Towards Formal co-Validation of Hardware and Software Timing Models of CPS

Mihail Asavoae[1], Imane Haur[1], Mathieu Jan[1],
Belgacem Ben Hedia[1], Martin Schoeberl[2]

[1] CEA, List, France, firstname.lastname@cea.fr
[2] Technical University of Denmark, masca@dtu.dk

Abstract. Timing analysis of safety-critical systems derives timing bounds of applications (SW) executed on dedicated platforms (HW). The ensemble HW–SW features, from a timing perspective, two different types of computation – a SW-specific, instruction-driven timing progression and a HW-specific, cycle-driven one. The two timings are unified under a concept of *timing model*, which is crucial to establish sound and precise timing reasoning. In this paper we propose an investigation on how to systematically derive and formally prove such timing models. Our approach is exemplified on a simple, accumulator-based processor called Lipsi.

Keywords: timing analysis, timing model, formal semantics, Chisel, HW/SW co-validation

1 Introduction

Cyber-physical systems (CPSs) integrate computations running on embedded platforms into physical systems that they interact with. This integration, expressed through feedback loops between software and the physical environment, may need to satisfy strong timing guarantees. To verify them, worst-case timing analyses of safety-critical systems are combined. Such analysis cover timing behaviors of computations (e.g., the worst-case execution time analysis, WCET), communications (e.g., the worst-case traversal time analysis) or both (e.g., the worst-case response time analysis). The commonality of all these analyses is that the application semantics are projected on the timing behavior of the underlying platforms. In this work, we are interested in manipulating the computations, viewed as a set of binaries of software (SW), i.e. sequences of instructions described at the instruction set architecture (ISA) binary level. On the embedded platforms side, i.e. hardware level (HW), our inputs are register transfer-level (RTL) descriptions of processors in a given hardware description language (HDL). We thus omit any networking components and focus for now on single-core or multi-core architectures only.

At these levels of design, a timing verification relies on a WCET analysis [41]. A WCET analysis computes safe and accurate timing bounds of a SW executed on a HW, where the HW is abstracted to micro-architecture elements (e.g., instruction and data caches, pipeline, speculation mechanisms, etc.). The workflow of the WCET analysis consists of a series of static SW- and HW-level analyses. A first analysis extracts the control flow graph (CFG) from the binary, with nodes being basic (single-entry, single-exit) blocks and edges safely representing the program flow. Then, the CFG is augmented with flow facts (e.g., loop bounds) and HW-related information (e.g., cache behavior). Finally, a path analysis, performed on the augmented CFG, determines the longest execution path which represents the WCET bound. In this workflow, a cache analysis reports that, for example, a basic block is classified as "always miss" in the cache and the timing behavior associated is a cache miss penalty, for example 10 cycles. However, it is not always simple to establish such architecture-dependent bounds due to complexity and unwanted timing phenomena [40]. WCET analyses require a proper timing model, whose correctness and accuracy ensure safe but also tight WCET estimations.

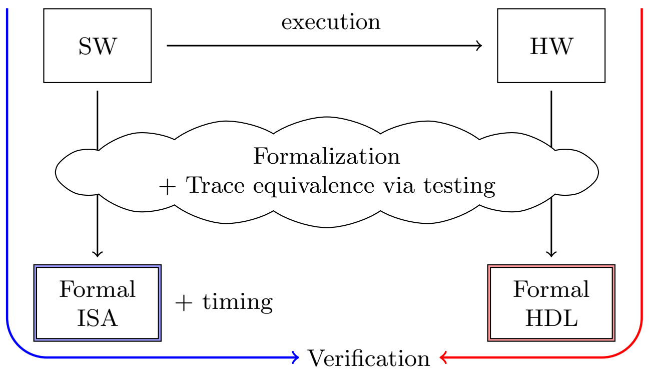

Fig. 1: General workflow for the HW/SW co-validation of timing models for CPS.

In this work, we focus on building and validating such timing models for CPS. Figure 1 shows our proposed general workflow that combines the formalization of the timing behaviors of both SW and HW sides. For the SW side, a timing model is added to a formal description of the ISA by relying on the input textual specification of the architecture. Both the functional and timing behaviors of this timing augmented formal SW model are tested, for confidence, against traces of actual code first over an Instruction Set Simulator (ISS) and then over a real execution. Note that by real execution, we mean traces generated by executing HDL code, i.e. the circuit, should be on a FPGA or by a cycle accurate simulator. Similarly, on the HW side, a formal model is built which thus includes by construction a timing behavior. The confidence in this formal HW model is also verified by applying trace equivalence. Finally, a last step consists in the co-validation of these hardware and software formal timing models, i.e. verifying the consistency of their two timing models. More specifically, the instruction-level timing, from the SW model is encoded as a set of assertions, called *timing invariants*, in the formal HW model. A timing invariant identifies the hardware pattern, i.e. the sequence of updates on registers and wires that must be executed in x cycles, corresponding to the execution of an instruction in the timing augmented SW model in x cycles. We then prove the obtained timing model of a particular pairing of programs (SW) over platform (HW) by model checking the formal HW model with these timing invariants.

