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

Automated Verification In Hardware / Software Co-design

Thesis Summary

Lukáš Charvát
icharvat@fit.vutbr.cz

Faculty of Information Technology
Brno University of Technology

Božetěchova 2
612 66 Brno
Czech Republic

August 9, 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.

Contents

1 Introduction

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 [34, 4, 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 sacri-

ficed 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 [5, 6], 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 [7, 8, 9, 10], 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 [10] and, in this thesis, we present promising experimental results applying the tool to multiple pipelined microprocessors.

Outline. The rest of this thesis summary is organized as follows. Section 2 gives an overview of microprocessor architectures together with an introduction to the former and contemporary techniques used during the design of embedded systems. Section 3 presents the main goals of the thesis. Section 4 describes a newly proposed technique for automatic generation of abstract models of memories that can be used for efficient formal verification of hardware designs. Next, Section 5 introduces our new automated approach built on a formal basis that we use for checking the correspondence between an RTL implementation of a microprocessor and its ISA description. Further, Section 6 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, Section 7 concludes the thesis summary.

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 [28, 26].

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 [26]. 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 subsections de-

scribe some of microprocessor and hardware architectures that are typically used in the embedded devices. The last subsection 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 [14]. 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 [25]. 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 [29]. 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 [28, 26].

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`, `amd64`, `armv7`, or `riscv`. 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 [33].

Taken all together, the typical organization of a simple microprocessor with a single pipeline is shown in Fig. 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

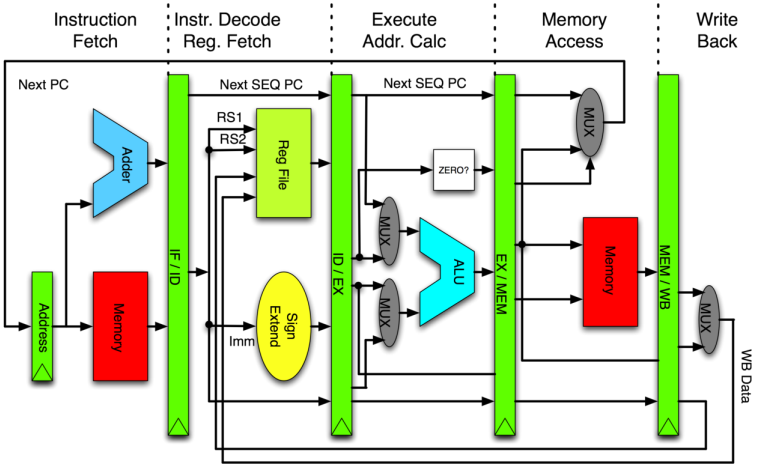

Figure 1: A typical organization of a simple microprocessor with a single pipeline.

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) [26]. 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 subsections).

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 [26].

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) [30] 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 [20] or Verilog [19]. 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 [26].

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.

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 [13]. 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, 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

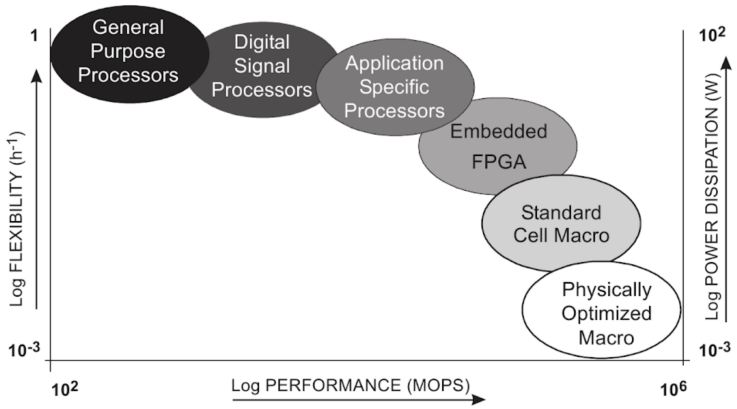

Figure 2: Trade-off between flexibility and performance among various components used in embedded systems. Source: [31].

