \mathbb{X}$ leading from X to some target storage. There are multiple executions of spoiler and victim instructions that satisfy the above criteria. Each of them must be considered in order to verify that the design is free of WAR hazards. For instance, one may consider a spoiler execution (π_{sp}, τ_{sp}) with $\pi_{sp} = \langle X,\ \mathit{Inc}.\mathtt{a}_1,\ \mathit{Inc},\ \mathit{MxInc}.\mathtt{c}_0,\ \mathit{MxInc},\ X.\mathtt{d},\ X\rangle$ and $\tau_{sp} = \{1^{X} \mapsto 2,\ 2^{\mathit{Inc}.\mathtt{a}_1} \mapsto 2,\ 3^{\mathit{Inc}} \mapsto 2,\ 4^{\mathit{MxInc}.\mathtt{c}_0} \mapsto 2,\ 4^{\mathit{MxInc}} \mapsto 2,\ 5^{X.\mathtt{d}} \mapsto 2,\ 6^{X} \mapsto 3\}$. Further, we can consider a victim execution (π_{vi}, τ_{vi}) with the target memory cell Mem_j written in Stage 5 where $\pi_{vi} = \langle X,\ \mathit{Cmp}_j.\mathtt{a}_0,\ \mathit{Cmp}_j,\ \mathit{MxSel}_j.\mathtt{c}_1,\ \mathit{MxSel}_j,\ \mathit{Mem}_j.\mathtt{en},\ \mathit{Mem}_j\rangle$. An instance of an execution plan τ_{vi} for the walk π_{vi} is $\{1^{X} \mapsto 4,\ 2^{\mathit{Cmp}_j.\mathtt{a}_0} \mapsto 4,\ 3^{\mathit{Cmp}_j} \mapsto 4,\ 4^{\mathit{MxSel}_j.\mathtt{c}_1} \mapsto 4,\ 5^{\mathit{MxSel}_j} \mapsto 4,\ 6^{\mathit{Mem}_j.\mathtt{en}} \mapsto 4,\ 7^{\mathit{Mem}_j} \mapsto 5\}$. The given pair of a spoiler and victim is clearly a candidate for a WAR hazard since the needed data are overwritten before they are read (unless some control logic over the involved executions prevents the hazard, which will be the subject of further checking). ◁

Example 5. Further, as an example of a control hazard, one can consider a spoiler execution (π_{sp}, τ_{sp}) with $\pi_{sp} = \langle \mathit{ExAlu},\ \mathit{MxAlu}.\mathtt{sel},\ \mathit{MxAlu},\ \mathit{MxPC}.\mathtt{c}_1,\ \mathit{MxPC},\ \mathit{PC}.\mathtt{d},\ \mathit{PC}\rangle$ and $\tau_{sp} = \{1^{\mathit{ExAlu}} \mapsto 4,\ 2^{\mathit{MxAlu}.\mathtt{sel}} \mapsto 4,\ 3^{\mathit{MxAlu}} \mapsto 4,\ 4^{\mathit{MxPC}.\mathtt{c}_1} \mapsto 4,\ 5^{\mathit{MxPC}} \mapsto 4,\ 6^{\mathit{PC}.\mathtt{d}} \mapsto 4,\ 7^{\mathit{PC}} \mapsto 5\}$. As an instance of a victim execution (π_{vi}, τ_{vi}), we can consider an execution walk $\pi_{vi} = \langle \mathit{PC},\ \mathit{MxProg}.\mathtt{sel},\ \mathit{MxProg},\ \mathit{IdIr}.\mathtt{d},\ \mathit{IdIr}\rangle$ with an execution plan $\tau_{vi} = \{1^{\mathit{PC}} \mapsto 1,\ 2^{\mathit{MxProg}.\mathtt{sel}} \mapsto 1,\ 3^{\mathit{MxProg}} \mapsto 1,\ 4^{\mathit{IdIr}.\mathtt{d}} \mapsto 1,\ 5^{\mathit{IdIr}} \mapsto 2\}$. Note that, in this case, $\mathit{IdIr} \notin V_a$, but we know from Table 9.1 that the pipeline register IdIr is a pivot, and so it is still a valid terminating element for a transfer execution. ◁

9.5 Parametric Systems for Potential Hazards

We will now describe how the potentially hazardous behavior of a spoiler and a victim instruction described by a hazard case can be modeled and checked for feasibility using a parametric system P: if the behavior is not feasible, the hazard case does not describe a real hazard (the suspected hazard gets prevented by the pipeline flow logic). In the system P, we map $n \geq 2$ instructions in the pipeline to n processes in a linear array (with the earliest instruction on the left). Note that the value of n is not constrained from above. Indeed, while there is a single spoiler and victim, we do not know how many "padding" instructions should appear between the spoiler and the victim for the hazard to manifest. That is why, we model the system as parametric, with n being the parameter, and verify it for any value of n.

Initially, the instructions 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 means of the 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.

In the following explanation, we start by constructing the set of states of the system P. Then, we proceed to capturing the above mentioned influence of the pipeline flow logic and reflect it in the transition relation of the system P. Finally, we define the set of minimal bad configurations of the system P that describes the prohibited interleavings of instructions causing the hazard.

9.5.1 States and Edge Conditions of the Parametric System

Given a hazard case of the form $(\chi_{sp}, \chi_{vi}) \in \mathbb{X}^2$, $\chi_{sp} = (\pi_{sp}, \tau_{sp})$, $\chi_{vi} = (\pi_{vi}, \tau_{vi})$, the parametric system P will model interactions among four classes of processes, resp. instructions, $\mathbb{K} := \{sp$ "spoiler", vi "victim", sf "stall-flow", nf "normal-flow"$\}$. This follows the fact that each type of the considered pipeline hazard is caused by some pair of instructions. The sp class represents the spoiler part of the hazard case, i.e., an instruction that writes to a storage $v \in V_a$ in a stage $\tau_{sp}(v)$. The vi class then represents an instruction corresponding to the victim part of the hazard case, reading or writing from/to v in a stage $\tau_{vi}(v)$. Further, the sf and nf classes both denote any other instructions than the spoiler and victim—we just differentiate two operating modes of these instructions. As we will discuss later in Section 9.5.2, the difference between the stall- and normal-flow operation modes is that an sf-class instruction in a stage $s_0 \in \mathbb{S}$ causes that all pipeline stages $s \in \mathbb{S}$ s.t. $s < s_0$ get stalled. Both the sf and nf classes serve as a pipeline filler and a sink for cleared (flushed) instructions.

To facilitate the construction of a parametric system allowing us to verify whether a given hazard case corresponds to a real hazard or not, we need to introduce an *extended* set of stages. Let $\bar{\mathbb{S}} := \mathbb{S} \cup \{\bot, \top\}$ be the set of stages extended with auxiliary initial "$\bot$" and final "$\top$" stages. We will then represent the behavior of instructions given by a hazard case $h = (\chi_{sp}, \chi_{vi})$ in the form of a labelled parametric system, called a *hazard system* (HS), $P^h = (Q^h, \Delta^h, \alpha^h)$ where $Q^h := \mathbb{K} \times \bar{\mathbb{S}}$, Δ^h will be introduced in Section 9.5.2, and $\alpha^h \colon Q^h \to 2^{\mathbb{E}}$ is a state labelling function. The labelling function α^h associates each state with a set of edge conditions that should hold in this state for the hazard to be executable. We will show the construction of the labelling below. Note that each state $q \in Q^h$ represents a unique instruction class and a stage in which an instruction of this class is supposed to be. Finally, for a proper understanding of the rest of the section, we once again stress that the particular states in Q^h are states of *individual instructions*, not of the entire system. A configuration of the system P^h is a sequence of such states.

Next, we define the mapping α^h describing which edge conditions must hold in a state $q = \langle \kappa, s \rangle \in Q^h$, which is a state of an instruction of the class $\kappa \in \mathbb{K}$ in the stage $s \in \bar{\mathbb{S}}$, for that instruction to execute in accordance with the hazard case h. First, for instructions of the classes $\kappa = sf$ and $\kappa = nf$, we define $\alpha^h(\langle \kappa, s \rangle) := \emptyset$ for every $s \in \bar{\mathbb{S}}$ since we do not expect any special behavior from instructions of these classes, and, on every realistic processor, we can always find instructions that do not interfere with the spoiler and victim instructions and may serve as the needed pipeline filler. Likewise, we define $\alpha^h(\langle \kappa, s \rangle) := \emptyset$ for any $\kappa \in \mathbb{K}$ and $s \in \{\bot, \top\}$, i.e., for instructions that have not yet started or that have already ended.

For the spoiler and victim instructions, the idea is to extract the edge conditions by looking for the necessary settings of selector, enable, and clear edges so that the data involved in the potential hazard are carried over the walks π_κ for $\kappa \in \{sp, vi\}$ that are a part of the concerned spoiler and victim executions $\chi_\kappa = (\pi_\kappa, \tau_\kappa)$. The mapping α^h can be constructed from three auxiliary mappings $\alpha^h_{\mathtt{sel}}$, $\alpha^h_{\mathtt{en}}$, and $\alpha^h_{\mathtt{rst}} \colon \mathbb{X} \to 2^{\mathbb{E} \times \mathbb{S}}$ where $\alpha^h_{\mathtt{sel}}$ will be examining all edges but the last one (hence covering all edges that route the data through multiplexers) and the last edge will be covered by exactly one of the two remaining mappings (related to enabling a write of the data to the target storage or clearing the storage). In particular, the $\alpha^h_{\mathtt{sel}}$ mapping is defined as

$$\begin{aligned}\alpha^h_{\mathtt{sel}}(\chi) := \{(\sigma_{v_i}(e_{i-1}), \tau(2i-1)) \mid 1 < i < k \wedge \\ v_i \in V_{mx} \wedge \chi = (\langle v_1, e_1, \ldots, e_{i-1}, v_i, \ldots, v_k\rangle, \tau)\}.\end{aligned} \tag{9.6}$$

Intuitively, the $\alpha^h_{\mathtt{sel}}$ mapping produces a set of pairs consisting of a condition $\sigma_{v_i}(e_{i-1}) \in \mathbb{E}$ over selector edges that is required by the multiplexer $v_i \in V_{mx}$ to propagate the data along the execution walk π and the stage $\tau(v_i)$ in which the particular condition must be satisfied. Similarly, the $\alpha^h_{\mathtt{en}}$ and $\alpha^h_{\mathtt{rst}}$ mappings establish the necessary condition for the final edge of the execution's target storage, making sure that either writing of the data into the storage is enabled or the storage is cleared:

$$\begin{aligned}\alpha^h_{\mathtt{en}}(\chi) := \{(v_k.\mathtt{en} \rightsquigarrow 1, \tau(2k-1)) \mid \\ \chi = (\langle v_1, e_1, \ldots, e_{k-1} = v_k.\mathtt{d}, v_k\rangle, \tau)\},\end{aligned} \tag{9.7}$$

$$\begin{aligned}\alpha^h_{\mathtt{rst}}(\chi) := \{(v_k.\mathtt{rst} \rightsquigarrow 1, \tau(2k-1) \mid \\ \chi = (\langle v_1, e_1, \ldots, e_{k-1} = v_k.\mathtt{rst}, v_k\rangle, \tau)\}.\end{aligned} \tag{9.8}$$

In particular, $\alpha^h_{\mathtt{en}}$ ensures that the data transferred along the path described by the execution χ are indeed written to its destination storage v_k at the end of the execution. Therefore, $\alpha^h_{\mathtt{d}}$ produces a singleton containing a pair consisting from the condition $v_k.\mathtt{en} \rightsquigarrow 1$ and the stage $\tau(2k-1)$ which is the stage where the data reside just prior to the write. Similarly, $\alpha^h_{\mathtt{rst}}$ produces a singleton containing a pair consisting from the condition $v_k.\mathtt{rst} \rightsquigarrow 1$ and the stage $\tau(2k-1)$ so that the target storage is indeed cleared. Using the above mappings, we can define α^h for the given hazard case $h = (\chi_{sp}, \chi_{vi})$ such that the following holds for any state $\langle \kappa, s\rangle \in \{sp, vi\} \times \bar{\mathbb{S}}$:[5]

$$\begin{aligned}\alpha^h(\langle \kappa, s\rangle) := \{c \in \mathbb{E} \mid (c, s) \in \alpha^h_{\mathtt{sel}}(\chi_\kappa) \cup \alpha^h_{\mathtt{en}}(\chi_\kappa) \cup \\ \alpha^h_{\mathtt{rst}}(\chi_\kappa)\}.\end{aligned} \tag{9.9}$$

Example 6. Assume the hazard case (χ_{sp}, χ_{vi}) shown in Example 4 for the microprocessor from Example 1. First, we focus on the spoiler execution $\chi_{sp} = (\pi_{sp}, \tau_{sp})$. Since the microprocessor contains five pipeline stages, the spoiler gets associated with the set of states $Q^h_{sp} := \{sp\} \times \bar{\mathbb{S}}$ where $\bar{\mathbb{S}} = \{\perp, 1, \ldots, 5, \top\}$. We will now show how the α^h mapping is computed for the states of Q^h_{sp}. From the definition of α^h, it directly follows that

$$\alpha^h(\langle sp, \perp\rangle) = \alpha^h(\langle sp, \top\rangle) = \emptyset.$$

For the states $\langle sp, 1\rangle$, ..., $\langle sp, 5\rangle$, one has to first compute the auxiliary mappings $\alpha^h_{\mathtt{sel}}$, $\alpha^h_{\mathtt{en}}$, and $\alpha^h_{\mathtt{rst}}$ from Equation 9.9. As the X register is written via its $\mathtt{d}$ connection, it immediately follows that

$$\alpha^h_{\mathtt{rst}}(\chi_{sp}) = \emptyset.$$

Next, since the walk π_{sp} of the spoiler store execution χ_{sp} passes through a single multiplexer, namely, *MxInc*, via the edge *MxInc*.$\mathtt{c}_0$ with $\tau_{sp}(4^{MxInc.\mathtt{c}_0}) = 2$, we get

$$\alpha^h_{\mathtt{sel}}(\chi_{sp}) = \{(MxInc.\mathtt{sel} \rightsquigarrow 0, 2)\}.$$

[5] Note that the executions can also end by an $v_k.\mathtt{en}$ edge. However, in this case, no matter what the value of the enable signal is a hazard happens by enabling/not enabling a write of some data into an architectural storage. Hence, no further condition is needed in this case.

For $\alpha^h_{\texttt{en}}$, we only need to assure that the storage X is written at the end of the execution. Since $\tau_{sp}(5^{X.\texttt{d}}) = 2$, we let

$$\alpha^h_{\texttt{en}}(\chi_{sp}) = \{(X.\texttt{en} \rightsquigarrow 1, 2)\}.$$

Finally, by uniting the above computed auxiliary mappings, we get that

$$\alpha^h(\langle sp, 2\rangle) = \{\mathit{MxInc}.\texttt{sel} \rightsquigarrow 0, X.\texttt{en} \rightsquigarrow 1\}$$

and $\forall i \in \mathbb{S} \setminus \{2\} \colon \alpha^h(\langle sp, i\rangle) = \emptyset$. Analogically, for the victim execution $\chi_{vi} = (\pi_{vi}, \tau_{vi})$ of the analyzed hazard case, we would infer that

$$\alpha^h_{\texttt{sel}}(\chi_{vi}) = \{(\mathit{MxSel}_j.\texttt{sel} \rightsquigarrow 1, 4)\}$$

and $\alpha^h_{\texttt{rst}}(\chi_{vi}) = \alpha^h_{\texttt{en}}(\chi_{vi}) = \emptyset$. Therefore, we get that

$$\alpha^h(\langle vi, 4\rangle) = \{\mathit{MxSel}_j.\texttt{sel} \rightsquigarrow 1\}$$

and $\forall i \in \mathbb{S} \setminus \{4\} \colon \alpha^h(\langle vi, i\rangle) = \emptyset$. ◁

9.5.2 Transition Relation of the Parametric System

For the construction of the transition relation Δ^h presented later on, we will first introduce three predicates that characterise mutual interactions of pairs of instructions whose execution has reached some states $q_1, q_2 \in Q^h$ of the verified HS P^h. We stress that q_1 and q_2 are states of the execution of two considered instructions, which are of course a part of a single configuration of the HS P^h. Before providing rigorous definitions of the predicates, which are given later in this section, we first provide some intuition behind them.