To illustrate how this general approach can be used when designing CPS, we apply it over Lipsi processors [37]. Lipsi is a sequential tiny processor whose simplicity enables us to focus on building the complete approach for this HW platform. Lipsi comes with two components: an instruction-set simulator and the circuit, specified in Chisel [4]. We then build both HW and SW models using respectively the Temporal Logic of Actions (TLA$^+$) formal specification language [20] and the Sail formal language dedicated to express ISA [1]. We detail these models, the simple timing model of Lipsi that we add in the SW model and how we add these timing invariants in the HW model of Lipsi. These timing invariants are then checked using the model-checker of TLA$^+$, called TLC. In the formalization phase we discovered semantic issues between the intended behavior of Lipsi, from [37] and the actual Lipsi code. Finally, we report a possible use case for such verified timing models of CPS: the detection and minimization of memory interferences in a multi-core setting of Lipsi.

The remainder of this paper is organized as follows. Section 2 introduces the formal languages we rely on to build the HW and SW models of our general approach. It also provides an overview of the timing behavior of the Chisel language, from which the timing behavior of our HW model is constructed. In Section 3, we present the workflow we develop to achieve the HW/SW co-verification of timing models of CPS. Section 4 applies this approach to a case study: the Lipsi processor. We provide concrete examples of the HW and SW formal models of Lipsi using TLA$^+$ and Sail languages. Section 5 reports and discuss the obtained results when considering this Lipsi case study. Section 6 presents related work, before concluding in Section 7.

2 Preliminaries

In this section, we briefly introduce two specialized specification languages we rely on to build the formal HW and SW models. To formally specify hardware behavior, we use the TLA$^+$ language [20, 43], as it is a high-level specification language for modeling concurrent systems and comes with a model-checker. On the SW side, we use Sail [1] to formally model programs as it is tailored for expressing Instruction-Set Architecture (ISA) semantics of processors. It has been successfully applied to formalize various ISAs, such as ARM, RISC-V and MIPS [1]. The K framework, which was recently used for a formal executable semantics of x86-64 ISA [10], is another option we have not yet considered so far [2].

Chisel. We first introduce the programming language we assume as input to perform hardware designs. We select the Chisel programming language [4] due to its current rising popularity in the hardware design community and its ability to reduce hardware design times. Chisel generates Verilog HDL, whose formal semantic has been studied in [12, 27]. Compared to Verilog, Chisel provides some higher-level constructs to raise the level of hardware design abstraction, as it is embedded in the Scala programming language. Scala promotes a functional-style programming and uses a strong static type system to facilitate concise and re-usable code – some necessary attributes to address complex hardware designs. Chisel also supports testing through an internal cycle-accurate hardware simulator. We now briefly describe several language elements of Chisel, while more elaborate examples are presented in Section 4.3 when we formally specify the Lipsi processor from Chisel code.

Hardware designs are constructed on Chisel typed values which flow through wires, i.e. the combinational part of circuits, or held in state elements, i.e. the sequential part of circuits. The keyword `val` is used to declare variables whose values do not change. As basic datatypes, `Bool` represents the boolean values, `SInt` and `UInt` represent signed and respectively unsigned integers. For example, `val x = UInt(2)` declares `x` to be the unsigned int `2`. The bit size of `x` is unspecified here, but could be inferred from the usage of `x` or specified to be 32-bit with `UInt(2, width=32)`. Simple combinational circuits are described using the `val` keyword. For example, `val land = a & b` has in `land` the bitwise and of `a` and `b`. Chisel uses registers for state elements, for instance, `val r = RegInit(0.U)` initializes `r` to 0. There are also conditional combinational circuits built with the class `Wire` and the conditional construct `when`. For example, `val x = Wire(UInt(0))` declares `x` to be an wire initialized to unsigned int `0` and modified to `1` as follows: `when (cond) { x := 1.U }`. The assignment operator `:=` connects the input of the left-hand side to the output of the right-hand side.

TLA$^+$ is a modeling language proposing an advanced module system, untyped set theory and predicate logic, making it suitable to specification of complex computational systems such as computer architectures. TLA$^+$ language is based on the notion of action (i.e., a transition predicate), which captures a state change as follows: an action `x' = x + 1` updates the next value, primed of x based on the current, unprimed value of `x`. If `x` is a record, its field `f` is accessed as `x.f` and a partial record update is expressed with `[x EXCEPT!.f = v]`, changing `f` to `v` and leaving the other fields of `x` unchanged. A module `M` having an action `A` could be referred in another module via the operator "`!`", as `M!A`. An instance of `M` could be created inside another module with the construct `INSTANCE M WITH`, with the state variables of `M` being initialized after the keyword `WITH`.