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 [16] or signal processing [32].

2.4 Modern Hardware/Software Co-Design

As was discussed in the previous subsections, 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 in-

formation about assembler syntax, binary encoding of instructions, etc. This is the reason why specially crafted *architecture description languages* ADLs were introduced [27].

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.

A common exploration co-design flow consists of the following steps [37]. 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).

ADLs play key role in the modern hardware/software co-design. Therefore, Chapter 3 of the thesis describes and classifies them in a more detail together with their accompanying tools. An example of a microprocessor component description using the ADL can be seen in Section 5.1 of this summary.

3 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 Section 5 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 Section 4.

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 Section 6, we propose an approach for detection of problems caused by data and control hazards in pipelined microprocessor designs.

4 Large Memory Abstraction

This section introduces a novel technique for automatic generation of abstract models of memories that can be used for efficient formal verification of hardware designs. The approach can 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 for 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. Finally, the 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 we witnessed in experiments with the approach described in Chapter 8 of the thesis.

4.1 Related Work

Numerous works have focused on memory abstraction, notably within the area of formal verification. Some of the proposed abstractions are tightly coupled with the verification procedure used: for instance, many of them rely on that SAT-based bounded model checking is used [24, 11]. An approach not tailored for a specific verification approach has been presented in [12] which introduces a theory for 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. This approach does, however, not support addressing of different sizes of data. A recently published work [17] formally specifies and verifies a model of a large memory that supports efficient simulation. The model is tailored for `amd64` implementations only in order to offer a good trade-off between the speed of simulation and the needed computational resources. The approach assumes starting from the nullified state of the memory, not from a random state.

Unlike the above approaches, an algorithm proposed in the thesis can generate abstractions of memories that support addressing of arbitrary addressable units with multiple read and write ports, and it allows the memory to start from a random initial state. Moreover, the algorithm is not bound to any specific verification technique. The generated abstraction can be described in any language for which the user can provide templates specifying (i) how to express declarations of state and signal variables, (ii) how to encode

propositional logic expressions over state and signal variables, (iii) and how to define initial and next states of state variables.

4.2 Summary of the Approach

When formally verifying a system which uses memories, a need to deal with large capacity memories causes a state explosion. The technique of memory abstraction proposed in the paper builds on 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 as actually stored in the memory (abstracting away the random contents stored at unused memory locations). Our abstraction preserves the interface of the abstracted memory, but 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 of couples of variables storing corresponding pairs of addresses and values. When using bounded model checking (BMC) as the verification technique, the needed number of address-value pairs can be easily determined from the depth of BMC. For unbounded verification, this number 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. If the memory is accessed for reading, the address-value pairs that are in use are searched. If a location is read that has already been accessed in the past, then the value associated with the appropriate address is simply returned. If a location that has never been accessed is read, a corresponding pair is not found in the table, and a new couple is allocated. Its address part will store the particular address that is accessed while the value stays unconstrained. However, the variable representing the value associated with the accessed location stays constant in the future (unless there occurs a write operation to the concerned address). This ensures that subsequent reads from the location return the same value. In the case of writing, the address and value are both known. In the

case of writing to a location that has not been accessed yet, a new address-value pair is allocated to store the given address-value pair. Otherwise, a value associated with the given address is updated.

In order to support dealing with 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 and a set of functions mapping accesses to ports of the modeled memory to ports of the low-level memory. The low-level memory consists of cells whose size equals the size of the least addressable unit of the modeled memory. In order to allow for reading/writing the allowed sizes of data in one step, the number of read and write ports of the low-level memory is appropriately increased.

In order to prove the usefulness of the proposed model, we used it in a manual, but strictly algorithmic way to derive memory models for the approach of checking correspondence between the ISA and RTL level descriptions of microprocessors described in Chapter 8 of the thesis. Our experiments (discussed in Chapter 7 of the thesis) showed significant improvements in the verification time.

5 RTL-ISA Correspondence Checking

In Chapter 8 of the thesis, we propose 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). The approach is original in its very high level of automation: the only user inputs are an RTL implementation, an ISA description (possibly complemented by a specification of assumed restrictions on the possible values of instruction operands), and a time limit for the verification.