A pair of states $q_1, q_2 \in Q^h$ and a stage $s \in \mathbb{S}$ satisfy the ternary *stage stall* predicate $\xleftrightarrow{\text{st},h} \subseteq Q^h \times \mathbb{S} \times Q^h$ provided that the edge conditions associated with the states q_1 and q_2 ensure that the stage s is stalled, and thus the contents of all pipeline storages of s stays unchanged. We will further use the shorthand $q_1 \xleftrightarrow{\text{st},h,s} q_2$ for $(q_1, s, q_2) \in \xleftrightarrow{\text{st},h}$.

Further, a pair of states $q_1, q_2 \in Q^h$ and a stage $s \in \mathbb{S}$ satisfy the ternary *stage clear* predicate $\xleftrightarrow{\text{cl},h} \subseteq Q^h \times \mathbb{S} \times Q^h$ provided that the stage s is cleared, i.e., the contents of all pipeline storages of s is nullified. We will further use the shorthand $q_1 \xleftrightarrow{\text{cl},h,s} q_2$ for $(q_1, s, q_2) \in \xleftrightarrow{\text{cl},h}$.

Finally, a pair of states $q_1, q_2 \in Q^h$ satisfies a binary *state conflict* predicate $\xleftrightarrow{\text{cf},h} \subseteq Q^h \times Q^h$ provided that the given processor excludes a configuration where two instructions would appear in the states q_1, q_2 at the same time. We will further use the shorthand $q_1 \xleftrightarrow{\text{cf},h} q_2$ for $(q_1, q_2) \in \xleftrightarrow{\text{cf},h}$. For instance, one of the typical scenarios when two states q_1, $q_2 \in Q^h$ are in a state conflict occurs when there exists an edge $e \in E$ so that $e \rightsquigarrow b_1 \in \alpha^h(q_1) \wedge e \rightsquigarrow b_2 \in \alpha^h(q_2)$, b_1, $b_2 \in \mathbb{B}$, while $b_1 \neq b_2$.

In order to formally define the above described predicates, we first introduce two auxiliary notions: in particular, (i) a mapping $\mathit{unwind}_h \colon Q^h \to 2^C$ where C is the set of configurations of the TS $T^h = (C, \hookrightarrow)$ induced by the PSG and (ii) a predicate $\mathit{csat}_h \subseteq 2^{\mathbb{E}} \times 2^{Q^h}$.

The purpose of the unwind_h mapping is to compute all configurations of the TS T^h in which T^h (and hence the processor it represents) can be when the processor contains an instruction of a class κ in a stage s while executing within the given hazard case h. The considered configurations must be such that the processor can reach them by going through all preceding stages and such that the processor can finish the execution of the instruction by going through all its further stages, all the time executing within the hazard case h. In particular, let $m = \max(\mathbb{S})$ be the number of stages and let $\langle \kappa, s \rangle \in Q^h$ be an instruction state representing an instruction of a class κ in a stage s within a hazard case h. Then, $unwind_h(\langle \kappa, s \rangle)$ consists of exactly all those configurations $k_0 \in C$ such that there is a trace $\langle k_{-s}, \ldots, k_0, \ldots, k_{m-s} \rangle$ in T^h that conforms to the following rules for all i such that $-s < i \leq m - s$:

$$k_i \hookrightarrow k_{i+1}, \tag{9.10}$$

$$k_i \in \gamma(\alpha^h(\langle \kappa, s + i \rangle)). \tag{9.11}$$

The first constraint above ensures that we indeed consider a trace in the TS T^h. The second condition then ensures that the trace passes all stages of an instruction of the given class while the processor is executing within the given hazard case.

The above described computation of the $unwind_h$ mapping can be implemented symbolically using a BVL formula $unwind^\star_h(q)$ for any $q \in Q^h$. To describe the computation, we introduce the notation $\hookrightarrow^\star_{(i,i+1)}$ to denote the result of a (straightforward) conversion of the relation $\hookrightarrow$ to a BVL formula where all variables representing the current state of the TS T^h are indexed with i and those representing the future state are indexed with $i + 1$. Moreover, as in Section 9.3.2, we use $e^\star_i$ to denote the conversion of an edge $e \in E$ indexed with the trace index i to a BVL variable. Then, given $q = \langle \kappa, s \rangle \in Q^h \setminus \mathbb{K} \times \{\perp, \top\}$, the BVL formula $unwind^\star_h(q)$ is obtained as follows:

$$\begin{array}{rcll}
F_1 & := & \bigwedge\limits_{i=-s+1}^{m-s-1} & \hookrightarrow^\star_{(i,i+1)}, \\
F_2(q) & := & \bigwedge\limits_{i=-s+1}^{m-s} & \bigwedge\limits_{e \rightsquigarrow b \in \alpha^h(\langle \kappa, s+i \rangle)} e^\star_i = b, \\
F_3 & := & \bigwedge\limits_{e \in E} & e^\star = e^\star_0, \\
unwind^\star_h(q) & := & \multicolumn{2}{l}{\exists \overline{E} : F_1 \wedge F_2(q) \wedge F_3.}
\end{array} \tag{9.12}$$

Above, the existential quantification ranges over the set $\overline{E} = \{e^\star_i \mid e \in E \wedge -s < i \leq m - s\}$. Its reason is to get rid of the concrete past and future values of the variables that appear in the execution, keeping only their impact on the current values of the variables.[6] Finally, in order to extend the definition of $unwind_h$ for initial and final states $q' \in \mathbb{K} \times \{\perp, \top\}$, we define $unwind^\star_h(q') := true$.

Further, we proceed to the second auxiliary predicate: $csat_h$. The $csat_h$ predicate determines satisfiability of a set of edge conditions $I \subseteq \mathbb{E}$ in a situation when the pipeline contains instructions in states from a set $S \subseteq Q^h$. Formally, it is defined as follows:

$$csat_h(I, S) \Leftrightarrow \bigcap_{q \in S} unwind_h(q) \cap \bigcap_{c \in I} \gamma(c) \neq \emptyset. \tag{9.13}$$

The evaluation of $csat_h(I, S)$ can be naturally reduced to checking the satisfiability of a BVL formula as follows:

$$csat_h(I, S) \Leftrightarrow sat\Big(\bigwedge_{q \in S} unwind^\star_h(q) \wedge \bigwedge_{e \rightsquigarrow b \in I} e^\star = b \Big). \tag{9.14}$$

[6] In our implementation of the approach, we replace the existential quantification by simply pruning away all variables unrelated with any $e^\star$ for any $e \in E$ and by renaming the remaining variables in a unique way such that no conflicts arise when constructing more complex formulae on top $unwind^\star_h(q)$.

Now, the predicate $csat_h$ can be used to precisely define the needed predicates $\xleftrightarrow{\text{st},h}$, $\xleftrightarrow{\text{cl},h}$, and $\xleftrightarrow{\text{cf},h}$ as follows.

Definition 23. For any instruction states q_1, $q_2 \in Q^h$ and any stage $s \in \mathbb{S}$, the *stage stall* predicate $q_1 \xleftrightarrow{\text{st},h,s} q_2$ is defined as follows:

$$\begin{aligned} q_1 \xleftrightarrow{\text{st},h,s} q_2 \iff \exists \quad & v_p \in V_p : \varphi(v_p) = s \;\wedge \\ & \neg csat_h(\{v_p.\texttt{en} \rightsquigarrow 1\}, \{q_1, q_2\}) \;\wedge \\ & \neg csat_h(\{v_p.\texttt{rst} \rightsquigarrow 1\}, \{q_1, q_2\}). \end{aligned} \tag{9.15}$$

Intuitively, the definition requires that the presence of some instructions in states q_1 and q_2 in the pipeline ensures that there is a pipeline storage v_p in stage s, which we denote as a *representative storage* below, such that the value of v_p can neither be updated nor cleared, i.e., v_p keeps its value. Note that the already established validity of the consistency Rules 1 and 4 implies that the setting of any control edge (`en`, `rst`) is the same for all pipeline storages across the given pipeline stage, and so the fact that some representative storage is stalled means that all storages of the given stage are stalled (and the instruction that is now in stage s stays in it).

In a similar fashion, we define the $\xleftrightarrow{\text{cl},h}$ predicate.

Definition 24. For any instruction states q_1, $q_2 \in Q^h$ and any stage $s \in \mathbb{S}$, the *stage clear* predicate $q_1 \xleftrightarrow{\text{cl},h,s} q_2$ is defined as follows:

$$\begin{aligned} q_1 \xleftrightarrow{\text{cl},h,s} q_2 \iff \exists \quad & v_p \in V_p : \varphi(v_p) = s \;\wedge \\ & \neg csat_h(\{v_p.\texttt{rst} \rightsquigarrow 0\}, \{q_1, q_2\}). \end{aligned} \tag{9.16}$$

Note that the definition requires that the representative storage *must* be cleared (since the formula cannot be satisfied with the v_p.`rst` edge being zero). The consistency rules then assure that the same holds for all storages of the given stage.

In order to be able to define the $\xleftrightarrow{\text{cf},h}$ predicate, we only need to be able to determine whether two given instruction states are prohibited from occurring together in a single pipeline configuration by the control logic of the considered processor. This is, however, easy thanks to the $csat_h$ predicate as shown below.

Definition 25. For any instruction states $q_1, q_2 \in Q^h$, the *state conflict* predicate $q_1 \xleftrightarrow{\text{cf},h} q_2$ is defined as follows:

$$q_1 \xleftrightarrow{\text{cf},h} q_2 \iff \neg csat_h(\emptyset, \{q_1, q_2\}). \tag{9.17}$$

Intuitively, the expression $csat_h(\emptyset, \{q_1, q_2\})$ does not put any constraints on edge conditions, but it still checks whether some concurrently executing instructions can simultaneously get into states q_1 and q_2. Hence, its negation says that this is excluded in the given processor, allowing us to define the $\xleftrightarrow{\text{cf},h}$ predicate.

Example 7. In this example, we will demonstrate how the predicate $\xleftrightarrow{\text{st},h}$ can be evaluated for a given pair of states and a given stage. Let us consider states $\langle sp, 2\rangle$, $\langle vi, 3\rangle$, Stage 2, and the hazard case $h = (\chi_{sp}, \chi_{vi})$ from Example 4. Here, the spoiler instruction in state $\langle sp, 2\rangle$ writes into the register X the (auto-incremented) value previously read from the same register. The victim instruction in state $\langle vi, 4\rangle$ then reads the value j from the register X and uses it as an index to access the memory cell Mem_j.

From Definition 23, we know that, in order to determine the value of $\langle sp, 2\rangle \xleftrightarrow{\mathrm{st},h,2} \langle vi, 3\rangle$, one has to (i) pick a representative pipeline storage $v_p \in \{v \in V_p \mid \varphi(v) = 2\}$, (ii) evaluate $\Phi_1 := \neg csat(\{v_p.\texttt{en} \rightsquigarrow 1\}, \{\langle sp, 2\rangle, \langle vi, 3\rangle\})$, and (iii) evaluate $\Phi_2 := \neg csat(\{v_p.\texttt{rst} \rightsquigarrow 1\}, \{\langle sp, 2\rangle, \langle vi, 3\rangle\})$.

As for Step (i) above, it suffices to look in Table 9.1 and choose, for instance, *IdIr* as the representative storage. Moreover, in Example 1, we have pointed out that the value of the enable edge on the *IdIr* storage is determined by the following expression in *BVL*:

$$IdIr.\texttt{en}^\star = \neg IncX.\texttt{q}^\star \vee \neg OfWrMem.\texttt{q}^\star. \tag{9.18}$$

Now, to address Step (ii), we know that, according to Equation 9.14, Φ_1 expands to

$$\begin{aligned}&\neg sat(unwind^\star_h(\langle sp, 2\rangle) \wedge unwind^\star_h(\langle vi, 3\rangle) \wedge \\ &\quad IdIr.\texttt{en}^\star = 1).\end{aligned} \tag{9.19}$$

We further concetrate on the expansion of $unwind^\star_h(\langle sp, 2\rangle)$. According to Equation 9.12, we need to construct formulae F_1, $F_2(\langle sp, 2\rangle)$, and F_3. First, the transition relation described by Formula F_1 contains the following conjuncts[7]:

$$\begin{aligned}Impl.\texttt{q}^\star_0 &= (IncX.\texttt{q}^\star_0 \Rightarrow ExWrX.\texttt{q}^\star_0) \quad \wedge \\ MxInc.\texttt{sel}^\star_0 &= Impl.\texttt{q}^\star_0.\end{aligned} \tag{9.20}$$

To see that the above holds, it suffices to check how the value of $MxInc.\texttt{sel}$ is computed from its predecesors in the PSG shown in Fig. 9.2[8]. The formula $F_2(\langle sp, 2\rangle)$ then gives

$$MxInc.\texttt{sel}^\star_0 = 0 \wedge X.\texttt{en}^\star_0 = 1, \tag{9.21}$$

which is a direct consequence of the result that we have obtained in Example 6 where we have shown

$$\alpha(\langle sp, 2\rangle) = \{MxInc.\texttt{sel} \rightsquigarrow 0, X.\texttt{en} \rightsquigarrow 1\}.$$

Finally, Formula F_3 simply asserts equality between zero-indexed and non-indexed variables. We can then apply the existential quantification from Equation 9.12, which allows us to get rid of the indexed variables, leading to that the below equality must hold:

$$IncX.\texttt{q}^\star = 1. \tag{9.22}$$

[7] The entire formula is, of course, much bigger—indeed, it describes the entire transition relation. When the satisfiability checking is done automatically, the solver will consider the entire formula. However, we select its relevant parts only so that the example is readable.

[8] We assume that the *Impl* vertex of the PSG computes the standard implication function $f_{impl}(a_0, a_1) := a_0 \Rightarrow a_1$ for $a_0, a_1 \in \mathbb{B}$.

Now, we will apply a similar approach to expand the formula $\mathit{unwind}^{\star}_{h}(\langle \mathit{vi}, 3\rangle)$. In this case, the following conjuncts of Formula F_1 turn out to be relevant:

$$\begin{aligned}
\mathit{ExWrMem}.\mathtt{d}^{\star}_{0} &= \mathit{OfWrMem}.\mathtt{q}^{\star}_{0} && \wedge \\
\mathit{ExWrMem}.\mathtt{q}^{\star}_{1} &= f^{\mathit{next}}_{\mathit{ExWrMem}}(\mathit{ExWrMem}.\mathtt{q}^{\star}_{0}, \\
&\quad \mathit{ExWrMem}.\mathtt{d}^{\star}_{0}, \mathit{ExWrMem}.\mathtt{en}^{\star}_{0}, \\
&\quad \mathit{ExWrMem}.\mathtt{rst}^{\star}_{0}) && \wedge \\
\mathit{MxSel}_{j}.\mathtt{sel}^{\star}_{1} &= \mathit{ExWrMem}.\mathtt{q}^{\star}_{1}.
\end{aligned} \tag{9.23}$$

Above, $f^{\mathit{next}}_{\mathit{ExWrMem}}$ is the next-state function that was defined in Section 9.1.2 and that propages the value on the data-in edge $\mathtt{d}$ to the data-out edge $\mathtt{q}$ iff the enable edge $\mathtt{en}$ is set and the reset edge $\mathtt{rst}$ is unset. Moreover, if $\mathtt{rst}$ is set, then the data-out $\mathtt{q}$ is nullified. Otherwise, when both $\mathtt{en}$ and $\mathtt{rst}$ are unset, the data-out edge $\mathtt{q}$ keeps the value from the previous cycle. Further, in Example 6, we have seen that

$$\alpha(\langle \mathit{vi}, 4\rangle) = \{\mathit{MxSel}_{j}.\mathtt{sel} \rightsquigarrow 1\},$$

which imples that the formula $F_2(\langle \mathit{vi}, 3\rangle)$ must ensure

$$\mathit{MxSel}.\mathtt{sel}^{\star}_{1} = 1. \tag{9.24}$$

By combining the observations from Formulae 9.23 and 9.24, and by adding Formula F_3 and the existential quantification of Equation 9.12, we obtain the following statement:

$$\begin{aligned}
\big(\ (\mathit{ExWrMem}.\mathtt{en}^{\star} = 1) &\Rightarrow (\mathit{OfWrMem}.\mathtt{q}^{\star} = 1)\ \big) \quad \wedge \\
\mathit{ExWrMem}.\mathtt{rst}^{\star} &= 0.
\end{aligned} \tag{9.25}$$

Here, the $\mathit{ExWrMem}.\mathtt{rst}^{\star} = 0$ conjuct comes from the fact that the data-out edge must not be zero because of the constraint in Formula 9.24.