Abstraction and refinement are natural with TLA$^+$ and, supported by an explicit model checker called TLC, form a powerful formal specification and verification framework. It features stuttering invariance to reason about the specification paths at different levels of granularity and temporal existential quantification to slice away the unnecessary state elements. We elaborate more on the language and model checking of TLA$^+$ specifications in Section 4.3 where we present the case study of Lipsi processor.

Sail is a first order imperative language that comes with a type system for bitvector lengths and indexing to enables static checking. A Sail specification relies on the definition of an Abstract Syntax Type (AST) of the ISA of an architecture, i.e. an union of types with parameters. To each AST value, specific `execute` and `decode` functions are associated to respectively specify the sequential semantic of the instruction and the matching of its binary representation to its AST value. For readability and modularity, Sail supports the definition of scattered functions and unions allowing to group `decode`, `execute` and the AST value of an instruction in one place. The memory space and the registers of the architectural state of a processor manipulated by each instruction can also be specified to model data transfer paths. To this end, a specific `register` type is supported and it is possible to annotate functions with effects to describe their impact on either the declared memories or registers. From a Sail specification, both emulators and theorem-prover definitions can be generated to support fast execution of programs or deductive reasoning. In section 4.2, we provide examples on how we use Sail to formally specify the ISA of the Lipsi processor.

3 Co-validation of timing models: general approach

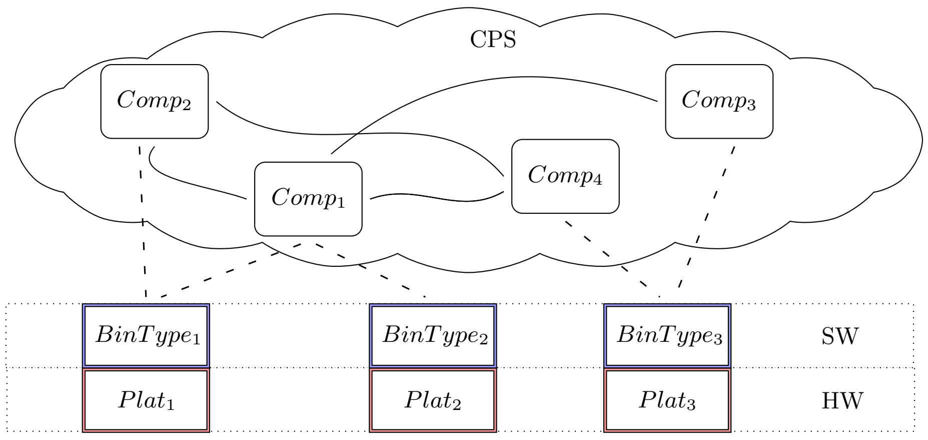

Fig. 2: Components $Comp_i$ of a CPS are classified based on their execution environments.

Typical CPSs are organized as networks of computational and communication elements which interact with the physical environment. CPSs are subject to various properties such as adaptability, autonomy, reliability, security or safety. From a structural point of view, a CPS features multiple components specified and implemented at different levels of details and using different modeling or specification languages. From a functional point of view, each such language comes with its own semantics to address specific points. As such, the CPS semantics landscape includes hybrid approaches, synchronous data flow approaches, simulation or verification languages, general-purpose programming languages, such as C, etc.. We thus abstract a CPS to ensembles of communicating binaries running of various platforms, as shown on Fig. 2. Heterogeneous computational and communicational components, $Comp_i$ are to be deployed for execution on various platforms, which is, in the general case, a many-to-many relation. For example, $Plat_3$ could be an multi/many-core system with applications from $Comp_{3-4}$ being executed in parallel. Also, different applications from $Comp_1$ could be executed on $Plat_1$ and $Plat_2$, whereas the applications of $Comp_2$ are executed only on $Plat_1$.

3.1 Motivations: consistency of timings

Let us elaborate on the semantics landscape of CPSs with respect to timing properties, because well-defined timing is crucially important to ensure several of the aforementioned properties of CPSs. Timing models capture how software applications, or programs, are mapped on hardware resources, when it comes to timing behavior. Timing models are necessary to perform timing analyses. A timing analysis needs to be safe and accurate and the most precise timing is to be found at the execution platform-level, i.e. the HW-level. To bridge the gap between HW-level timing and various ways of expressing timing at the application-level, i.e. the SW level, a timing analysis is usually performed on binaries. From a semantic perspective, the working language is the assembly languages specific to the execution platform under consideration, i.e. the ISA.