The main idea behind our approach is to use bounded model checking (BMC) to compare the outputs produced by automatically derived RTL and ISA models of a given processor for all possible

instructions and their inputs. In order to guarantee that some useful result is obtained in a given time limit, each instruction is checked in parallel for several bit-widths of its input, and the maximum bit-width for which a result is obtained in the given time limit is used.

Compared to the techniques proposed, e.g., in [3, 21], the approach presented in this section 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 testing, and due to a different way of exploring the state space of the verified design, it can find bugs not found by, for instance, 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 also shown that almost every instruction of a simple pipeline processor (of a form commonly used in light-weight embedded devices) is verified within seconds. Shortened input data were used only in a few cases, typically for instructions such as multiplication (and even in such cases, one can argue that most typical bugs would anyway manifest even for shortened input).

5.1 Background: Design and Verification 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 Chapter 3 of the thesis), but the proposed method can be used within other microprocessor development toolchains too if they are able to provide all needed information about the processor (as discussed below).

Our method uses both the ISA and RTL models 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 [18, 35]. From the RTL-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. 3. 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 format with a few simple optimizations.

5.2 Summary of the Approach

We concentrate on checking the 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

```
 1 element reg represents regs {
 2     use imm4 as num;
 3     assembler { "r" ~ num };
 4     binary { num };
 5     return { num; };
 6 }
 7 element add {
 8     assembler { "ADD" };
 9     binary { OP_ADD:4 };
10     return { OP_ADD; };
11 }
12 set opc = add, /* ... */;
13 element instr_alu {
14     use reg as { dst, sA, sB };
15     use opc;
16     assembler { opc dst "," sA "," sB };
17     binary    { opc dst     sA     sB };
18     semantics {
19         switch (opc) {
20             case OP_ADD:
21                 regs[dst] = regs[sA] + regs[sB];
22                 cf = add_carry(regs[sA], regs[sB]);
23                 break;
24             /* ... */
25         }
26     };
27 }
```

Figure 3: A description of the `add` instruction in CodAL.

is still useful, especially when combined with other techniques (such as the one discussed in Section 6).

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 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 [3], 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.

A more detailed description, as well as experimental evaluation of the above-sketched approach, is given in Chapter 8 of the thesis.

6 Analysis of Pipeline Hazards

Implementation of pipeline-based execution of instructions in ASIPs is an error-prone task, which implies a need for proper verification of the resulting designs. Various techniques were proposed for this purpose, but they usually require a significant manual intervention of the developers. Thus, in Chapter 9 of the thesis, we propose a novel, highly automated approach for discovering *read-after-write (RAW)*, *write-after-write (WAW)*, *write-after-read (WAR)*, and *control hazards*.

6.1 Summary of the Approach

We expect the verified processor to be expressed by a so-called *processor structure graph* (PSG) that describes interconnections between *storages* (i.e., registers and memories) and *combination logic* of the design.[1] The graph can be easily obtained from a description of the processor on the register transfer level (RTL) written either in

[1] A rigorous description of the PSG is given in Chapter 9 of the thesis.

some ADL (such as CodAL) or in a hardware description language such as VHDL or Verilog.