Next, according to the consistency Rule 3 from Section 9.3.2 that holds globally at any pipeline cycle, the following expression must hold:

$$\begin{aligned}
(\mathit{ExWrMem}.\mathtt{en}^{\star} &= 0 \wedge \mathit{ExWrMem}.\mathtt{rst}^{\star} = 0) \quad \Rightarrow \\
(\mathit{IdIr}.\mathtt{en}^{\star} &= 0 \wedge \mathit{IdIr}.\mathtt{rst}^{\star} = 0).
\end{aligned} \tag{9.26}$$

In particular, the above comes from the fact that $\varphi(\mathit{IdIr}) + 1 = \varphi(\mathit{ExWrMem})$, i.e., IdIr and $\mathit{ExWrMem}$ are two pipeline storages in adjacent stages.

By applying the modus tollens rule on Formula 9.26, we get

$$\begin{aligned}
(\mathit{IdIr}.\mathtt{en}^{\star} &= 1 \vee \mathit{IdIr}.\mathtt{rst}^{\star} = 1) \quad \Rightarrow \\
(\mathit{ExWrMem}.\mathtt{en}^{\star} &= 1 \vee \mathit{ExWrMem}.\mathtt{rst}^{\star} = 1).
\end{aligned} \tag{9.27}$$

Finally, if we put together our observations made in Formulae 9.18, 9.22, 9.25, and 9.27, we can conclude that the expression

$$\mathit{unwind}^{\star}_{h}(\langle \mathit{sp}, 2\rangle) \wedge \mathit{unwind}^{\star}_{h}(\langle \mathit{vi}, 3\rangle) \wedge \mathit{IdIr}.\mathtt{en}^{\star} = 1$$

is not satisfiable. Thus, the expression Φ_1 evaluates to *true*.

Analogically, for Step (iii), we would also derive that Φ_2 is *true*, and therefore the predicate $\langle sp, 2\rangle \xleftrightarrow{\text{st},h,2} \langle vi, 3\rangle$ necessarily holds. In other words, this means that the `NOP` injection into Stage 3 takes place whenever there is a spoiler defined by χ_{sp} in Stage 2 and a victim described by χ_{vi} in Stage 3. $\lhd$

We can now define transitions that the transition relation Δ^h of the HS P^h contains. First, for every instruction state $q = \langle \kappa, s\rangle \in Q^h$, Δ^h contains a transition $q \to q$ allowing the instruction that is in q to stay in q whenever the state q appears in a configuration of the pipeline of the given processor (i.e., a configuration of the transition system induced by P^h) that contains a combination of instruction states $q_1,\ q_2 \in Q^h$ which causes the instruction in the state q to be stalled. Formally, $\forall q = \langle \kappa, s\rangle, q_1, q_2 \in Q^h$:

$$(\exists_{\leftrightarrow}\colon \{q_1, q_2\} \models q \to q) \in \Delta^h \Leftrightarrow q_1 \xleftrightarrow{\text{st},h,s} q_2. \tag{9.28}$$

As we have already mentioned at the beginning of Section 9.5, we use the stall-flow *sf* and normal-flow *nf* instruction classes to model pipeline-filler instructions, i.e., to model all other instructions than the spoiler and victim. The difference between the stall- and normal-flow operation modes is that an *sf*-class instruction in a stage $s' \in \mathbb{S}$ causes all pipeline stages $s \in \mathbb{S}$ s.t. $s < s'$ to be stalled. In other words, an instruction stays in a state $q = \langle \kappa, s\rangle \in Q^h$ whenever q appears in a configuration of the pipeline containing an earlier instruction in the stall-flow operation mode. Formally, $\forall q = \langle \kappa, s\rangle,\ q' = \langle sf, s'\rangle \in Q^h$:

$$(\exists_{\leftarrow}\colon \{q'\} \models q \to q) \in \Delta^h \Leftrightarrow s < s'. \tag{9.29}$$

Including stalls caused by stall-flow instructions is necessary as they may introduce otherwise unreachable configurations of the verified HS P^h. Moreover, since a pipeline stall caused by some filler instruction may occur at any processor cycle, we will always allow random transitions between stall- and normal-flow operation modes of filler instructions in the upcoming explanation.

Next, an instruction in a state $q = \langle \kappa, s\rangle \in \widehat{Q^h}$, $\widehat{Q^h} = \mathbb{K} \times \widehat{\mathbb{S}}$, $\widehat{\mathbb{S}} = \mathbb{S} \setminus \{\max(\mathbb{S})\}$, is cancelled, i.e., yields a transition $q \to \langle \kappa', s+1\rangle$, $\kappa' \in \{nf, sf\}$, provided that q appears in a configuration of the pipeline in which there exist instructions in states q_1 and q_2 that cause the stage $s+1$ to be cleared. More formally, $\forall q = \langle \kappa, s\rangle \in \widehat{Q^h}$, $\forall q_1, q_2 \in Q^h$, $\forall \kappa' \in \{nf, sf\}$:

$$\begin{aligned}(\exists_{\leftrightarrow}\colon \{q_1, q_2\} \models q \to \langle \kappa', s+1\rangle) \in \Delta^h \Leftrightarrow \\ q_1 \xleftrightarrow{\text{cl},h,s+1} q_2 \wedge \neg\Big(q_1 \xleftrightarrow{\text{st},h,s} q_2\Big).\end{aligned} \tag{9.30}$$

Note that for a successful clearing of an instruction in the stage s, it is also required that s is not stalled at the same time.

For the case when our over-approximating abstraction allows two states q and q' that are conflicting to be reached in a single configuration of the transition system induced by the HS P^h, we introduce the following solution to reduce the number of possible false alarms. Namely, we kill the instruction that entered the pipeline later assuming that this instruction is in the state $q = \langle \kappa, s\rangle$, i.e., we introduce the transition $q \to \langle \kappa', s+1\rangle$, $\kappa' \in \{nf, sf\}$, into Δ^h. Formally, $\forall q = \langle \kappa, s\rangle \in \widehat{Q^h}$, $\forall q' \in Q^h$, $\forall \kappa' \in \{nf, sf\}$:

$$(\exists_{\leftarrow}\colon \{q'\} \models q \to \langle \kappa', s+1\rangle) \in \Delta^h \Leftrightarrow \langle \kappa, s\rangle \xleftrightarrow{\text{cf},h} q'. \tag{9.31}$$

As for the possibility of new instructions entering the pipeline, only the left-most instruction in a given configuration that has so far not entered the pipeline is allowed to enter it. Moreover, new instructions cannot enter the first stage if it is stalled. More precisely, $\forall q = \langle \kappa, \bot \rangle$, $q' = \langle \kappa', \bot \rangle$, q_1, $q_2 \in Q^h$:

$$(\exists_{\leftarrow}\colon \{q'\} \models q \to q) \in \Delta^h, \tag{9.32}$$

$$(\exists_{\leftrightarrow}\colon \{q_1, q_2\} \models q \to q \in \Delta^h) \Leftrightarrow q_1 \xleftrightarrow{\mathrm{st},h,1} q_2. \tag{9.33}$$

Next, an instruction can proceed to the next stage iff none of the above rules is applicable. To model this fact, we use local transitions, building on that we define all global transitions (used above) to be of a higher probability than the local ones. Further, we add transitions reflecting that once finalized instructions stay in their final state forever. More rigorously, $\forall \langle \kappa, s \rangle \in \widehat{Q^h}$:

$$(\langle \kappa, s \rangle \to \langle \kappa, s+1 \rangle) \in \Delta^h, \tag{9.34}$$

$$(\langle \kappa, \bot \rangle \to \langle \kappa, 1 \rangle) \in \Delta^h, \tag{9.35}$$

$$(\langle \kappa, \max(\mathbb{S}) \rangle \to \langle \kappa, \top \rangle) \in \Delta^h, \tag{9.36}$$

$$(\langle \kappa, \top \rangle \to \langle \kappa, \top \rangle) \in \Delta^h. \tag{9.37}$$

To ensure a possibility of the pipeline being stalled by some filler instruction, we allow switching between stall- and normal-flow operation modes. More formally, $\forall \langle sf, s \rangle$, $\langle nf, s \rangle \in \widehat{Q^h}$:

$$(\langle nf, s \rangle \to \langle sf, s+1 \rangle) \in \Delta^h, \tag{9.38}$$

$$(\langle sf, s \rangle \to \langle nf, s+1 \rangle) \in \Delta^h. \tag{9.39}$$

Finally, we recall that apart from the higher priority of global (i.e., guarded) transitions over local (i.e., unguarded) ones, the transition relation Δ^h is constructed under the assumption that, in each step of the transition system induced by the HS P^h, each instruction whose state is a part of the given configuration of P^h must make a step. This is, if we take, e.g., a configuration $q_1 q_2 q_3$ consisting of three states of three instructions, all of the three instructions must synchronously fire some of the above described transitions such that we get the successor configuration $q_1' q_2' q_3'$.

9.5.3 Construction of the Minimal Bad Set

In the previous section, we have constructed a hazard system $P^h = (Q^h, \Delta^h, \alpha^h)$ that models possible interactions of a spoiler and a victim instruction, forming a hazard case $h = (\chi_{sp}, \chi_{vi}) \in \mathbb{X} \times \mathbb{X}$, surrounded by other instructions during a pipelined execution. We now need to be able to check whether some kind of data or control hazard occurs.

To facilitate detection of possible hazards from the constructed HS, we will construct a set B^h of *minimal bad configurations* describing minimal illegal configurations whose reachability (within possibly larger configurations) will mean that the given hazard case h does indeed lead to a hazard. We define the set B^h wrt an extended hazard system $P^h_\top$

Table 9.2: Roles of e-/ℓ-class instructions in hazards cases.

Hazard	e-class	Role	ℓ-class	Role
RAW	writes	spoiler (too slow)	reads	victim
WAR	reads	victim	writes	spoiler (too fast)
WAW	writes	victim	writes	spoiler (too fast)
CTL	writes	spoiler (too slow)	jumps	victim

(defined later in this section), which is obtained by applying four transformations, described also later in the section, on the input system P^h. Since the ordering of instructions within a hazard case is an important factor in the following explanation, we will be speaking about pairs of instruction classes consisting of an e ("earlier") instruction class and an ℓ ("later") instruction class such that either $e = sp \wedge \ell = vi$ or $e = vi \wedge \ell = sp$, meaning that an earlier instruction always enters the pipeline sooner than the later one. For the e and ℓ class instructions, one of the following statements always holds: (a) For RAW and CTL hazards, the e-class instruction is a spoiler that enters the pipeline first and should write data to be read by the later instruction, but it is too slow and the later instruction uses obsolete data. (b) For WAR and WAW hazards, the spoiler is an ℓ-class instruction that enters the pipeline later, but it is too fast and it either destroys data to be read by the earlier instruction (WAR), or it stores its result too early and the result is overwritten by the obsolete result of the earlier instruction (WAW). These scenarios are summarized in Table 9.2.

We are going to build the set B^h such that it will contain so-called *hazard pairs* $q_e^1 q_\ell^1, \ldots, q_e^n q_\ell^n$ of states of the earlier and later instruction such that a hazard described by the hazard case h may occur iff there exists a configuration of the system $P_\top^h$ that contains as a subword some hazard pair from the set B^h and that is reachable from the set of initial configurations I^h. Note, however, that the control states of the earlier/later instructions that signify that something relevant for the hazard has happened (some critical value has been written or read) do not necessarily occur at the same time. On the other hand, hazard pairs consist of pairs of states that should be reached at the same time. To resolve this discrepancy, we will pass information that the critical control state of an instruction has been reached to its successor states. For that, we will introduce several auxiliary notions, which will be introduced such that the detection of the different kinds of hazards may be described in an as uniform way as possible.

We first introduce the *hazard distance* δ that, intuitively, determines the maximum delay (measured in pipeline cycles) with which the later instruction can still cause a hazard. Intuitively, the basis of the distance is the difference in the number of the stages in which the colliding read/write operations happen within the concerned instructions. However, sometimes, this basic difference has to be decreased by one since one of the colliding operations must appear by at least one cycle earlier than the other, while in other cases a hazard appears even when they occur at the same time. More details on that are given below the definition.

Definition 26. The *hazard distance* $\delta\colon \mathbb{X}\times\mathbb{X}\to\mathbb{N}$ is defined as follows for all hazards $h=(\chi_{sp},\chi_{vi})\in\mathbb{X}\times\mathbb{X}$ where $\chi_k=(\pi_k,\tau_k)$ for $k\in\{sp,vi\}$:

$$\delta(h)=\begin{cases}\tau_{sp}^{\mathrm{lst}}-\tau_{vi}^{\mathrm{fst}}-1 & \text{if } h \text{ is a RAW or CTL hazard,}\\ \tau_{vi}^{\mathrm{fst}}-\tau_{sp}^{\mathrm{lst}} & \text{if } h \text{ is a WAR hazard, and}\\ \tau_{vi}^{\mathrm{lst}}-\tau_{sp}^{\mathrm{lst}}-1 & \text{if } h \text{ is a WAW hazard.}\end{cases}$$

Notice that the hazard distance is indeed always non-negative as the definitions of RAW and CTL hazard cases (Definitions 19, 22) imply that $\tau_{vi}^{\mathrm{fst}}<\tau_{sp}^{\mathrm{lst}}$, and the definitions of WAR and WAW hazards (Definitions 20, 21) imply that $\tau_{sp}^{\mathrm{lst}}<\tau_{vi}^{\mathrm{fst}}$ (and, for the case of WAW hazards, one can add the fact that $\tau_{vi}^{\mathrm{fst}}<\tau_{vi}^{\mathrm{lst}}$). For RAW and CTL hazard cases, the distance is decremented by one because reading a value at a cycle when its writing was finished, which is what the corresponding value of τ records (recall that the writing starts one cycle earlier), is safe. On the other hand, in WAR hazards, overwriting the value that is read/written by the earlier instruction at the same time is an error. Finally, WAW hazards are special in that the conflict arises between two write operations where the most extreme case arises when the write operation in the spoiler appears one cycle before the write in the victim: that is why, we have the decrement by one in the formula of WAW hazards. For a further illustration of the notion, see Figure 9.3.

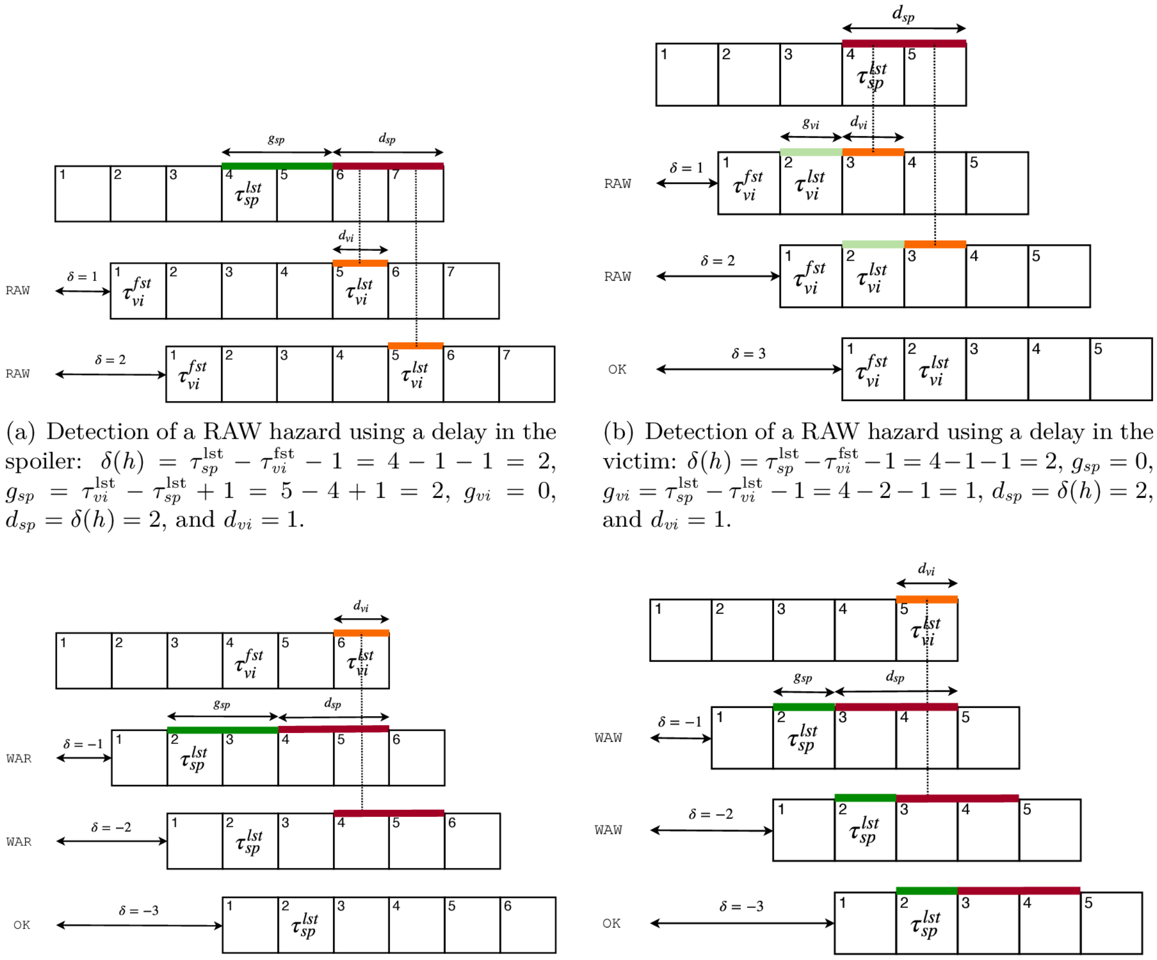
We will next introduce the so-called spoiler/victim gap and detection windows. Intuitively, the *gap window* g_{sp}/g_{vi} of a spoiler/victim instruction ι will tell us for how many cycles one has to wait within the execution of ι, starting from its critical write operation, until the detection of a possible hazard may start. In some cases, the gap will be zero while in some other cases it will be positive. The latter case will happen when the victim/spoiler instruction ι', possibly colliding with ι, has no chance to perform its write operation before the moment when the write operation of ι happens even if ι' starts right after ι. The *detection window* (of size at least one) will then tell us for how many cycles the detection of a possible hazard should be performed within a given instruction after the gap window passes.