The high-level functional and temporal properties are however obfuscated or even lost when translated to this binary low-level, i.e. $BinType_{1-3}$ in Fig. 2. Let us consider a Model-Based Design (MBD) workflow, based on a synchronous language [7] as high-level programming language. The high-level specification is transformed into an intermediate language code, usually the C language, in a correct by construction way, ensured by the well-synchronized property of the high-level code. Then, a general purpose compilation chain, in the absence of compiler optimization, could produce binaries which are traceable to the source code. However, the initial high-level timing properties can no longer be directly expressed at the binary level, as most ISAs simply do not include timing. A few exceptions exist, such as PRET [26] and the Patmos [38] architectures that provide instructions that explicitly manipulate timings. Even when it is up to the compiler to schedule instructions over a given hardware architecture, the defined timing behavior is limited to a single instruction in isolation. There is no way to impact the timing behavior of the whole architecture, as in PRET or Patmos. This prevents to implement in a proper way the initial high-level temporal properties, which thus simply disappear at the binary level. It is the goal of WCET analyses to recover them at this level.

When designing an hardware architecture, its timing behavior is in general not formalized. Besides, it remains to demonstrate, or at least show enough confidence, that the timings are correctly implemented by the circuit. Most formal verification of hardware designs focus on the functional side of an implementation, as for instance in [42], to cite an open-source tool/project. While the timing semantic of HDL languages has been the subject of various work (see section 2), the timing behavior of a given micro-architecture is much more complex to identify and verify due to pipeline stalling, forwarding, interlocking, etc. The associated logic can be dispatched in different places of the design and mixed with rich low-level functionalities, such as the logic in charge of the functional part of the architecture. It is in general unclear when an instruction terminates its execution. Appropriate abstraction and slicing are necessary to extract the timing model of a hardware design so that it can be used to perform WCET analyses, i.e. if the high-level temporal properties can be fulfilled over the given architecture.

3.2 Building timing models

In this work, we consider the pair SW and HW represented by $BinType_i$ and respectively $Plat_i$ in Fig. 2 and how models can be established between them. The SW model is supported by instruction-level simulation, sometimes with an ad-hoc cycle-level timing, whereas the HW model is a cycle-accurate execution. Now, a *timing model* in this context is a function between the time progression of SW, measured in executed instructions and the cycle-accurate timing, corresponding to HW. Since we use such timing models in worst-case reasoning (to safely guarantee timing behaviors), we accompany this function with information on timing predictability [39] and timing compositionality [14]. In other words, our definition of timing model also includes knowledge on timing anomalies [35].

We propose a formal framework to construct and validate, from these HW and SW models, such *timing models* of CPS, abstracted as in Fig. 2. This framework relies on a combination of trace equivalences and model checking, as shown in Fig. 1. On the SW side, the steps of our workflow are thus the following:

1. We formalize the SW component by defining a formal executable semantics of $BinType_i$ ISA language.
2. This formal SW model is first tested, by comparing its output against traces generated from an Instruction Set Simulator (ISS), to gain confidence in the correctness of the functional part of the formal ISA model.
3. The formal SW model is then extended with timing behavior for each instruction. The augmented model is tested by comparing its output against traces from ISS, augmented with cycle-level timing behaviors, to gain confidence in the added timing model.

The result is a thoroughly tested formal SW model, though manually asserted, with a cycle-level timing behavior, i.e. formal ISA + timing in Fig. 1. Note that if a cycle-level simulator is not available, the timing behavior added in the above step (3) must be verified against traces from executions of the considered circuit.

Note that we can leverage existing formal SW models, such as [1] or [10] to avoid steps (1) and (2).

We propose similar steps to construct a formal HW model for $Plat_i$. Compared to the SW side of our workflow, a timing behavior already exists at the HDL level: we have to extract it. The steps on the HW side are thus the following:

1. We construct a formal executable HW model of the considered circuit. This is currently being done manually, automatically generating the HW model being a work in progress.
2. The built formal HW model is then tested, by comparing its outputs against traces from executions of the circuit, to gain confidence that it behaves as precisely as possible to the timing behavior of the HDL.

The result of these steps is a formal HW model with a cycle-accurate timing, denoted by formal HDL in Fig. 1. Obviously, we assume that a description of the circuit in HDL or using a higher-level language, such as Chisel, is available. If it is not the case, a more abstract HW model can be built from the specifications of the data paths. The timing accuracy of such models is of course more limited. Finally, note by executions of the circuit for step (2) on the HW side and step (3) on the SW side, we mean either the use of a cycle-accurate HDL simulators, such as Verilator or the built-in simulator of Chisel, or runs over a FPGA board. We ignore any timing inconsistency that may occur between considered tools.