The first step of our approach is a simple *data flow analysis* performed upon the PSG in order to get, for each vertex v of the PSG representing a storage or combination logic, (i) its pipeline stage $\varphi(v) \in \mathbb{S}$ where $\mathbb{S}$ is the set of all pipeline stages, (ii) a set of writing stages $\varphi^{\mathrm{wr}}(v) \subseteq \mathbb{S}$ that directly influence the value of v, and (iii) a set of reading stages $\varphi^{\mathrm{rd}}(v) \subseteq \mathbb{S}$ that are influenced by v. Next, we examine the PSG, and using results of the previous step, we identify a finite set of so-called *hazard cases*, each of them describing one possible source of a RAW, WAW, WAR, or control hazard. For instance, in the case of a WAR hazard which arises when an instruction writes to a storage that some earlier instruction reads while there is a possibility for the earlier instruction to read a new value already written by the later instruction, the hazard case is given by a 7-tuple $(v_w, s_w, v_r, s_r, v_t, s_t, \pi)$. The components of the hazard case represent (i) a storage v_w, (ii) its write stage $s_w \in \varphi^{\mathrm{wr}}(v_w)$, (iii) a reading storage v_r such that $v_r = v_w$ if v_r is a register or such that v_r and v_w are ports of the same memory, (iv) the read stage $s_r \in \varphi^{\mathrm{rd}}(v_r)$ such that $s_w < s_r$ in order that the storage is written before it is read to evoke a WAR hazard, (v) a target storage v_t where the potentially incorrect value read from v_r is stored, (vi) a stage $s_t \in \varphi^{\mathrm{wr}}(v_t)$, $s_r \leq s_t$, in which the incorrect value is stored, and (vii) a data path π describing how data are propagated from v_r to v_t between the stages s_r and s_t.

As it is shown in Chapter 9 of the thesis, the behavior of the instructions given by constraints of a hazard case can be modelled using a parameterized system which maps n instructions in the pipeline to n processes in a linear array. The system is then checked whether there exists some sequence of instructions that could generate hazard conditions. If there is no such case, hazards are handled properly by the processor. To show the usefulness of the proposed approach, we performed a series of experiments (also discussed in Chapter 9 of thesis) on non-trivial microprocessors where the approach was able to correctly locate errors caused by incorrectly handled hazards.

6.2 Related Work

Proving the absence of pipeline hazards is a native part of checking equivalence between an RTL design and its ISA description. One of the most cited approaches to such checking is the so-called flushing technique [3], which has been extended, e.g., in [21, 22, 36, 15], to handle rather complicated designs including multi-cycle execution units, exceptions, and branch prediction. The main challenge of these works is to overcome the semantic gap between the different levels of a processor description which often requires a significant user intervention in the means of, e.g., providing various additional assertions. Approaches described in [2, 23] introduce an abstract formal model whose components are to be linked by the user with the concrete cycle-accurate implementation through a number of mappings. Afterward, the model is checked whether it satisfies several properties together implying correctness of the pipeline behavior.

Compared with the above approaches, we do not aim at full conformance checking between RTL and ISA implementations. Instead, we address several specific properties, in particular, the absence of problems caused by data and control hazards. On the other hand, our approach is almost fully automated—the only step required from the user is to identify programmer visible registers and memories of the verified design.

7 Conclusion

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 of the thesis.

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

[3] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Proc. of Computer Aided Verification (CAV)*, volume 818 of *LNCS*, pages 68–80. Springer, 1994.

[4] Cadence. *Tensilica Software Development Toolkit (SDK)*, 2014.

[5] L. Charvát, A. Smrčka, and T. Vojnar. Automatic formal correspondence checking of ISA and RTL microprocessor description. In *Proc. of Microprocessor Test and Verification (MTV'12)*, pages 6–12. IEEE, 2012.

[6] L. Charvát, A. Smrčka, and T. Vojnar. An abstraction of multi-port memories with arbitrary addressable units. In *Proc. of Computer Aided Systems Theory (EUROCAST'13)*, volume 8111 of *LNCS*, pages 460–468. Springer, 2013.

[7] L. Charvát, A. Smrčka, and T. Vojnar. Using formal verification of parameterized systems in RAW hazard analysis in microprocessors. In *Proc. of Microprocessor Test and Verification (MTV'14)*, pages 83–89. IEEE, 2014.

[8] L. Charvát, A. Smrčka, and T. Vojnar. Using formal verification of parameterized systems in RAW hazard analysis in microprocessors. Technical Report FIT-TR-2014-04, Brno University of Technology, 2014.

[9] L. Charvát, A. Smrčka, and T. Vojnar. Microprocessor hazard analysis via formal verification of parameterized systems. In *Proc. of Computer Aided Systems Theory (EUROCAST'15)*, volume 9520 of *LNCS*, pages 605–614. Springer, 2015.