In particular, we will define all the windows such that the detection window of victim instructions, denoted d_{vi}, will be fixed to one, i.e., $d_{vi}=1$. Intuitively, the hazard detection will always be performed as soon as the victim instruction writes (and hence "publishes") the wrong data and the gap window of that instruction is over.

The detection window of a spoiler instruction will be possibly longer, in particular, it will correspond to the hazard distance, i.e., $d_{sp}=\delta(h)$ where $h=(\chi_{sp},\chi_{vi})$ is the considered hazard case. The definition of the gap windows must then be done in such a way that any hazard may be detected with the detection windows defined as above, i.e., the detection within the particular instructions must be postponed such that the hazard can always be caught within the detection windows. This definition is more complex and is given below separately for different types of hazards.

Gap Windows for RAW and CTL Hazards

First, notice that $\tau^{\mathrm{fst}}<\tau^{\mathrm{lst}}$ holds for each forward execution $(\pi,\tau)\in\mathbb{X}$ where $\pi^{\mathrm{fst}},\pi^{\mathrm{lst}}\in V_s$. Second, recall that the definitions of RAW and CTL hazard cases (Definitions 19, 22) imply that $\tau_{vi}^{\mathrm{fst}}<\tau_{sp}^{\mathrm{lst}}$. If put together, one can see that there are two possible orderings of τ_{vi}^{fst},



(a) Detection of a RAW hazard using a delay in the spoiler: $\delta(h) = \tau_{sp}^{\text{lst}} - \tau_{vi}^{\text{fst}} - 1 = 4 - 1 - 1 = 2$, $g_{sp} = \tau_{vi}^{\text{lst}} - \tau_{sp}^{\text{lst}} + 1 = 5 - 4 + 1 = 2$, $g_{vi} = 0$, $d_{sp} = \delta(h) = 2$, and $d_{vi} = 1$.

(b) Detection of a RAW hazard using a delay in the victim: $\delta(h) = \tau_{sp}^{\text{lst}} - \tau_{vi}^{\text{fst}} - 1 = 4 - 1 - 1 = 2$, $g_{sp} = 0$, $g_{vi} = \tau_{sp}^{\text{lst}} - \tau_{vi}^{\text{lst}} - 1 = 4 - 2 - 1 = 1$, $d_{sp} = \delta(h) = 2$, and $d_{vi} = 1$.

(c) Detection of a WAR hazard using a delay in the spoiler: $\delta(h) = \tau_{vi}^{\text{fst}} - \tau_{sp}^{\text{lst}} = 4 - 2 = 2$, $g_{sp} = \tau_{vi}^{\text{lst}} - \tau_{vi}^{\text{fst}} = 6 - 4 = 2$, $g_{vi} = 0$, $d_{sp} = \delta(h) = 2$, and $d_{vi} = 1$.

(d) Detection of a WAW hazard using a delay in the spoiler: $\delta(h) = \tau_{vi}^{\text{lst}} - \tau_{sp}^{\text{lst}} - 1 = 5 - 2 - 1 = 2$, $g_{sp} = 1$, $g_{vi} = 0$, $d_{sp} = \delta(h) = 2$, and $d_{vi} = 1$.

Figure 9.3: An illustration of the notions of hazard distance and gap and detection windows used to construct minimal bad sets.

τ_{vi}^{lst}, and τ_{sp}^{lst}:

$$\tau_{vi}^{\text{fst}} \quad < \quad \tau_{sp}^{\text{lst}} \quad \leq \quad \tau_{vi}^{\text{lst}} \tag{9.40}$$

$$\tau_{vi}^{\text{fst}} \quad < \quad \tau_{vi}^{\text{lst}} \quad < \quad \tau_{sp}^{\text{lst}} \tag{9.41}$$

We start with the ordering (9.40), which is illustrated by the scenarios in Fig. 9.3(a). In this case, the spoiler finishes its write operation earlier, and the RAW hazard occurs as soon as the victim performs its write operation. Hence, in order to be able to detect the hazard via states simultaneously reached in the spoiler and the victim, the detection needs to be put off in the spoiler. Provided that the we consider a victim that starts right after the spoiler, $\tau_{vi}^{\text{lst}} - \tau_{sp}^{\text{lst}} + 1$ cycles need to be skipped in the spoiler (including the cycle in which the write operation of the spoiler happens), and so $g_{sp} = \tau_{vi}^{\mathrm{lst}} - \tau_{sp}^{\mathrm{lst}} + 1$.[9] On the other hand, no cycles need to be skipped before the detection starts in the victim, and so $g_{vi} = 0$. Note that the detection of hazards with victims that start later than one cycle behind the spoiler is handled through the detection window d_{sp}.

Next, we consider the ordering (9.41), which is illustrated in Fig. 9.3(b). In this case, the victim performs the write operation first, and the hazard occurs as soon as the spoiler performs its write operation. Hence, this time, the detection needs to be put off in the victim. Using a similar reasoning as above, we define $g_{sp} = 0$ and $g_{vi} = \tau_{sp}^{\mathrm{lst}} - \tau_{vi}^{\mathrm{lst}} - 1$.[10]

Gap Windows for WAR Hazards

For an illustration of the gap and detection windows of WAR hazards, see Fig. 9.3(c). As above, we can use the fact that $\tau^{\mathrm{fst}} < \tau^{\mathrm{lst}}$ holds for each forward execution $(\pi, \tau) \in \mathbb{X}$ where $\pi^{\mathrm{fst}}, \pi^{\mathrm{lst}} \in V_s$. Moreover, the definition of WAR hazards (Definition 20) implies that $\tau_{sp}^{\mathrm{lst}} < \tau_{vi}^{\mathrm{fst}}$. Hence, for WAR hazards, τ_{vi}^{fst}, τ_{vi}^{lst}, and τ_{sp}^{lst} can be ordered as follows only:

$$\tau_{sp}^{\mathrm{lst}} \qquad < \qquad \tau_{vi}^{\mathrm{fst}} \qquad < \qquad \tau_{vi}^{\mathrm{lst}} \tag{9.42}$$

Intuitively, after the spoiler instruction writes, the WAR hazard does not occur until the victim performs its write as well. Unlike for RAW/CTL hazards, we now consider as the base case not the situation when the later instruction starts right after the earlier, but the case when the later instruction starts as late as possible to be still able to cause a hazard, i.e., the case when the spoiler starts $\delta(h)$ cycles after the victim. Then, it is easy to see that the detection needs to be put off by $\tau_{vi}^{\mathrm{lst}} - (\tau_{sp}^{\mathrm{lst}} + \delta(h))$ cycles. Hence, we define $g_{sp} = \tau_{vi}^{\mathrm{lst}} - (\tau_{sp}^{\mathrm{lst}} + \delta(h)) = \tau_{vi}^{\mathrm{lst}} - (\tau_{sp}^{\mathrm{lst}} + \tau_{vi}^{\mathrm{fst}} - \tau_{sp}^{\mathrm{lst}}) = \tau_{vi}^{\mathrm{lst}} - \tau_{vi}^{\mathrm{fst}}$ while $g_{vi} = 0$. The cases of the spoiler that start sooner are then handled appropriately by using the detection window $d_{sp} = \delta(h)$ as also illustrated in Fig. 9.3(c).

Gap Windows in WAW Hazards

As with WAR hazards, for WAW hazards, the ordering between writes given in Equation 9.42 is the only possible. After the spoiler instruction writes, the WAW hazard does not occur until the victim performs its write as well. This cannot happen sooner than after passing through at least one pipeline stage. Therefore, we put the spoiler gap distance equal to one and the victim gap distance equal to zero, i.e., $g_{sp} = 1$ and $g_{vi} = 0$.

Tracking Passage through Gap and Detection Windows

To facilitate tracking whether a spoiler/victim instruction is inside a gap or detection window and, if so, how far inside the window it is, we will introduce a notion of *extended hazard systems* (EHS). In an EHS, each state of the execution of a spoiler/victim instruction will be labelled by a set of tags saying whether the write operation of the spoiler/victim has already happened and, if so, how many cycles have passed since then. The universe of tags $\mathcal{T}$ will therefore include all couples from the set $\{\mathtt{win}_{sp}, \mathtt{win}_{vi}\} \times \mathbb{N}$. The universe of tags is, however, not defined to be equal to the above set since we will need to add some more tags

[9] Intuitively, the addition of 1 is needed since the victim starts by one cycle later. Further, note that the gap is appropriately defined also for the case when $\tau_{sp}^{\mathrm{lst}} = \tau_{vi}^{\mathrm{lst}}$ when a gap window of size 1 is needed to compensate the fact that the victim starts by one cycle later.

[10] The subtraction of 1 comes from that the spoiler starts by one cycle earlier.

into it later on when we examine the effect of stalling of an instruction, which we will need to reflect in the tags as well. We defer the discussion of the stalling-related tags behind we properly explain the basic spoiler/victim tags.

Below, we will introduce the EHSs step-wise by first adding tracking of spoiler windows, then victim windows, and then adding tracking of stalled instructions. This will lead to introduction of EHSs of various levels, with the zero level being the original hazard system, level one being the extension by tracking spoilers, etc.

More formally, for a hazard case $h = (\chi_{sp}, \chi_{vi})$ and the associated HS $P^h = (Q^h, \Delta^h, \alpha^h)$, the corresponding *extended hazard system (EHS) of level* $n \geq 0$ is a tuple $P^h_n = (Q^h_n, \Delta^h_n, \alpha^h_n, \beta^h_n)$ where:

1. Q^h_n is a finite subset of the set $Q^h \times (\mathbb{N} \cup \{\bot, \top\})^n$.[11] We let $Q^h_0 = Q^h$, and we give the precise construction of the set Q^h_n for $n \geq 1$ below. Intuitively, the additional components of the states will allow us to track the passage of the spoiler/victim instructions through the gap and detection windows, for which some states of the original HS will need to be split to multiple occurrences to reflect whether an instruction in that state is in the window and, if so, how far. Moreover, some further splitting will be needed when some of the tracked instructions are stalled some number of times. The finiteness of Q^h_n will stem from that the tracked gap and detection windows are finite, that we are tracking a pair of instructions, and that the stalling can happen for finite time only.

2. The transition relation Δ^h_n and the labelling function α^h_n lift the transition relation Δ^h and the labelling function α^h to the extended set of states. We have $\Delta^h_0 = \Delta^h$ and $\alpha^h_0 = \alpha^h$, and the construction of the relations for $n \geq 1$ is described in detail below.

3. Finally, $\beta^h_n \colon Q^h_n \to 2^{\mathcal{T}}$ is the new tag function. We let $\beta^h_0(q) = \emptyset$ for any $q \in Q^h_0$. For $n \geq 1$, the construction of the function will also be shown below.

For $n \geq 1$, the construction of the EHS P^h_n will be based on applying Alg. 3 and 4 several times on the EHS P^h_0. We start by presenting Alg. 3 that implements a procedure denoted as *window*. This procedure extends the input EHS such that it allows for tracking a spoiler/victim instruction, which performs its critical write instruction w in a state from some given set of states S, through its gap and detection windows whose combined length is k. Here, note that we monitor the gap and detection windows joint into one window which is possible since the latter follows immediately after the former (and we can distinguish in which of the original windows we are by just looking at how deep into the combined window we are).

Intuitively, the algorithm extends all states of the input EHS by one more component that ranges over the set $I := \{\bot, \top, 0, \ldots, k-1\}$. When the additional component is $\bot$, the tracked instruction has not yet entered the gap/detection window. If the additional component i is from the set $\{\bot, \top, 0, \ldots, k-1\}$, the instruction is in the window for $i+1$ cycles. If the additional component is $\top$, the instruction has already got out of the window.

The transition relation is updated straightforwardly such that the monitoring phase can be entered whenever an instruction is in some state from the given set S (and the

[11] For convenience, by a slight abuse of the notation, we let $(Q^h \times (\mathbb{N} \cup \{\bot, \top\})) \times (\mathbb{N} \cup \{\bot, \top\}) = Q^h \times (\mathbb{N} \cup \{\bot, \top\}) \times (\mathbb{N} \cup \{\bot, \top\})$ and $((q, i_1), i_2) = (q, i_1, i_2)$ for any $q \in Q^h$ and $i_1, i_2 \in \mathbb{N} \cup \{\bot, \top\}$, and likewise for higher values of n.

monitoring has not yet started). If the monitoring is started, every executed transition increases the number of cycles spent in the window (recorded in the additional component of states) until the end of the window is reached. Note that, for transitions with guards, the states used in the guards must be lifted to the new set of states, which is done by allowing them to appear with any value of the additional component. Indeed, satisfaction of the guard is not subject to the cycle in which it is reached.

The α function does not depend on the additional component, and so it is lifted to the new set of states by ignoring the additional component. On the other hand, the β function is extended such that states that are inside the monitored window will be tagged by a couple (w, i), which says that the operation w is in the $(i+1)$-th cycle of its gap/detection window.

To be able to compute the set S where the tracking of gap/detection windows starts, which we need to be able to apply Alg. 3, we introduce some further notation. Namely, given a state $q = \langle \kappa, s, i_1, \ldots, i_n \rangle \in Q^h \times (\mathbb{N} \cup \{\bot, \top\})^n$, representing the state of execution of some instruction, we denote by $\mathbb{K}(q) = \kappa$ and $\mathbb{S}(q) = s$ the class and stage of execution of the concerned instruction, respectively. To identify the states where the critical write operations happen and the tracking of the passage of the gap/detection windows starts, we introduce the following function. Namely, given an EHS $P = (Q, \Delta, \alpha, \beta)$ of any level and an instruction class $\kappa \in \mathbb{K}$, we define $wr^P_\kappa \colon \mathbb{X} \to 2^Q$ as the function that maps any execution $(\pi, \tau) \in \mathbb{X}$ to the set $\{q \in Q \mid \mathbb{K}(q) = \kappa \wedge \mathbb{S}(q) = \tau(\pi^{\mathrm{lst}})\}$ of all the states of P where a κ-class instruction makes the write π^{lst} to its target storage in the execution (π, τ).

We can now proceed to the transformation of the original EHS P_0 to the EHS P_1 extended to track the spoiler gap and detection windows. With the above notation and algorithm in hand, the EHS P_1 can be obtained simply as

$$P_1^h := \mathit{window}(P_0^h, wr_{sp}^{P_0^h}(\pi_{sp}, \tau_{sp}), \mathtt{win}_{sp}, g_{sp} + d_{sp}).$$

Indeed, the critical operation is writing in a spoiler, which we denote as $\mathtt{win}_{sp}$. The write operation can happen in one of the states returned by $wr_{sp}^{P_0^h}(\pi_{sp}, \tau_{sp})$. These states thus serve as the initial states for tracking the gap and detection windows. Their sizes are g_{sp} and d_{sp}, respectively, which gives the length $g_{sp} + d_{sp}$ of the combined window whose tracking is ensured in P_1^h by Alg. 3.