3.3 Verifying timing invariants

Following the aforementioned workflow, one may wonder why the timing behavior added to the formal SW model is not directly compared against such executions of circuits and thus considered as our timing model. These ad-hoc timings within the formal SW model are only compared to traces of an ISS augmented with a cycle-level timing behaviors. The verification process would thus be limited to trace equivalence, with thus a limited coverage. While trace coverage can be easily computed on the SW side for the functional part, the diversity of timing behaviors on the HW side would be simply ignored.

The timing behaviors of the HW and the SW models must instead be verified together to establish the function which we name a timing model. Our verification thus proceeds as follows. The ad-hoc timings of the SW model are first encoded as *timing invariants* in the formal HW model. To achieve this, we identify for each instruction the sequence of hardware updates on the processor, such as changes on wires and registers. For a non-pipelined processor, a simple input-ouput relation is defined for each timing invariant. The input represents the instruction fetch, i.e. when the opcode is taken into account, whereas the output is instruction-specific, e.g., a memory load terminates with the correct value in the corresponding register, an ALU instruction modifies the accumulator register, etc.. The input-output relation depends on the current clock cycle. A timing invariant should be read as follows: x cycles after an instruction is fetched, where x is given by the formal SW model, the instruction terminates its execution.

Let us denote a timing invariant by $TInv_i$, where i is an instruction from the language $\mathcal{L}$, $i \in \mathcal{L}$. A timing model for an architecture A is thus defined as:

$$TM_A = (\bigwedge_{i \in \mathcal{L}} TInv_i)$$

The corresponding assertions are then verified via model checking. Finally, we perform a timing anomaly detection, also via model checking as in [3, 11] to establish, together with the verified timings, the co-validation of the timing model.

4 Case study: Lipsi processor

We now illustrate our approach described in the previous section over a case study. We select a very simple processor, the Lipsi processor [37], to better illustrate the various steps of our approach. In this section, we thus first briefly describe the Lipsi processor. Then, we present its formal ISA and HDL models using, respectively, the Sail and TLA$^+$ languages.

4.1 Overview

Lipsi is a tiny sequential 8-bit accumulator-based processor to be used in auxiliary functions or for teaching purpose. The ISA of Lipsi includes ALU operations using registers or immediate operands, load/store from/to the memory, unconditional and conditional branch, and an input/output (i/o) operation. A complete list of instructions and their encodings are shown in [37], Table 1. Instructions of Lipsi are encoded using a single byte, except branch operations and ALU operations with immediate operands. For these instructions, a second byte is used to store either the address of the target branch or the value of the immediate operand.

On the hardware side, Lipsi consists of an accumulator register (A), a program counter (PC), 16 additional registers and a single on-chip memory. Its datapath is shown in Figure 3. Lipsi targets the use of a single block RAM in FPGA, which can be as low as 512 bytes. Addresses are thus 9-bits values. The memory is accessible through 2 ports: one for reads, the other one for writes. The lower half of the 9-bit memory space therefore stores up to 256 bytes of instructions, while the upper half stores first 16 additional registers ($R[x]$) followed by up to 240 bytes of data. $R[X]$ can be used to store intermediate results when performing ALU operations. The specification of Lipsi allows to identify up to 16 different ports using bits [3:0] in the encoding of the i/o instruction. However, its current hardware implementation only uses 2 ports to exchange values with the accumulator A: one for outputting the value of A and the other one to load a new value in A. The hardware implementation of Lipsi is written in Chisel, and it has been synthesized to the Cyclon IV FPGA of the DE2-115 FPGA board. An Instruction Set Simulator (ISS), written in Scala, is also available.

The hardware implementation of Lipsi comes with a very simple timing model, as a single memory is connected to the processor. Two clock cycles are required

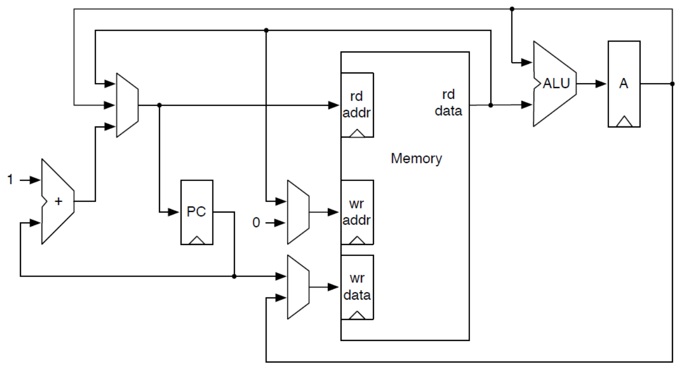

Fig. 3: The datapath of Lipsi, using an accumulator A, a single memory, an ALU and a program counter PC (extracted from [37]).