[10] L. Charvát, A. Smrčka, and T. Vojnar. 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, pages 87–93. EPTCS, 2016.

[11] M. K. Ganai, A. Gupta, and P. Ashar. Verification of embedded memory systems using efficient memory modeling. In *Proc. of Design, Automation and Test in Europe (DATE)*, volume 2, pages 1096–1101. IEEE, 2005.

[12] Steven M. German. A theory of abstraction for arrays. In *Proc. of the International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 176–185. FMCAD, 2011.

[13] Matthias Gries and Kurt Keutzer. *Building ASIPs: The Mescal Methodology*. Springer US, 2005.

[14] Eldon C. Hall. *Journey to the Moon: The History of the Apollo Guidance Computer*. American Institute of Aeronautics, 1996.

[15] K. Hao, S. Ray, and F. Xie. Equivalence checking for function pipelining in behavioral synthesis. In *Proc. of Design, Automation and Test in Europe (DATE)*, pages 1–6. IEEE, 2014.

[16] I. Hautala, J. Boutellier, J. Hannuksela, and O. Silvén. Programmable low-power multicore coprocessor architecture for

hevc/h.265 in-loop filtering. *IEEE Transactions on Circuits and Systems for Video Technology*, 25(7):1217–1230, July 2015.

[17] W. A. Hunt and M. Kaufmann. A formal model of a large memory that supports efficient execution. In *Proc. of Formal Methods in Computer-Aided Design (FMCAD)*, pages 60–67. IEEE, 2012.

[18] Adam Husár, Miloslav Trmač, Jan Hranáč, Tomáš Hruška, Karel Masařík, Dušan Kolář, and Zdeněk Přikryl. Automatic c compiler generation from architecture description language isac. In *6th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*, pages 84–91. Masaryk University, 2010.

[19] IEEE. *IEC/IEEE Behavioural Languages – Part 4: Verilog Hardware Description Language*, 2004.

[20] IEEE. *IEEE Standard VHDL Language Reference Manual*, 2009.

[21] R. B. Jones, C. H. Seger, and D. L. Dill. Self-consistency checking. In *Proc. of Formal Methods in Computer-Aided Design (FMCAD)*, volume 1166 of *LNCS*, pages 159–171. Springer, 1996.

[22] A. Koelbl, R. Jacoby, H. Jain, and C. Pixley. Solver technology for system-level to rtl equivalence checking. In *Proc. of Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 196–201. IEEE, 2009.

[23] U. Kuhne, S. Beyer, J. Bormann, and J. Barstow. Automated formal verification of processors based on architectural models. In *Proc. of Formal Methods in Computer-Aided Design (FMCAD)*, pages 129–136. IEEE, 2010.

[24] P. Manolios, S. K. Srinivasan, and D. Vroon. Automatic memory reductions for rtl model verification. In *Proc. of IEEE/ACM*

International Conference on Computer Aided Design (ICCAD), pages 786–793. IEEE, 2006.

[25] Marshall William McMurran. *ACHIEVING ACCURACY: A Legacy of Computers and Missiles*. Xlibris Corp., 2008.

[26] Miloš Minařík. *Concurrent Evolutionary Design of Hardware and Software*. PhD thesis, Brno University of Technology, Faculty of Information Technology, 2017.

[27] P. Mishra and N. Dutt. Architecture description languages for programmable embedded systems. *IEE Proceedings – Computers and Digital Techniques*, 152(3):285–297, May 2005.

[28] D. A. Patterson and J. L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2011.

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

[30] F. R. Ramsay. Automation of design for uncommitted logic array. In *Proc. of the 17th Design Automation Conference (DAC)*, pages 100—107, New York, NY, USA, 1980. ACM.

[31] Oliver Schliebusch, Heinrich Meyr, and Rainer Leupers. *Optimized ASIP Synthesis from Architecture Description Language Models*. Springer Netherlands, 2007.