The EHS P_2^h extended to track the victim gap and detection windows can be obtained from P_1^h in a very similar way as follows:

$$P_2^h := \mathit{window}(P_1^h, wr_{vi}^{P_1^h}(\pi_{vi}, \tau_{vi})), \mathtt{win}_{vi}, g_{vi} + d_{vi}).$$

An example of a computation of the tracking window is demonstrated in Fig. 9.4.

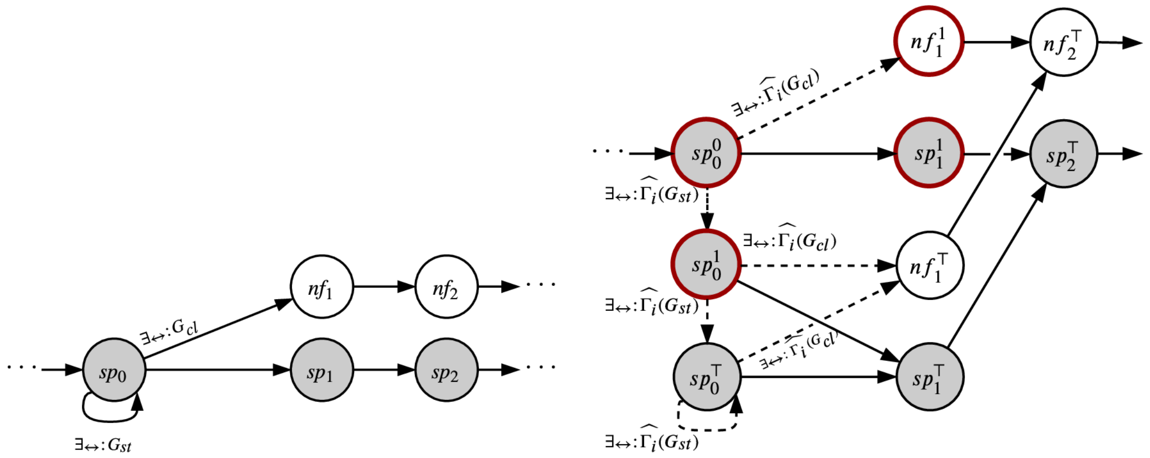
Tracking Windows in Stalled Instructions

Since our approach builds on counting the exact number of cycles spent within the tracking windows, we also need to deal with any scenario when an ℓ-class instruction is stalled while the corresponding e-class instruction is not.[12] This scenario breaks the counting scheme introduced in the previous paragraphs as the later instruction can get delayed and the earlier instruction might get out of the detection window before the later one gets into its detection window. The goal of the following transformations is to compensate such misalignments

[12] The converse cannot happen due to the basic consistency checks that we perform.

Algorithm 3 The *window* procedure transforming an EHS P_n to an EHS P_{n+1} to facilitate tracking of the execution of an instruction that performs a critical write operation w in a state from some given set S through a window of some given length k.

```
Require: An EHS $P_n = (Q_n,\ \Delta_n,\ \alpha_n,\ \beta_n)$ of any level $n \geq 0$, a set $S \subseteq Q_n$ of states to
    start the transformation from, a tag $w \in \{\texttt{win}_{sp}, \texttt{win}_{vi}\}$, and the length of the tracking
    window $k \in \{1, \ldots, \max(\mathbb{S})\}$.
Ensure: An EHS $P_{n+1} = (Q_{n+1}, \Delta_{n+1},\ \alpha_{n+1},\ \beta_{n+1})$ where each state based on $q \in S$
    together with its $k$ reachable successors is tagged by a pair $(w, i)$ where $0 \leq i < k$
    denotes the distance of the successor from the original occurrence of $q$.
$I := \{\bot, \top, 0, \ldots, k-1\}$.
$Q_{n+1} := Q_n \times I$.
$\Delta_{n+1}$ is defined as the minimal relation such that the following two conditions hold:
    (a) For every global transition $\mathbb{Q}_\circ\colon G \models q_1 \to q_2 \in \Delta_n$ and for every injection $\Gamma : Q_n \to I$, the following transitions are in $\Delta_{n+1}$:
        • $\mathbb{Q}_\circ\colon \widehat{\Gamma}(G) \models (q_1, \bot) \to (q_2, \bot)$,
        • $\mathbb{Q}_\circ\colon \widehat{\Gamma}(G) \models (q_1, \bot) \to (q_2, 0)$ if $q_2 \in S$,
        • $\mathbb{Q}_\circ\colon \widehat{\Gamma}(G) \models (q_1, i) \to (q_2, i+1)$ for all $0 \leq i < k-1$,
        • $\mathbb{Q}_\circ\colon \widehat{\Gamma}(G) \models (q_1, i) \to (q_2, \top)$ for $i = k-1$,
        • $\mathbb{Q}_\circ\colon \widehat{\Gamma}(G) \models (q_1, \top) \to (q_2, \top)$
    where $\widehat{\Gamma}\colon 2^{Q_n} \to 2^{Q_{n+1}}$ is defined such that $\forall Q' \subseteq Q_n\colon \widehat{\Gamma}(Q') := \{(q, \Gamma(q)) \mid q \in Q'\}$.
    (b) For every local transition $q_1 \to q_2 \in \Delta_n$, the following transitions are in $\Delta_{n+1}$:
        • $(q_1, \bot) \to (q_2, \bot)$,
        • $(q_1, \bot) \to (q_2, 0)$ if $q_2 \in S$,
        • $(q_1, i) \to (q_2, i+1)$ for all $0 \leq i < k-1$,
        • $(q_1, i) \to (q_2, \top)$ for $i = k-1$,
        • $(q_1, \top) \to (q_2, \top)$.
$\forall (q, i) \in Q_n \times I : \alpha_{n+1}(q, i) = \alpha_n(q)$.
$\forall (q, i) \in Q_n \times \{\bot, \top\} : \beta_{n+1}(q, i) = \beta_n(q)$.
$\forall (q, i) \in Q_n \times \{0, \ldots, k-1\} : \beta_{n+1}((q, i)) = \beta_n(q) \cup \{(w, i)\}$.
```



(a) A part of an EHS $P_n = (Q_n, \Delta_n, \alpha_n, \beta_n)$ modeling the behavior of a spoiler instruction before an application of the *window* procedure. Note that, the spoiler instruction in the state sp_0 might stall (if there are instructions from the set G_{st}), be cleared (if there are instructions from the set G_{cl}), or proceed to the next stage (represented by the state sp_1).

(b) A part of the EHS $P_{n+1} = \mathit{window}(P_n, \{sp_0\}, \mathtt{win}_{sp}, 2)$ that correpons to the same part of P_n depicted in Part (a). States sp_i^j, $0 \leq i \leq 2$, $0 \leq j < 2$, for which $(\mathtt{win}_{sp}, j) \in \beta_n(sp_i^j)$, are highlighted in red. Please note that each global transition from the orignal EHS P_n corresponds to a family of transitions given by all possible injections $\Gamma_1, \ldots, \Gamma_k \colon Q_n \to \{\bot, 0, 1, \top\}$ with the mappings $\widehat{\Gamma}_i$, $1 \leq i \leq k$, defined such that $\forall Q' \subseteq Q_n \colon \widehat{\Gamma}_i(Q') := \{(q, \Gamma_i(q)) \mid q \in Q'\}$. These families of transactions are denoted by the dashed lines in the figure.

Figure 9.4: An illustration of an application of the *window* procedure on a fragment of an EHS P_n.

by (1) using so-called *slack tags* to count how many times the later instruction gets stalled and (2) by expanding the detection window of the earlier instruction correspondingly.

The introduction of slack tags, which are drawn from the set $\{\mathtt{sl}\} \times \mathbb{N}$, is implemented in Alg. 4, which takes us from the EHS P_2^h obtained by the previous transformations to EHS P_3^h as follows:

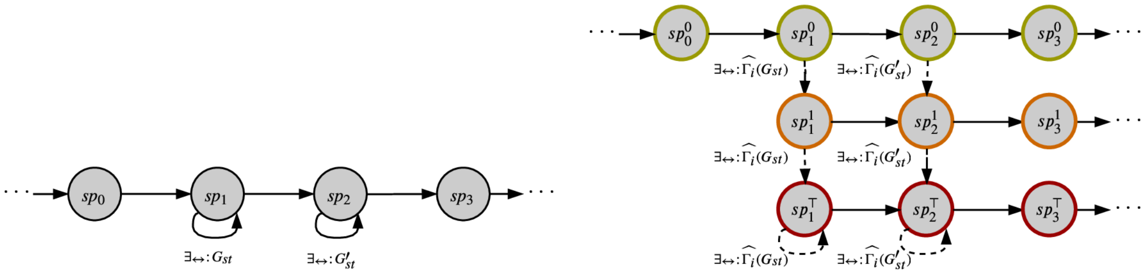
$$P_3^h := \mathit{slack}(P_2^h, \max(\mathbb{S})).$$

Intuitively, all states from the EHS P_2^h are considered to have the initial slack zero. Then, whenever a self-loop on any such state is possible, the self-loop is changed into a transition going to a new copy of the concerned state with the slack being one. More generally, a self-loop on a state with the slack being i is transformed into a transition to a new copy of that state with the slack being $i + 1$ (unless the number of slack steps reaches the maximum number of pipeline stages—going to such a number and beyond is not necessary since such behaviors are ruled out by the initial sanity checks). The number of stalls (slack transitions) performed by an instruction is thus remembered in the structure of the states, and, in addition, we add it into the tags of the states at the end of Alg. 4 so that the slack information is easier to access. An example of an application of the *slack* mapping is demonstrated in Fig. 9.5.

What remains to be done is to adjust the tracking window of the earlier instruction, which has to be done such that the extension corresponds to the number of the slack

Algorithm 4 A procedure for computing the *slack* mapping.

```
Require: An EHS $P_n = (Q_n, \Delta_n, \alpha_n, \beta_n)$ of any level $n \geq 0$ and the total number of pipeline stages $m \geq 1$.
Ensure: An EHS $P_{n+1} = (Q_{n+1}, \Delta_{n+1}, \alpha_{n+1}, \beta_{n+1})$ whose states $P_{n+1}$ are tagged by pairs $(\texttt{sl}, i)$ where $0 \leq i < m$ denotes the number of self-loop transitions taken by the later tracked instruction in the EHS $P_n$.
$I := \{\top, 0, \ldots, m-1\}$.
$Q_{n+1} := Q_n \times I$.
$\Delta_{n+1}$ is defined as the minimal relation such that the following two conditions hold:
    (a) For every global transition $\mathbb{Q}_\circ \colon G \models q_1 \to q_2 \in \Delta_n$ and for every injection $\Gamma : Q_n \to I$, the following transitions are in $\Delta_{n+1}$:
        • $\mathbb{Q}_\circ \colon \widehat{\Gamma}(G) \models (q_1, i) \to (q_2, i+1)$ if $q_1 = q_2$ for all $0 \leq i < m-1$,
        • $\mathbb{Q}_\circ \colon \widehat{\Gamma}(G) \models (q_1, i) \to (q_2, i)$ if $q_1 \neq q_2$ for all $0 \leq i < m$,
        • $\mathbb{Q}_\circ \colon \widehat{\Gamma}(G) \models (q_1, i) \to (q_2, \top)$ if $q_1 = q_2$ and $i = m-1$,
        • $\mathbb{Q}_\circ \colon \widehat{\Gamma}(G) \models (q_1, \top) \to (q_2, \top)$
    where $\widehat{\Gamma} \colon 2^{Q_n} \to 2^{Q_{n+1}}$ is defined such that $\forall Q' \subseteq Q_n \colon \widehat{\Gamma}(Q') := \{(q, \Gamma(q)) \mid q \in Q'\}$.
    (b) For every local transition $q_1 \to q_2 \in \Delta_n$, the following transitions are in $\Delta_{n+1}$:
        • $(q_1, i) \to (q_2, i+1)$ if $q_1 = q_2$ for all $0 \leq i < m-1$,
        • $(q_1, i) \to (q_2, i)$ if $q_1 \neq q_2$ for all $0 \leq i < m$,
        • $(q_1, i) \to (q_2, \top)$ if $q_1 = q_2$ and $i = m-1$,
        • $(q_1, \top) \to (q_2, \top)$.
$\forall (q, i) \in Q_n \times I : \alpha_{n+1}(q, i) = \alpha_n(q)$.
$\forall (q, i) \in Q_n \times \{\top\} : \beta_{n+1}(q, i) = \beta_n(q)$.
$\forall (q, i) \in Q_n \times \{0, \ldots, m-1\} : \beta_{n+1}((q, i)) = \beta_n(q) \cup \{(\texttt{sl}, i)\}$.
```


sp_0
sp_1
sp_2
sp_3
$\exists_{\leftrightarrow}: G_{st}$
$\exists_{\leftrightarrow}: G'_{st}$
sp_0^0
sp_1^0
sp_2^0
sp_3^0
sp_1^1
sp_2^1
sp_3^1
$sp_1^\top$
$sp_2^\top$
$sp_3^\top$
$\exists_{\leftrightarrow}: \widehat{\Gamma}_i(G_{st})$
$\exists_{\leftrightarrow}: \widehat{\Gamma}_i(G'_{st})$
$\exists_{\leftrightarrow}: \widehat{\Gamma}_i(G_{st})$
$\exists_{\leftrightarrow}: \widehat{\Gamma}_i(G'_{st})$
$\exists_{\leftrightarrow}: \widehat{\Gamma}_i(G_{st})$
$\exists_{\leftrightarrow}: \widehat{\Gamma}_i(G'_{st})$

(a) A part of an EHS P_n modeling the behavior of a spoiler instruction before an application of the *slack* mapping. Note that the spoiler instruction in the states sp_1 and sp_2 might be stalled (if there are instructions from the set G_{st}, resp. G'_{st}).

(b) A part of an EHS $P_{n+1} = \mathit{slack}(P_n, 2)$ that correponds to the same part of P_n depicted in Part (a). States sp_i^j, $0 \leq i \leq 3$, $0 \leq j < 2$, for which $(\mathtt{sl}, j) \in \beta_n(sp_i^j)$ with the same value of j, indicating that the instructions passed the same number of self-loops, share the same color. Please note that each global transition from the orignal EHS P_n corresponds to a family of transitions given by all possible injections $\Gamma_1, \ldots, \Gamma_k \colon Q_n \to \{\bot, 0, 1, \top\}$ with the mappings $\widehat{\Gamma}_i$, $1 \leq i \leq k$, defined such that $\forall Q' \subseteq Q_n \colon \widehat{\Gamma}_i(Q') := \{(q, \Gamma_i(q)) \mid q \in Q'\}$. These families of transactions are denoted by the dashed lines in the figure.

Figure 9.5: An illustration of an application of the *slack* mapping on a fragment of an EHS P_n.

transitions taken by the later instruction. For that, we will again use the *window* procedure from Alg. 3, but we will instruct it to add special tags of the form $\mathtt{win}^{(i)}_{vi/sp}$ meaning that the tracking window of the earlier instruction is extended by i cycles. The definition of the bad configurations will then match states of the earlier instruction tagged by $\mathtt{win}^{(i)}_{vi/sp}$ with $\mathtt{win}_{sp/vi}$-tagged states of the later instruction that are at the same time tagged by such $\mathtt{sl}$ tags which show that the later instruction went through i slack transitions more than the earlier one.

To be able to formalize the above, we need to be able to distinguish whether the earlier instruction of a hazard case h is a spoiler or a victim. For that, we define the following notation: $\kappa(e, h) = sp$ provided that h is a RAW or CTL hazard and $\kappa(e, h) = vi$ provided that h is a WAR or WAW hazard. Likewise, for later use, we define the analogous notation for the later instruction too: $\kappa(\ell, h) = vi$ provided that h is a RAW or CTL hazard and $\kappa(\ell, h) = sp$ provided that h is a WAR or WAW hazard.

With all the notation at hand, it is now easy to derive the EHSs P^h_{3+i} of levels $3 + i$ for $1 \leq i \leq m$ with $m = \max(\mathbb{S})$ being the maximum number of pipeline stages that extend the tracking window of the earlier instruction by i cycles. Let $\kappa = \kappa(e, h)$. For i iterating from 1 to m, we get

$$P^h_{3+i} := \mathit{window}(P^h_{3+i-1}, \mathit{wr}_\kappa^{P^h_{3+i-1}}(\pi_\kappa, \tau_\kappa), \mathtt{win}^{(i)}_\kappa, g_\kappa + d_\kappa + i).$$

Finally, we put $P^h_\top := P^h_{3+m}$.