to execute an ALU instruction: one for fetching the instruction and one for accessing the data and executing the ALU operation. Loading A with a value in $R[x]$ also takes 2 cycles, while writing to $R[x]$ only takes 1 cycle due to the separated read/write ports to the memory. Updating $R[x]$ is performed while the next instruction is being from the read port. Memory store and load operations uses the additional registers $R[x]$ to store the targeted memory address. Those operations thus perform three memory accesses: one for fetching the instruction, another to retrieve the memory address from $R[x]$, and finally a last one to perform the memory operation at the specified memory address. A memory load thus takes 3 cycles, while a memory write takes only 2 cycles as the last access occurs meanwhile the next instruction is being fetched. Finally, the i/o operation takes only 1 cycle.

4.2 Formal SW model

We now present the formal SW model of Lipsi using the Sail language. We first define the architectural state of Lipsi, i.e. its accumulator `A`, its program counter `PC` and the ports used by the i/o instruction, i.e. `din` and `dout` for respectively the input and output ports. The `nextPC` register is used to store the address of the branch, i.e. the second byte of a branch instruction, when it is decoded. All these variables are 8-bits registers, as in the hardware implementation of Lipsi.

```
type len_t = bits(8) /* 8-bit architecture */
register A : len_t /* Accumulator */
register PC : len_t /* Program Counter */
register nextPC : len_t /* For branch instructions */
register din : len_t /* For the i/o instruction, input port */
register dout : len_t /* output port */
```

Memory model. The structure `Memory` represents the memory of Lipsi. It embeds respectively the instruction and data spaces, which are defined as a vector of bytes. These vectors are organized in downward memory addresses. Finally, a vector of registers `Rs` represents the additional registers $R[x]$ of Lipsi.

```
type memory_data = vector(256, dec, bits(8)) /* Data space */
type memory_inst = vector(240, dec, bits(8)) /* Instruction space */
struct Memory = { Inst : memory_inst, Data : memory_data }
register Rs : vector(16, dec, bits(8)) /* R[x] */
```

We now show the formal specification of the write operations, for both the memory space but also for $R[x]$. The function `mem_write` updates the content of `Memory` with the value `v` at the memory address `adr`. Either the instruction or the data vector of `Memory` get updated, depending on the value of the Most Significant Bit (MSB) of `adr`, a 9-bit value. Note that the data vector is updated only if `adr` does not target `Rs`.

```
val mem_write : (bits(9), bits(8), Memory) -> vector(256, dec, bits(8))
function mem_write (adr, v, mem) = {
  if (adr[8] == 0b1) then {
    if (adrbits_to_adrno(adr[7..0]) >= 16) then
      plain_vector_update (mem.Data, length(mem.Data) - 1 -
          ↪ adrbits_to_adrno(adr[7..0]), v);
    else return mem.Data;
  } else {
    plain_vector_update(mem.Inst, length(mem.Inst) - 1 - adrbits_to_adrno
        ↪ (adr[7..0]),v);
  }}
```

The function `reg_write` updates $R[x]$ and shares with the function `mem_write` a similar signature. A string representing the name of register, noted `r`, is however used instead of an memory address. `r` is mapped into an offset in the data vector using the functions `reg_name` and `regbits_to_regno`. `X` is the setter function to update `Rs` with the value `v`, that uses the overload feature of Sail to abstract read (not shown) and write accesses (function `wX`, signature not shown).

```
function wX (r, v) = if r < 16 then { Rs[15 - r] = v; }
overload X = {rX, wX}
val reg_write : (string, bits(8)) -> unit /* unit equivalent to void */
function reg_write (r, v) = {
  X(regbits_to_regno(reg_name(r)), v); }
```

Finally, the overload feature of Sail is used to abstract the organization of the memory. Writing to $R[x]$ or the memory to implement the semantic of instruction is performed by simply calling the function `lipsi_write`. Similar functions are used for read operations (shown in the next paragraph).

```
overload lipsi_write = {mem_write, reg_write}
```