[32] Shahriar Shahabuddin, Janne Janhunen, Markku Juntti, Amanullah Ghazi, and Olli Silvén. Design of a transport triggered vector processor for turbo decoding. *Analog Integrated Circuits and Signal Processing*, 78(3):611–622, Mar 2014.

[33] John Paul Shen and Mikko H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. Waveland Press, Inc., 2013.

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

[35] Miloslav Trmač, Adam Husár, Jan Hranáč, Tomáš Hruška, and Karel Masařík. Instructor selector generation from architecture description. In *6th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*, pages 167–174. Masaryk University, 2010.

[36] M. N. Velev and P. Gao. Automatic formal verification of multithreaded pipelined microprocessors. In *Proc. of International Conference on Computer Aided Design (ICCAD)*, pages 679–686. IEEE, 2011.

[37] W. Wolf and J. Madsen. Embedded systems education for the future. *Proceedings of the IEEE*, 88(1):23–30, 2000.

A Curriculum Vitae

Education

Bachelor's degree, Information Technology, 2006 – 2009
Brno University of Technology

Masters' degree, Information Technology, 2009 – 2011
Brno University of Technology

Experience

Senior Software Design Engineer Oct 2017 – now
Oracle – NetSuite, Platform Integration

- Software development in finance, ERP, and CRM cloud-based system.
- SW Developer / Team Lead / Scrum Master in a small team (of 6) responsible for the design and implementation of a new framework allowing exposure of NetSuite customizable records (250+ in total) via the REST API endpoint.

Software Design and Verification Engineer Jul 2015 – Sep 2017
Honeywell Inc., Homes and Building Technologies

- SW Developer / Technical Product Owner / Architect for radio firmware of connected devices specializing in dynamic and formal analysis of multi-threaded embedded systems (memory safety, data-race detection) and automation of the analysis.

Notable Products and Projects

Smart Home Security (Honeywell) Feb 2017 – Sep 2017

- Participation in design and implementation of a Linux daemon responsible for maintenance of secure telemetry connection in a smart home security device.
- Technologies: C, AMQP, Linux, OpenSSL, Atmel ATECC508.

Lyric T5/T6 Wi-Fi Thermostat (Honeywell) Jul 2015 – Feb 2017

- Software design and dynamic verification of radio controller firmware in a project aimed at the development of a new platform for cloud-connected thermostats (consisting of 450+ kloc in total).
- Introduced a hardware abstraction layer and SW simulator for dynamic analysis. Detection of new issues (40+), saving (est.) 2.5 man-months compared to traditional debugging.
- Implementation of Klocwork static code analyzer plugins that revealed 120+ new critical issues saving (est.) 2 man-months of development.
- Optimized a process of ECDCA signature and verification of telemetry messages using ARMv7 assembler resulting in speed-up of 1.8x (500 to 280ms).
- Technologies: C/C++, AMQP, Mocana, HomeKit, ThreadX, NetX, ARM Cortex M7, Broadcom BCM43012.

Hazard Detection System (BUT) Mar 2013 – now

- An automated formal verification tool designed for detection of data hazards in single pipelined microprocessors. Available at `http://www.fit.vutbr.cz/research/groups/verifit/tools/hades/`
- Technologies: C++, Python, SMTLib, Bash.

Skills and Expertise
Software

- Advanced knowledge of C, C++, C#, Python, Java, Javascript.
- Advanced experience with software verification tools (e.g., Frama-C, Valgrind plugins).
- Intermediate knowledge of PHP, Visual Basic, Haskell, SQL.

- Knowledge of UML and software design tools (Enterprise Architect, Visual Paradigm).
- Databases: OracleDB, ElasticSearch.
- Certificates: Cisco CCNA Routing and Switching Certificate, Design For Six Sigma Core (Green Belt), Scaled Agile Framework (SAFe).

Hardware

- Advanced experience with formal HW verification tools, NuSMV, ABC, Z3.
- Good knowledge of VHDL, Verilog / SystemVerilog.
- Basic knowledge of methodologies used to verify integrated circuits, OVM, UVM.