Initial and Bad Configurations

Above, we have finished the construction of the EHS $P^h_\top$ designed to facilitate the construction of the set B^h of minimal bad configurations describing minimal illegal configurations whose reachability (within possibly larger configurations) will mean that the given hazard case h does indeed lead to a hazard. It now remains to define the set B^h along with the corresponding set of initial configurations between which reachability will have to be checked.

We first define the regular set I^h of initial configurations of $P^h_\top$ that consists solely of instructions in the state $\bot$, i.e., before entering the pipeline. An initial configuration may be of an arbitrary length, and it may contain exactly one spoiler *sp* and one victim instruction *vi*, interleaved by any other instructions in any order, modeled using the *nf* class. Formally, the set I^h of the initial states of EHS $P^h_\top$ is defined as follows

$$I^h := I^h_1 \cup I^h_2$$

where

$$I^h_1 := \{\langle nf, \bot\rangle\}^*\{\langle vi, \bot\rangle\}\{\langle nf, \bot\rangle\}^*\{\langle sp, \bot\rangle\}\{\langle nf, \bot\rangle\}^*$$

and

$$I^h_2 := \{\langle nf, \bot\rangle\}^*\{\langle sp, \bot\rangle\}\{\langle nf, \bot\rangle\}^*\{\langle vi, \bot\rangle\}\{\langle nf, \bot\rangle\}^*.$$

Next, we define the set B^h of minimal bad configurations that describe hazardous configurations. The main challenge behind the construction of B^h is to correctly match detection states of the earlier and later instructions. For that, we will use the tracking mechanism that we have provided by the $\mathtt{win}_{sp/vi}$ tags. Namely, we will construct B^h to include all configurations that contain any pair of states $q_e, q_\ell \in Q^h_\top$, $\mathbb{K}(q_e) = \kappa(e,h)$, $\mathbb{K}(q_\ell) = \kappa(\ell,h)$, where the $\mathtt{win}$ tags correspond to the detection part of the tracking window, i.e.,

$$\beta^h_\top(q_e) \in \{(\mathtt{win}_{\kappa(e,h)}, i) \mid g_{\kappa(e,h)} \le i < g_{\kappa(e,h)} + d_{\kappa(e,h)}\}$$

and

$$\beta^h_\top(q_\ell) \in \{(\mathtt{win}_{\kappa(\ell,h)}, i) \mid g_{\kappa(\ell,h)} \le i < g_{\kappa(\ell,h)} + d_{\kappa(\ell,h)}\}.$$

It now remains to deal with situations when some of the instructions are stalled. This is monitored using the $\mathtt{sl}$ tags. First, we can observe that we do not have to further elaborate cases when both (earlier and later) instructions are stalled together. Clearly, any hazard that would occur after these cases would also occur in the case when the instructions are not stalled. Second, the case when the earlier instruction is stalled while the later is not is excluded by the consistency of the pipeline. Therefore, it suffices to only consider those states of the earlier instruction q_e for which $(\mathtt{sl}, 0) \in \beta(q_e)$. Next, let i be a counter that increases each time the later instruction is stalled while the earlier one is not. Since the consistency Rules 1–4 from Section 9.3.2 guarantee that each instruction leaves the pipeline in a final number of steps, the value of the counter i may only range from 0 to $\max(\mathbb{S})$. Every time the counter i is increased, the detection in the earlier instruction is postponed by a single pipeline cycle.

Taken all together, the set B^h of minimal bad configurations describing hazardous configurations is defined as

$$B^h := \bigcup_{i=0}^{\max(\mathbb{S})} B^h_i \tag{9.43}$$

where

$$\begin{aligned} B_i^h := \{q_e q_\ell \mid {} & \{(\mathtt{sl}, 0), (\mathtt{win}^{(i)}_{\kappa(e,h)}, i + j)\} \subseteq \beta^h_\top(q_e) \wedge \\ & \{(\mathtt{sl}, i), (\mathtt{win}_{\kappa(\ell,h)}, k)\} \subseteq \beta^h_\top(q_\ell) \wedge \\ & g_{\kappa(e,h)} \leq j < g_{\kappa(e,h)} + d_{\kappa(e,h)} \wedge \\ & g_{\kappa(\ell,h)} \leq k < g_{\kappa(\ell,h)} + d_{\kappa(\ell,h)} \wedge \\ & q_e, q_\ell \in Q^h_\top\}. \end{aligned} \tag{9.44}$$

With the EHS $P^h_\top$ and the sets of initial I^h and minimal bad configurations B^h at hand, checking whether the hazard h is feasible reduces to checking whether there is some configuration in B^h that is reachable from some configuration in I^h, for which one can use techniques described, e.g., in [3, 17].

9.6 Experimental Evaluation

We have implemented the above described method in a prototype tool called Hades [33]. Hades is written in C++ combined with Python and consists of several components depicted in Figure 9.6. The tool first reads an RTL description of the processor to be verified and converts it into its internal PSG representation. Currently, Hades supports the RTL format expressed in CodAL which is an architectural description language used in the processor design IDE [1]. 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.

The obtained PSG representation is then normalised and simplified. This step includes, for instance, a replacement of conditional branching by multiplexors, an application of value propagation, and a removal of redundant nodes and edges. The normalisation is done using an internal component of Hades called as the *RTL query engine* (RQE), which allows one to search for data-paths and substitute parts of the microprocessor RTL design described via a PSG. Subsequently, pipeline stages are identified by the data-flow analysis discussed in Section 9.3.1. Next, pipeline consistency is checked using Rules 1–4 from Section 9.3.2 by an SMT solver for bit-vector logic. Hades is compatible with all SMT solvers accepting the SMT2 formula format. In particular, for the below experiments, Z3 [100] was used. Further, after the PSG is annotated by pipeline stages identified by the data-flow analysis, Hades repeatedly utilizes the RQE and the SMT solver to extract potential hazard cases as described in Section 9.4 and to generate the appropriate hazard systems (HSs) for each hazard case as we have seen in Section 9.5. The generated HSs are then checked using the abstract regular model checker (ARMC) of [17]. The process of evaluation of the inputs and generation of the results by the above mentioned subsystems is orchestrated by the so-called “core” component of Hades.

We have tested the tool on six kinds of processors. The first four are identical to the ones already presented in Section 7.6. *CompAcc* is then an 8-bit processor based on an accumulator architecture with a very similar structure as the one shown in Fig. 9.2. Finally, *DLX5* is a 5-staged 32-bit processor able to execute a subset of the instruction set of the DLX architecture [108] (with no floating point instructions).

We consider multiple variants of the above introduced processors, which gives us 17 unique test cases in total. In particular, the variants of the particular processors differ in the following aspects: (i) the way how data hazards are avoided (pipeline stalling and

Table 9.3: Experimental results.

Processor /		Simpl.	Data Flow	Consistency			Parametric System				Total	Hazard
Variant		Time [s]	Analysis [s]	Checking [s]			Generation and Verification [s]				Time [s]	Cases [#]
				rqe	smt	core	rqe	smt	armc	core		
TinyCPU	S	0.03	0.01	<0.01	0.43	0.23	0.01	1.21	16.17	1.83	19.92	6
	SA	0.03	0.01	<0.01	0.61	0.28	0.06	8.71	114.04	14.54	138.28	20
	B	0.03	0.01	<0.01	0.57	0.24	0.02	1.62	16.93	2.76	22.18	7
	BA	0.04	0.01	<0.01	0.67	0.31	0.05	5.38	43.86	11.67	61.99	12
	SF	0.03	0.01	<0.01	0.46	0.24	0.04	6.50	67.45	9.19	83.92	29
	SFA	0.04	0.01	<0.01	0.64	0.30	0.13	19.95	221.43	32.95	275.45	42
SPP8	S	0.10	0.02	<0.01	0.61	0.34	0.04	5.97	43.27	8.73	59.08	29
	B	0.09	0.01	<0.01	0.70	0.42	0.05	6.66	43.24	13.48	64.65	29
SPP16	S	0.10	0.03	0.01	0.85	0.52	0.04	6.00	43.41	9.04	60.00	29
	B	0.11	0.03	0.01	0.90	0.53	0.04	6.59	43.19	13.80	65.20	29
Codea2	SF	0.24	0.07	0.01	1.17	0.53	0.42	80.60	339.33	115.68	538.05	243
CompAcc	SFA	0.10	0.02	0.01	1.00	0.53	0.15	30.89	323.27	33.01	388.98	44
	BFA	0.10	0.02	0.01	1.10	0.55	0.20	36.86	350.14	45.62	434.60	59
DLX5	S	0.13	0.04	0.01	1.95	0.97	0.13	26.92	243.66	38.82	312.63	27
	SA	0.15	0.05	0.01	2.03	1.01	0.57	95.50	521.91	182.09	803.32	95
	B	0.18	0.05	0.01	2.16	1.05	0.16	62.95	243.6	160.89	471.05	27
	BA	0.19	0.06	0.01	1.98	1.03	0.28	101.02	376.75	469.77	951.09	62

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

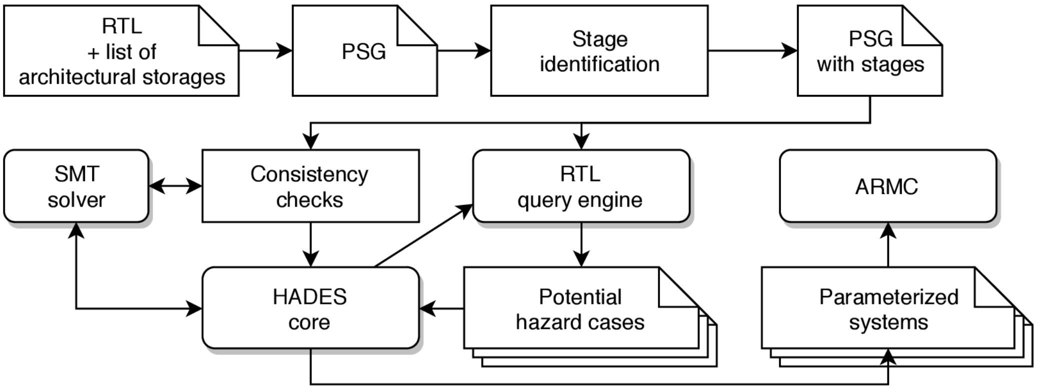


Figure 9.6: A schematic of the Hades verification tool.

clearing or data bypassing), (ii) the presence of flag/status registers, and (iii) utilization of the auto-increment logic.

We conducted a series of experiments on a PC with Intel Xeon E5-2630 v2 @2.60GHz and 32 GB RAM with results shown in Table 9.3. 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 parametric systems that are created for each hazard case. The times are split to the times consumed by the different parts of the tool's architecture.

The following column gives the overall verification time, which remains in the order of minutes even for complex designs. Moreover, the tool also scales well with the growing size of the processor data-path 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 data and control hazard cases that had to be generated and checked. Note that each hazard case represents a separate task so the part of generation and verification of the parametric systems 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. Our approach also correctly identified all potential control hazards that are supposed to be handled by the compiler (by explicitly generating series of `NOP` instructions after a conditional branch).

9.7 Conclusion

We have presented an approach that harnesses methods for formal verification of parametric systems in order to discover incorrectly handled data and control pipeline hazards in the RTL implementation of pipeline-based execution. 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.

A potential future work may include extension of the proposed approach to support microprocessors equipped with multiple pipelines. Further, as we have already mentioned in Section 9.4, another considerable topic is extending the approach so it can detect issues caused by spoilers and/or victims that consist of multiple instructions.

Chapter 10

Epilogue

The subject of the thesis was to design new verification techniques based on formal approaches that are optimized for use in the process of concurrent development of hardware and software, the so-called HW/SW co-design.

In accordance with the set-up goals, the thesis firstly presented a novel technique for dealing with memory modeling that can be used for efficient formal verification of hardware designs. The approach can accommodate different data sizes such as quad words, double words, words, or bytes. At the same time, it is also applicable to memories with multiple read and write ports and memories with read and write operations with zero- or single-clock delay. The memory is allowed to start with a random initial state permitting one to formally verify the given design for all initial contents of the memory. An abstraction used in the approach represents large register-files and memories in a way that dramatically reduces the state space explored during formal verification of microprocessors as can be witnessed by our experiments presented in Chapter 7.

Further, in Chapter 8, the thesis presents the correspondence checking approach based on the idea of utilizing bounded model checking to compare the outputs produced by automatically derived RTL and ISA models of a given processor for all possible instructions and their inputs. To guarantee that results are obtained in a given time limit, each instruction is checked in parallel for several bit-widths of its input. The approach then returns only the result of the verification task with maximal bit-width that finished within the time limit. Our experiments included a non-trivial single-pipelined processor in which, during its development, the approach revealed three previously unknown bugs confirmed by the developers. The experiments have also shown that vast majority of instructions of single-pipelined microprocessors, typically used within embedded devices, can be verified within seconds.

Finally, in Chapter 9, the thesis presents an approach that harnesses methods for formal verification of parametric systems in order to discover incorrectly handled data and control pipeline hazards in the RTL implementations of pipeline-based executions. The approach was developed with the aim to be highly automated, requiring no external information about the design (apart from specifying the architectural registers). The experimental implementation of the approach was successfully tested on several non-trivial microprocessors where the approach was able to discover a previously unknown flaw caused by an unhandled hazard.

The design of all the above-presented approaches was motivated by the general idea of splitting processor verification into several simpler, more specialized tasks. Moreover, each approach was designed to be highly automated, requiring minimal additional effort from developers.

Bibliography

[1] Codasip Studio for Rapid Processor Development. www.codasip.com. 2019.

[2] Aagaard, M. D.: A Hazards-Based Correctness Statement for Pipelined Circuits. In *Proc. of Correct Hardware Design and Verification Methods (CHARME)*, *LNCS*, vol. 2860. Springer. 2003. pp. 66–80.

[3] Abdulla, P. A.; Haziza., F.; Holík, L.: All for the Price of Few (Parameterized Verification through View Abstraction). In *Proc. of Verification, Model Checking, and Abstract Interpretation (VMCAI)*, *LNCS*, vol. 7737. Springer. 2013. pp. 476–495.

[4] Abdulla, P. A.; Jonsson, B.; Nilsson, M.; et al.: A Survey of Regular Model Checking. In *CONCUR 2004 - Concurrency Theory*, edited by P. Gardner; N. Yoshida. Berlin, Heidelberg: Springer Berlin Heidelberg. 2004. pp. 35–48. doi:10.1007/978-3-540-28644-8_3.

[5] Accellera: *Standard Universal Verification Methodology Class Reference, Release 1.2*. 2014.
Retrieved from: workspace.accellera.org/downloads/standards/uvm

[6] Alencar, R.; Rigo, S.; Azevedo, R.: Software Co-Verification Based on Program Traces from Different Processors. In *In 3rd Workshop on Infrastructures for Software/Hardware Co-design (WISH)*. 2011. pp. 1–6.

[7] AMD: *AMD64 Architecture Programmer's Manual, Volume 3*. 2018.
Retrieved from: www.amd.com/system/files/TechDocs/24594.pdf

[8] ARM: *Arm Instruction Set Reference Guide*. 2018.
Retrieved from: static.docs.arm.com/100076/0100/arm_instruction_set_reference_guide_100076_0100_00_en.pdf

[9] Baier, C.; Katoen, J.: *Principles of model checking*. MIT Press. 2008. ISBN 978-0-262-02649-9.

[10] Barrett, C.; Sebastiani, R.; Seshia, S. A.; et al.: Satisfiability Modulo Theories. In *Handbook of Satisfiability*, vol. 4, edited by A. Biere; H. van Maaren; T. Walsh. chapter 8. IOS Press. 2009.

[11] Basu, S.; Moona, R.: High level synthesis from Sim-nML processor models. In *16th International Conference on VLSI Design, 2003. Proceedings.*. Jan 2003. ISSN 1063-9667. pp. 255–260. doi:10.1109/ICVD.2003.1183146.

[12] Bayardo, R. J., Jr.; Schrag, R. C.: Using CSP Look-back Techniques to Solve Real-world SAT Instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*. AAAI'97/IAAI'97. AAAI Press. 1997. ISBN 0-262-51095-2. pp. 203–208.