Instruction and timing models. We now present the part describing the semantic of instructions. We have modeled in Sail all the ISA of Lipsi, as presented in [37] (Table 1). We only show the use of Sail to decode and execute ALU instructions that rely on registers. First, the syntactic sugar of scattered definitions is used to group functions related to each instruction in one place, i.e. AST union `ast`, mapping function `decode` and function `execute`. The AST type `ALU_TYPE_REG` represents the considered ALU instructions. The mapping `encdec_alu_func_reg` matches a binary value to a constant value representing the requested ALU operation. The mapping `decode` matches the machine code of instructions to the associated AST node within `ast`. The concatenation operator `@` is used to extract, from the input bitvector, the requested ALU operation (`func`) and the index of the additional register (`reg`). Finally, the function `execute` implements the semantic of the instructions by first reading the value from the specified additional register, i.e. `reg_val` and then performing the specified ALU operation on `reg_val` and A. `accureg` is a an accessor to A, for reading or writing.

```
scattered union ast
scattered mapping decode
scattered function execute

union clause ast = ALU_TYPE_Reg : (alu_func_reg, regbits)

mapping encdec_alu_func_reg: alu_func_reg <-> bits(3) = {
  LIPSI_ADD <-> 0b000,
  LIPSI_SUB <-> 0b001,
  ...
  LIPSI_XOR <-> 0b110,
  LIPSI_LD <-> 0b111
}
mapping clause decode = ALU_TYPE_Reg(func, reg) <->
       0b0 @ encdec_alu_func_reg(func) @ reg

function clause execute ALU_TYPE_Reg(func, reg) = {
  let reg_val : len_t = lipsi_read(regbits_to_regno(reg));
  let ret : len_t = match func {
    LIPSI_ADD => reg_val + accureg(),
    LIPSI_SUB => accureg() - reg_val,
    ...
    LIPSI_XOR => reg_val ^ accureg(),
    LIPSI_LD => reg_val
  };
  accureg(ret);
}
```

For the timing model, we simply use a register to represent clock cycles. This register is incremented by the clock cycles associated to each instruction being decoded. The formal SW model executes instructions in single steps, which is also equivalent to an instruction-level simulation of the input program. However this clock register tracks in a cycle-accurate manner the timing behavior of each instruction.

4.3 Formal HW model

We now present the formal HW model of Lipsi processor specified in the TLA$^+$ language. We partially present the specification, using actual code snapshots of both TLA$^+$ and Chisel codes, in Fig. 4 - 6. Our presentation emphasizes the traceability between the two semantics representations. TLA$^+$ being a specification framework, it lacks of program-specific infrastructure such as parsing, rich built-in libraries and AST manipulation. Hence, we also include several workarounds to preserve the traceability, such as convenient naming for instance. However, certain information is inevitably less obvious. For example, the type system of Chisel is abstracted in our TLA$^+$ model, and implicitly the type inference whose results are manually encoded in the corresponding TLA$^+$ specification. Our purpose is not to accurately provide a formal executable semantics of Chisel in TLA$^+$, but to specify, as precisely as possible, the Lipsi Chisel code.

We first present the memory model of Lipsi processor, in Fig. 4, its integration in the datapath, in Fig. 5 and finally, in Fig. 6 a representative snapshot of Lipsi processor. Our TLA$^+$ model of Lipsi is cycle-accurate and captures the execution of a Lipsi instruction through the circuit, with both combinational and sequential elements being encoded. The TLA$^+$ specification, `LipsiSpec` consists of the initial state, `LipsiInit` and the state transformer `LispiTrans`, the temporal operator ”always”, `[]` preceding the state transformer. It is applied on the system state `LVars` (i.e., the set of wires and registers):

```
LipsiSpec == LipsiInit /\ [] [LipsiTrans]_LVars
```

Before detailing `LispiSpec`, let us present the specification of the memory model.

Memory model. The TLA$^+$ specification of the Lipsi memory model, in Figure 4, adheres to this specification style (e.g., with `MemoryInit` and `MemoryTrans` being presented). On the Chisel side, the `Memory` class declares a memory zone,

```
class Memory(prog: String) extends Module {
  val mem = Mem(256, UInt(8.W))
  val data = mem(rdAddrReg(7, 0))
  when(io.wrEna) { mem(io.wrAddr) := io.wrData } ...
}
```

```
MemoryInit == ... /\ rdAddrReg = 0 /\ mem = mem_init /\ data = 0
MemoryTrans == ...
  /\ mem' = IF cond_io_wrEna
            THEN [ n \in 0..255 |->
              IF n = io.wrAddr THEN io.wrData ELSE mem[n] ]
            ELSE mem
  /\ rdAddrReg' = io.rdAddr
  /\ data' = mem[rdAddrReg]
```

Fig. 4: From Chisel HDL to TLA$^+$ for defining the memory system of Lipsi.