[13] Beyer, S.; Jacobi, C.; Kröning, D.; et al.: Putting it all together – Formal verification of the VAMP. *International Journal on Software Tools for Technology Transfer*. vol. 8, no. 4. Aug 2006: pp. 411–430. ISSN 1433-2787. doi:10.1007/s10009-006-0204-6.

[14] Biere, A.; Cimatti, A.; Clarke, E.; et al.: Symbolic Model Checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, edited by W. R. Cleaveland. Springer Berlin Heidelberg. 1999. ISBN 978-3-540-49059-3. pp. 193–207.

[15] Biere, A.; Heljanko, K.; Wieringa, S.: AIGER 1.9 And Beyond. Technical report. FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria. 2011.

[16] Bouajjani, A.; Habermehl, P.; Rogalewicz, A.; et al.: Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In *Proc. of 13th International Static Analysis Symposium (SAS)*, *LNCS*, vol. 4134. Springer. 2006. pp. 52–70.

[17] Bouajjani, A.; Habermehl, P.; Vojnar, T.: Abstract Regular Model Checking. In *Proc. of 16th International Conference on Computer Aided Verification (CAV)*, *LNCS*, vol. 3114. Springer. 2004. pp. 197–202.

[18] Bradley, A. R.: SAT-Based Model Checking without Unrolling. In *Verification, Model Checking, and Abstract Interpretation*, edited by R. Jhala; D. Schmidt. Berlin, Heidelberg: Springer Berlin Heidelberg. 2011. ISBN 978-3-642-18275-4. pp. 70–87. doi:10.1007/978-3-642-18275-4_7.

[19] Bradley, A. R.; Manna, Z.; Sipma, H. B.: What's Decidable About Arrays? In *Proc. of Verification, Model Checking, and Abstract Interpretation (VMCAI)*, *LNCS*, vol. 3855, edited by K. S. Emerson, E. Allenand Namjoshi. Berlin, Heidelberg: Springer Berlin Heidelberg. 2006. ISBN 978-3-540-31622-0. pp. 427–442. doi:10.1007/11609773_28.

[20] Brown, D.; Levine, J.; Mason, T.: *Lex & Yacc*. O'Reilly Media. 1992. ISBN 978-1565920002.

[21] Brummayer, R.; Biere, A.: 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)*, *LNCS*, vol. 5505. Springer. 2009. pp. 174–177.

[22] Bryant, R. E.: Symbolic Boolean Manipulation with Ordered Binary-decision Diagrams. *ACM Computing Surveys*. vol. 24, no. 3. 1992: pp. 293–318. ISSN 0360-0300. doi:10.1145/136035.136043.

[23] Bryant, R. E.: Formal Verification of Pipelined Y86-64 Microprocessors with UCLID5. Technical Report CMU-CS-18-122. 2018.

[24] Bryant, R. E.; German, S.; Velev, M. N.: Exploiting Positive Equality in a Logic of Equality with Uninterpreted Functions. In *Computer Aided Verification*, edited by N. Halbwachs; D. Peled. Springer Berlin Heidelberg. 1999. ISBN 978-3-540-48683-1. pp. 470–482.

[25] Bryant, R. E.; Velev, M. N.: Verification of pipelined microprocessors by comparing memory execution sequences in symbolic simulation. In *Proc. of Advances in Computing Science (ASIAN)*, *LNCS*, vol. 1345, edited by K. Shyamasundar, R. K.and Ueda. Berlin, Heidelberg: Springer Berlin Heidelberg. 1997. ISBN 978-3-540-69658-2. pp. 18–31. doi:10.1007/3-540-63875-X_40.

[26] Burch, J. R.; Clarke, E. M.; McMillan, K. L.; et al.: Symbolic Model Checking: 10^{20} States and Beyond. In *Proc. of Fifth Annual IEEE Symposium on Logic in Computer Science*. IEEE. 1990. ISBN 0-8186-2073-0. pp. 428–439. doi:10.1109/LICS.1990.113767.

[27] Burch, J. R.; Dill, D. L.: Automatic Verification of Pipelined Microprocessor Control. In *Proc. of Computer Aided Verification (CAV)*, *LNCS*, vol. 818. Springer. 1994. ISBN 978-3-540-48469-1. pp. 68–80.

[28] Cadence: *Tensilica Software Development Toolkit (SDK)*. 2014.
Retrieved from: ip.cadence.com/uploads/103/SWdev-pdf

[29] Cadence: *TIE Language — The Fast Path to High-Performance Embedded SoC Processing*. 2016.
Retrieved from: ip.cadence.com/uploads/980/TIP_WP_TIE_FINAL-pdf

[30] Cadence: *Xtensa LX7 Processor Datasheet*. 2016.
Retrieved from: ip.cadence.com/uploads/1099/TIP_PB_Xtensa_lx7_FINAL-pdf

[31] Charvát, L.; Smrčka, A.; Vojnar, T.: Automatic Formal Correspondence Checking of ISA and RTL Microprocessor Description. In *Proc. of Microprocessor Test and Verification (MTV'12)*. IEEE. 2012. pp. 6–12.

[32] Charvát, L.; Smrčka, A.; Vojnar, T.: An Abstraction of Multi-Port Memories with Arbitrary Addressable Units. In *Proc. of Computer Aided Systems Theory (EUROCAST'13)*, *LNCS*, vol. 8111. Springer. 2013. pp. 460–468.

[33] Charvát, L.; Smrčka, A.; Vojnar, T.: Hades Hades Hardware Verification Tool. www.fit.vutbr.cz/research/groups/verifit/tools/hades/. 2014.

[34] Charvát, L.; Smrčka, A.; Vojnar, T.: Using Formal Verification of Parameterized Systems in RAW Hazard Analysis in Microprocessors. In *Proc. of Microprocessor Test and Verification (MTV'14)*. IEEE. 2014. ISBN 978-1-4673-6858-2. pp. 83–89.

[35] Charvát, L.; Smrčka, A.; Vojnar, T.: Using Formal Verification of Parameterized Systems in RAW Hazard Analysis in Microprocessors. Technical Report FIT-TR-2014-04. Brno University of Technology. 2014.

[36] Charvát, L.; Smrčka, A.; Vojnar, T.: Microprocessor Hazard Analysis via Formal Verification of Parameterized Systems. In *Proc. of Computer Aided Systems Theory (EUROCAST'15)*, *LNCS*, vol. 9520. Springer. 2015. pp. 605–614.

[37] Charvát, L.; Smrčka, A.; Vojnar, T.: HADES: Microprocessor Hazard Analysis via Formal Verification of Parameterized Systems. In *Proc. of 11th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'16)*. 233. EPTCS. 2016. pp. 87–93. doi:10.4204/EPTCS.233.9.

[38] Chen, M.; Mishra, P.: Property Learning Techniques for Efficient Generation of Directed Tests. *IEEE Transactions on Computers*. vol. 60, no. 6. June 2011: pp. 852–864. ISSN 0018-9340. doi:10.1109/TC.2011.49.

[39] Clarke, E.; Grumberg, O.; Jha, S.; et al.: Counterexample-Guided Abstraction Refinement. In *Proc. of Computer Aided Verification (CAV)*, *LNCS*, vol. 1855, edited by E. A. Emerson; A. P. Sistla. Berlin, Heidelberg: Springer Berlin Heidelberg. 2000. ISBN 978-3-540-45047-4. pp. 154–169. doi:10.1007/10722167_15.

[40] Clarke, E.; Grumberg, O.; Minea, M.; et al.: State space reduction using partial order techniques. *International Journal on Software Tools for Technology Transfer*. vol. 2, no. 3. Nov 1999: pp. 279–287. ISSN 1433-2779. doi:10.1007/s100090050035.

[41] Clarke, E.; Talupur, M.; Veith, H.: Environment abstraction for parameterized verification. In *Proc. of Verification, Model Checking, and Abstract Interpretation (VMCAI)*, *LNCS*, vol. 3855. Springer. 2006. pp. 126–141.

[42] Codea2 Core IP in Codasip Studio. www.codasip.com/products/codea2/. 2013.

[43] Dang, T. N.; Roychoudhury, A.; Mitra, T.; et al.: Generating test programs to cover pipeline interactions. In *2009 46th ACM/IEEE Design Automation Conference*. July 2009. ISSN 0738-100X. pp. 142–147.

[44] Davis, M.; Logemann, G.; Loveland, D.: A Machine Program for Theorem-proving. *Communication of the ACM*. vol. 5, no. 7. Jul 1962. ISSN 0001-0782. doi:10.1145/368273.368557.

[45] Een, N.; Mishchenko, A.; Brayton, R.: Efficient Implementation of Property Directed Reachability. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*. FMCAD'11. Austin, TX: FMCAD Inc. 2011. ISBN 978-0-9835678-1-3. pp. 125–134.

[46] Fauth, A.; Knoll, A.: Automated generation of DSP program development tools using a machine description formalism. In *1993 IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 1. Apr 1993. ISSN 1520-6149. pp. 457–460. doi:10.1109/ICASSP.1993.319154.

[47] Fauth, A.; Lohr, F.; Freericks, M.: Sigh/Sim: An environment for retargetable instruction set simulation. Technical Report 1993/43. Technische Universität Berlin, Germany. 1993.

[48] Fey, G.; Drechsler, R.: Design understanding by automatic property generation. In *Proceedings of Workshop on Synthesis And System Integration of Mixed Information technologies*. 2004. pp. 274–281.

[49] Ganai, M. K.; Gupta, A.; Ashar, P.: Verification of embedded memory systems using efficient memory modeling. In *Proc. of Design, Automation and Test in Europe (DATE)*, vol. 2. IEEE. 2005. ISSN 1530-1591. pp. 1096–1101. doi:10.1109/DATE.2005.325.

[50] German, S. M.: A Theory of Abstraction for Arrays. In *Proc. of the International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. FMCAD. 2011. ISBN 978-0-9835678-1-3. pp. 176–185.

[51] Goossens, G.; Lanneer, D.; Geurts, W.; et al.: Design of ASIPs in multi-processor SoCs using the Chess/Checkers retargetable tool suite. In *International Symposium on System-on-Chip*. Nov 2006. ISBN 1-4244-0621-8. pp. 1–4. doi:10.1109/ISSOC.2006.321968.

[52] Gries, M.; Keutzer, K.: *Building ASIPs: The Mescal Methodology*. Springer US. 2005. ISBN 978-0-387-26057-0. doi:10.1007/b136892.

[53] Hadjiyiannis, G.; Devadas, S.: Techniques for accurate performance evaluation in architecture exploration. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*. vol. 11, no. 4. Aug 2003: pp. 601–615. ISSN 1063-8210. doi:10.1109/TVLSI.2003.812290.

[54] Hadjiyiannis, G.; Hanono, S.; Devadas, S.: ISDL: An Instruction Set Description Language for Retargetability and Architecture Exploration. *Design Automation for Embedded Systems*. vol. 6, no. 1. Sep 2000: pp. 39–69. ISSN 1572-8080. doi:10.1023/A:1008937425064.

[55] Hall, E. C.: *Journey to the Moon: The History of the Apollo Guidance Computer*. American Institute of Aeronautics. 1996. ISBN 978-1563471858.

[56] Hao, K.; Ray, S.; Xie, F.: Equivalence Checking for Function Pipelining in Behavioral Synthesis. In *Proc. of Design, Automation and Test in Europe (DATE)*. IEEE. 2014. pp. 1–6.

[57] Harrison, J.: Floating-Point Verification Using Theorem Proving. In *Proceedings of the 6th International Conference on Formal Methods for the Design of Computer, Communication, and Software Systems*. SFM'06. Springer-Verlag. 2006. ISBN 3-540-34304-0, 978-3-540-34304-2. pp. 211–242. doi:10.1007/11757283_8.

[58] Hartoog, M. R.; Rowson, J. A.; Reddy, P. D.; et al.: Generation of Software Tools from Processor Descriptions for Hardware/Software Codesign. In *Proceedings of the 34th Annual Design Automation Conference*. DAC'97. New York, NY, USA: ACM. 1997. ISBN 0-89791-920-3. pp. 303–306. doi:10.1145/266021.266110.

[59] Hautala, I.; Boutellier, J.; Hannuksela, J.; et al.: Programmable Low-Power Multicore Coprocessor Architecture for HEVC/H.265 In-Loop Filtering. *IEEE Transactions on Circuits and Systems for Video Technology*. vol. 25, no. 7. July 2015: pp. 1217–1230. ISSN 1051-8215. doi:10.1109/TCSVT.2014.2369744.

[60] Hopcroft, J. E.; Motwani, R.; Ullman, J. D.: *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.. 2006. ISBN 0321455363.

[61] Hosabettu, R.; Gopalakrishnan, G.; Srivas, M.: Verifying Advanced Microarchitectures that Support Speculation and Exceptions. In *Computer Aided Verification*, edited by E. A. Emerson; A. P. Sistla. Springer Berlin Heidelberg. 2000. ISBN 978-3-540-45047-4. pp. 521–537.

[62] Hunt, W. A.: Microprocessor design verification. *Journal of Automated Reasoning*. vol. 5, no. 4. Dec 1989: pp. 429–460. ISSN 1573-0670. doi:10.1007/BF00243132.

[63] Hunt, W. A.; Kaufmann, M.: A formal model of a large memory that supports efficient execution. In *Proc. of Formal Methods in Computer-Aided Design (FMCAD)*. IEEE. 2012. ISBN 978-0-9835678-2-0. pp. 60–67.

[64] Husár, A.; Trmač, M.; Hranáč, J.; et al.: Automatic C Compiler Generation from Architecture Description Language ISAC. In *6th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*. Masaryk University. 2010. ISBN 978-80-87342-10-7. pp. 84–91.

[65] IEEE: *IEC/IEEE Behavioural Languages – Part 4: Verilog Hardware Description Language*. 2004. doi:10.1109/IEEESTD.2004.95753.

[66] IEEE: *IEEE Standard VHDL Language Reference Manual*. 2009. doi:10.1109/IEEESTD.2009.4772740.

[67] Ienne, P.; Leupers, R.: *Customizable Embedded Processors: Design Technologies and Applications*. Morgan Kaufmann Publishers Inc.. 2007. ISBN 0123695260, 9780080490984.

[68] Intel: *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2*. 2016. Retrieved from: www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf

[69] Jones, R. B.; Seger, C. H.; Dill, D. L.: Self-Consistency Checking. In *Proc. of Formal Methods in Computer-Aided Design (FMCAD)*, *LNCS*, vol. 1166. Springer. 1996. pp. 159–171.

[70] Jones, R. B.; Skakkebæk, J. U.; Dill, D. L.: Formal Verification of Out-of-Order Execution with Incremental Flushing. *Formal Methods in System Design*. vol. 20, no. 2. Mar 2002: pp. 139–158. ISSN 1572-8102. doi:10.1023/A:1014118529369.

[71] Kesten, Y.; Maler, O.; Marcus, M.; et al.: Symbolic model checking with rich assertional languages. *Theoretical Computer Science*. vol. 256, no. 1-2. 2001: pp. 93–112.

[72] Kiesl, B.; Seidl, M.; Tompits, H.; et al.: Local Redundancy in SAT: Generalizations of Blocked Clauses. *Logical Methods in Computer Science*. vol. 14. 2018: pp. 1–23.

[73] Kildall, G. A.: A unified approach to global program optimization. In *In Conf. Rec. 1st Symp. Principles of Prog. Lang. (POPL)*. ACM. 1973. pp. 194—-206.

[74] Koelbl, A.; Burch, J. R.; Pixley, C.: Memory Modeling in ESL-RTL Equivalence Checking. In *Proc. of the 44th Annual Design Automation Conference (DAC)*. ACM. 2007. ISBN 978-1-59593-627-1. pp. 205–209. doi:10.1145/1278480.1278530.

[75] Koelbl, A.; Jacoby, R.; Jain, H.; et al.: Solver technology for system-level to RTL equivalence checking. In *Proc. of Design, Automation Test in Europe Conference Exhibition (DATE)*. IEEE. 2009. ISSN 1530-1591. pp. 196–201. doi:10.1109/DATE.2009.5090657.

[76] Korel, B.; Laski, J.: Dynamic Program Slicing. *Information Processing Letters*. vol. 29, no. 3. Oct 1988: pp. 155–163. ISSN 0020-0190. doi:10.1016/0020-0190(88)90054-3.

[77] Kozen, D. C.: *Automata and Computability*. New York, NY: Springer. 1997. ISBN 978-1-4612-7309-7. doi:10.1007/978-1-4612-1844-9.

[78] Kuhne, U.; Beyer, S.; Bormann, J.; et al.: Automated Formal Verification of Processors Based on Architectural Models. In *Proc. of Formal Methods in Computer-Aided Design (FMCAD)*. IEEE. 2010. pp. 129–136.

[79] Lanneer, D.; Van Praet, J.; Kifli, A.; et al.: *Chess: Retargetable Code Generation for Embedded DSP Processors*. Springer US. 2002. ISBN 978-1-4615-2323-9. pp. 85–102. doi:10.1007/978-1-4615-2323-9_5.

[80] Leupers, R.; Marwedel, P.: Retargetable Code Generation Based on Structural Processor Description. *Design Automation for Embedded Systems*. vol. 3, no. 1. Jan 1998: pp. 75–108. ISSN 1572-8080. doi:10.1023/A:1008807631619.

[81] Levitt, J.; Olukotun, K.: Verifying Correct Pipeline Implementation for Microprocessors. In *Proceedings of the 1997 IEEE/ACM International Conference on Computer-aided Design*. ICCAD '97. IEEE Computer Society. 1997. ISBN 0-8186-8200-0. pp. 162–169.

[82] Liang, J. H.; Oh, C.; Mathew, M.; et al.: Machine Learning-Based Restart Policy for CDCL SAT Solvers. In *Proceedings of Theory and Applications of Satisfiability Testing SAT 2018*, *Lecture Notes in Computer Science*, vol. 10929. Springer. 2018. ISBN 978-3-319-94143-1. pp. 94–110. doi:10.1007/978-3-319-94144-8.

[83] Liblit, B.; Naik, M.; Zheng, A. X.; et al.: Scalable Statistical Bug Isolation. *SIGPLAN Not.*. vol. 40, no. 6. Jun 2005: pp. 15–26. ISSN 0362-1340. doi:10.1145/1064978.1065014.

[84] Manolios, P.; Srinivasan, S. K.; Vroon, D.: Automatic Memory Reductions for RTL Model Verification. In *Proc. of IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE. 2006. ISSN 1092-3152. pp. 786–793. doi:10.1109/ICCAD.2006.320121.

[85] Marques Silva, J. P.; Sakallah, K. A.: GRASP-A new search algorithm for satisfiability. In *Proceedings of International Conference on Computer Aided Design*. 1996. ISBN 0-8186-7597-7. pp. 220–227. doi:10.1109/ICCAD.1996.569607.

[86] Mccarthy, J.: Towards a Mathematical Science of Computation. In *In IFIP Congress*. North-Holland. 1962. pp. 21–28.

[87] McMillan, K. L.: *The SMV System*. Boston, MA: Springer US. 1993. ISBN 978-1-4615-3190-6. pp. 61–85. doi:10.1007/978-1-4615-3190-6_4.

[88] McMurran, M. W.: *ACHIEVING ACCURACY: A Legacy of Computers and Missiles*. Xlibris Corp.. 2008. ISBN 978-1436381062.

[89] Meduna, A.: *Automata and Languages: Theory and Applications*. Berlin, Heidelberg: Springer-Verlag. 2000. ISBN 1-85233-074-0.

[90] Mentor: *The Veloce Strato Platform: Unique Core Components Create High-Value Advantages*. 2019.
Retrieved from: www.mentor.com/products/fv/request?selected=103372

[91] Minařík, M.: *Concurrent Evolutionary Design of Hardware and Software*. PhD. Thesis. Brno University of Technology, Faculty of Information Technology. 2017.

[92] Mishra, P.; Dutt, N.: Architecture description languages for programmable embedded systems. *IEE Proceedings – Computers and Digital Techniques*. vol. 152, no. 3. May 2005: pp. 285–297. ISSN 1350-2387. doi:10.1049/ip-cdt:20045071.

[93] Mishra, P.; Dutt, N. (editors): *Processor Description Languages: Applications and Methodologies*. 2008. ISBN 978-0-12-374287-2. doi:10.1016/B978-0-12-374287-2.X5001-0.

[94] Mishra, P.; Dutt, N.: Specification-driven Directed Test Generation for Validation of Pipelined Processors. *ACM Transactions on Design Automation of Electronic Systems*. vol. 13, no. 3. Jul 2008: pp. 42:1–42:36. ISSN 1084-4309. doi:10.1145/1367045.1367051.

[95] Mishra, P.; Dutt, N.; Dutt, N.; et al.: Modeling and Validation of Pipeline Specifications. *ACM Transactions on Embedded Computing Systems*. vol. 3, no. 1. Feb 2004: pp. 114–139. ISSN 1539-9087. doi:10.1145/972627.972633.

[96] Mishra, P.; Dutt, N. D.: *Functional Verification of Programmable Embedded Architectures: A Top-Down Approach*. Springer US. 2005. ISBN 978-0-387-26143-0. doi:10.1007/b137514.

[97] Mishra, P.; Koo, H.: Functional Test Generation Using Design and Property Decomposition Techniques. *ACM Transactions on Embedded Computing Systems*. vol. 8, no. 4. 2009.

[98] Mishra, P.; Tomiyama, H.; Dutt, N.; et al.: 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)*. IEEE. 2002. pp. 36–43. doi:10.1109/DATE.2002.998247.

[99] Moona, R.: Processor Models for Retargetable Tools. In *Proceedings of the 11th IEEE International Workshop on Rapid System Prototyping*. RSP'00. IEEE Computer Society. 2000. ISBN 0-7695-0668-2.

[100] Moura, L. D.; Bjorner, N.: Z3: An Efficient SMT Solver. In *Proc. of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, *LNCS*, vol. 4963. Springer. 2008. pp. 337–340.

[101] Nabeshima, H.; Iwanuma, K.; Inoue, K.: GlueMiniSat 2.2.5: A fast SAT solver with an aggressive acquiring strategy of glue clauses. *Computer Software*. vol. 29. 2012: pp. 146–160.

[102] Namjoshi, K. S.: Symmetry and completeness in the analysis of parameterized systems. In *Proc. of Verification, Model Checking, and Abstract Interpretation (VMCAI), LNCS*, vol. 4349. Springer. 2007. pp. 299–313.

[103] Nelson, C. G.; Oppen, D. C.: Simplification by Cooperating Decision Procedures. Technical report. Stanford, CA, USA. 1978.

[104] Nelson, G.; Oppen, D. C.: Simplification by Cooperating Decision Procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*. vol. 1, no. 2. 1979: pp. 245–257. ISSN 0164-0925. doi:10.1145/357073.357079.

[105] Ngyuen, M.; Thalmaier, M.; Wedler, M.; et al.: Unbounded Protocol Compliance Verification usign Interval Property Checking with Invariants. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*. vol. 27, no. 11. 2008.

[106] Paakki, J.: Attribute Grammar Paradigms — a High-level Methodology in Language Implementation. *ACM Computing Surveys*. vol. 27, no. 2. Jun 1995: pp. 196–255. ISSN 0360-0300. doi:10.1145/210376.197409.

[107] Patterson, D. A.; Hennessy, J. L.: *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann. 2011. ISBN 978-8178672663.

[108] Patterson, D. A.; Hennessy, J. L.: *Computer Organization and Design: The Hardware / Software Interface*. Boston: Morgan Kaufmann. fourth edition. 2012. ISBN 0123747503.

[109] Praet, J. V.; Lanneer, D.; Geurts, W.; et al.: *nML: A Structural Processor Modeling Language for Retargetable Compilation and ASIP Design. Systems on Silicon*, vol. 1. Burlington: Morgan Kaufmann. 2008. ISBN 9780123742872. pp. 65–93.

[110] Prikryl, Z.: Fast Simulation of Pipeline in ASIP Simulators. In *15th International Microprocessor Test and Verification Workshop, MTV 2014, Austin, TX, USA, December 15-16, 2014*. IEEE Computer Society. 2014. ISBN 978-1-4673-6858-2. pp. 10–15. doi:10.1109/MTV.2014.18.

[111] Prikryl, Z.; Masarík, K.; Hruska, T.; et al.: Fast Cycle-Accurate Interpreted Simulation. In *10th International Workshop on Microprocessor Test and Verification, MTV 2009*. IEEE Computer Society. 2009. ISBN 978-0-7695-4000-9. pp. 9–14. doi:10.1109/MTV.2009.11.

[112] Prikryl, Z.; Masarík, K.; Hruska, T.; et al.: Generated Cycle-Accurate Profiler for C Language. In *13th Euromicro Conference on Digital System Design, Architectures, Methods and Tools, DSD 2010, 1-3 September 2010, Lille, France*. IEEE Computer Society. 2010. ISBN 978-0-7695-4171-6. pp. 263–268. doi:10.1109/DSD.2010.39.

[113] Ramsay, F. R.: Automation of design for uncommitted logic array. In *Proc. of the 17th Design Automation Conference (DAC)*. New York, NY, USA: ACM. 1980. ISBN 0-89791-020-6. pp. 100—107.

[114] Rigo, S.; Araujo, G.; Bartholomeu, M.; et al.: ArchC: A SystemC-based Architecture Description Language. In *16th Symposium on Computer Architecture and High Performance Computing*. Oct 2004. ISSN 1550-6533. pp. 66–73. doi:10.1109/SBAC-PAD.2004.8.

[115] Risc-V Foundation: *Risc-V Instruction Set Architecture Specification*. 2017. Retrieved from: riscv.org/specifications/

[116] Rogin, F.; Klotz, T.; Fey, G.; et al.: Automatic Generation of Complex Properties for Hardware Designs. In *2008 Design, Automation and Test in Europe*. March 2008. ISSN 1530-1591. pp. 545–548. doi:10.1109/DATE.2008.4484908.

[117] Sanghavi, H. A.; Andrews, N. B.: Chapter 8 - TIE: An ADL for Designing Application-specific Instruction Set Extensions. In *Processor Description Languages*, edited by P. Mishra; N. Dutt. Morgan Kaufmann. 2008. pp. 183–216. doi:10.1016/B978-012374287-2.50011-2.

[118] Sawada, J.; Hunt, W. A.: Processor verification with precise exceptions and speculative execution. In *Computer Aided Verification*, edited by A. J. Hu; M. Y. Vardi. Springer Berlin Heidelberg. 1998. ISBN 978-3-540-69339-0. pp. 135–146.

[119] Schliebusch, O.; Meyr, H.; Leupers, R.: *Optimized ASIP Synthesis from Architecture Description Language Models*. Springer Netherlands. 2007. ISBN 978-1-4020-5685-7. doi:10.1007/978-1-4020-5686-4.

[120] Shahabuddin, S.; Janhunen, J.; Juntti, M.; et al.: Design of a transport triggered vector processor for turbo decoding. *Analog Integrated Circuits and Signal Processing*. vol. 78, no. 3. Mar 2014: pp. 611–622. ISSN 1573-1979. doi:10.1007/s10470-013-0183-y.

[121] Shen, J. P.; Lipasti, M. H.: *Modern Processor Design: Fundamentals of Superscalar Processors*. Waveland Press, Inc.. 2013. ISBN 978-1478607830.

[122] Sigasi: *Manual: Linting and Quick Fixes*. 2019. Retrieved from: insights.sigasi.com/manual/linting.html

[123] Šimková, M.; Lengál, O.; Kajan, M.: HAVEN: An Open Framework for FPGA-Accelerated Functional Verification of Hardware. In *Hardware and Software: Verification and Testing*. Berlin, Heidelberg: Springer Berlin Heidelberg. 2012. ISBN 978-3-642-34188-5. pp. 247–253.

[124] Stump, A.; Barrett, C. W.; Dill, D. L.; et al.: A decision procedure for an extensional theory of arrays. In *Proc. of 16th Annual IEEE Symposium on Logic in Computer Science*. IEEE. 2001. ISSN 1043-6871. pp. 29–37. doi:10.1109/LICS.2001.932480.

[125] Synopsys: *ASIP Designer: Design Tool for Application Specific Instruction-Set Processors, Designer Datasheet*. 2018. Retrieved from: synopsys.com/dw/doc.php/ds/cc/asip-designer-ds.pdf

[126] Synopsys: *ASIP Designer: Design Tool for Application Specific Instruction-Set Processors, Designer Datasheet*. 2018.

Retrieved from:
synopsys.com/cgi-bin/verification/dsdla/pdfr1.cgi?file=vcs-ds.pdf

[127] Synopsys: *SpyGlass Lint Datasheet*. 2018.
Retrieved from: synopsys.com/cgi-bin/verification/dsdla/docsdl/spyglass-lint-ds.pdf?file=spyglass-lint-ds.pdf

[128] Synopsys: *VC Formal Datasheet*. 2018.
Retrieved from:
synopsys.com/cgi-bin/verification/dsdla/pdfr1.cgi?file=vc_formal_ds.pdf

[129] Tepurov, A.; Tihhomirov, V.; Jenihhin, M.; et al.: Localization of Bugs in Processor Designs Using zamiaCAD Framework. In *2012 13th International Workshop on Microprocessor Test and Verification (MTV)*. Dec 2012. ISSN 1550-4093. pp. 41–47. doi:10.1109/MTV.2012.20.

[130] Trmač, M.; Husár, A.; Hranáč, J.; et al.: Instructor Selector Generation from Architecture Description. In *6th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*. Masaryk University. 2010. ISBN 978-80-87342-10-7. pp. 167–174.

[131] Velev, M.; Bryant, R. E.; Jain, A.: Efficient modeling of memory arrays in symbolic simulation. In *Proc. of Computer Aided Verification (CAV)*, *LNCS*, vol. 1254, edited by O. Grumberg. Springer Berlin Heidelberg. 1997. ISBN 978-3-540-69195-2. pp. 388–399. doi:10.1007/3-540-63166-6_38.

[132] Velev, M. N.: Efficient translation of Boolean formulas to CNF in formal verification of microprocessors. In *Asia and South Pacific Design Automation Conference (ASP-DAC'04)*. Jan 2004. ISBN 0-7803-8175-0. pp. 310–315. doi:10.1109/ASPDAC.2004.1337587.

[133] Velev, M. N.: Exploiting signal unobservability for efficient translation to CNF in formal verification of microprocessors. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, vol. 1. Feb 2004. ISSN 1530-1591. pp. 266–271. doi:10.1109/DATE.2004.1268859.

[134] Velev, M. N.: Using automatic case splits and efficient CNF translation to guide a SAT solver when formally verifying out-of-order processors. In *Artificial Intelligence and Mathematics (AI&MATH'04)*. 2004. pp. 242–254.

[135] Velev, M. N.; Gao, P.: Automatic Formal Verification of Multithreaded Pipelined Microprocessors. In *Proc. of International Conference on Computer Aided Design (ICCAD)*. IEEE. 2011. pp. 679–686.

[136] Velev, M. N.; Gao, P.: Automated debugging of counterexamples in formal verification of pipelined microprocessors. In *17th Asia and South Pacific Design Automation Conference (ASPDAC'12)*. Jan 2012. ISSN 2153-697X. pp. 689–694. doi:10.1109/ASPDAC.2012.6165044.

[137] Weiser, M.: Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering*. ICSE '81. Piscataway, NJ, USA: IEEE Press. 1981. ISBN 0-89791-146-6. pp. 439–449.

[138] Wilkes, M. V.: The best way to design an automatic calculating machine. In *Manchester University Computer Inaugural Conference*. Manchester, UK. 1951.

[139] Wilkes, M. V.; Stringer, J. B.: Micro-programming and the design of the control circuits in an electronic digital computer. *Mathematical Proceedings of the Cambridge Philosophical Society*. vol. 49, no. 2. 1953: pp. 230—238. doi:10.1017/S0305004100028322.

[140] Wolf, W.; Madsen, J.: Embedded systems education for the future. *Proceedings of the IEEE*. vol. 88, no. 1. 2000: pp. 23–30. ISSN 0018-9219. doi:10.1109/5.811598.

[141] Wolper, P.; Boigelot, B.: Verifying systems with infinite but regular state spaces. In *Computer Aided Verification*, edited by A. J. Hu; M. Y. Vardi. Berlin, Heidelberg: Springer Berlin Heidelberg. 1998. ISBN 978-3-540-69339-0. pp. 88–97. doi:10.1007/BFb0028736.

[142] Zachariášová, M.; Přikryl, Z.; Hruška, T.; et al.: Automated Functional Verification of Application Specific Instruction-set Processors. *IFIP Advances in Information and Communication Technology*. vol. 4, no. 403. 2013: pp. 128–138. ISSN 1868-4238. doi:10.1007/978-3-642-38853-8.