`mem`, to store data of a total size `256` and elements of type unsigned int of size 8 (e.g., `UInt(8.W)`). `data` is used to access to an element of `mem` and is initialized with the memory value at the address from the register `rdAddrReg` (a 9-bit address truncated to 8-bit as `mem` represents the data space of Lipsi). The equivalent TLA$^+$ code features the initialization, `MemoryInit`, of the state elements `mem`, `data` and `rdAddrReg` with their respective initial values. Note that `mem_init`, not listed, initializes `mem` with zeros. The Lipsi memory model features read/write memory operations, which are grouped in an interface `io`, whose fields are accessed using ".". For example, the Chisel code for a memory write at address `io.wrAddr` with value `io.wrData` is conditionally performed when memory write is enabled, i.e. `io.wrEna` is true. Its equivalent transition in the TLA$^+$ specification updates primed `mem` with value `io.wrData`, only when memory write is enabled (i.e., the predicate `cond_io_wrEna` tests if `io.wrEna` is true). Similarly, the new values, `rdAddrReg'` and `data'` are accordingly provided. Finally, class `Memory` is parameterized by `prog`, an input binary program, which is stored in a read only memory. These details are omitted due to space constraint.

```
class Lipsi(prog: String) extends Module {
  val mem = Module(new Memory(prog)) ...
}
```

```
LOCAL Lmem == INSTANCE Memory WITH ..., rdAddrReg <- 0,
                data <- 0, mem <- [ n \in 256..512 |-> 0 ]

LipsiInit == LET memory_state_init == Lmem!memory_init (program) IN
               /\ ... /\ mem = memory_state_init.mem
LipsiTrans == LET memory_state == Lmem!update_memory_state (...) IN
               /\ ... /\ mem' = memory_state.mem
```

Fig. 5: From Chisel HDL to TLA$^+$ when integrating the memory system of Lipsi.

Figure 5 shows the integration of this memory model and how we preserve the modularity of the Chisel code in our TLA$^+$ specification. The class `Lipsi` instantiates a local state variable `mem` with the memory model from Fig. 4. The actual creation of the memory object in Chisel (using `new`) is translated to a local instance `Lmem` of `Memory`, using the construct `INSTANCE WITH`. `LipsiInit` can then use `Lmem` and apply the function `memory_init` to set its initial state. Similarly, `LipsiTrans` uses the function `update_memory_state` to apply the state transformer of the memory model. This updates at each cycle the memory system `mem'`. These two functions `memory_init` and `update_memory_state` are the equivalent of `MemoryInit` and `MemoryTrans` from Fig. 4, following a function-based integration approach possible in TLA$^+$. Finally, the parametrization, with respect to the initial program, from Chisel is preserved in TLA$^+$ via the `memory_init` function.

```
val fetch::execute::stind::ldind1::ldind2::exit::Nil = Enum(6)
val stateReg = RegInit(fetch)
val exitReg = RegInit(false.B)

switch(stateReg) {
  is (fetch) {
    stateReg := execute ...
    when (rdData(7) === 0.U) {... enaAccuReg := true.B ...}
    ...
    when (rdData === 0xf0.U) {
      outReg := accuReg enaIoReg := true.B stateReg := fetch }
    when (rdData === 0xff.U) { stateReg := exit }
  } ...
  is (stind) { wrEna := true.B stateReg := fetch }
  is (execute) { stateReg := fetch }
  is (exit) { exitReg := true.B }
}
```

```
LOCAL fetch == 0 ... LOCAL exit == 5

cond_rdData_eq_Bits0 (rddata) == b7(rddata) = 0
cond_rdData_eq_Bits0xf0 (rddata) == rddata = 240
cond_rdData_eq_Bits0xff (rddata) == rddata = 255

cond_stateReg_eq_fetch == stateReg = fetch
cond_stateReg_eq_stind == stateReg = stind
cond_stateReg_eq_execute == stateReg = execute
cond_stateReg_eq_exit == stateReg = exit
...
LipsiInit == ... /\ stateReg = fetch /\ exitReg = FALSE

LipsiTrans == ...
  /\ exitReg' = IF cond_stateReg_eq_exit THEN TRUE ELSE FALSE
  /\ outReg' =
     IF cond_stateReg_eq_fetch
     THEN IF cond_rdData_eq_Bits0xf0 (memory_state.io.rdData)
          THEN accuReg ELSE outReg
     ELSE outReg
  /\ stateReg' = ...
     IF cond_stateReg_eq_fetch
     THEN ... IF X_cond_rdData_eq_Bits0xff (memory_state.io.rdData)
              THEN exit ELSE fetch
     ELSE IF cond_stateReg_eq_stind
          THEN fetch ELSE IF cond_stateReg_eq_execute ...
```

Fig. 6: From Chisel HDL to TLA$^+$ for snapshot of Lipsi FSM.

Instruction and timing models. Figure 6 presents a snapshot of how Lipsi executes instructions. Compared to the formal SW model, an instruction is here executed in potentially several clock cycles, i.e. cycle-level execution, by updating corresponding values in the wires and the registers of Lipsi. Lipsi is implemented with a Finite State Machine (FSM), whose states are first defined, represented as a list (`Enum`). The position in the list identifies the current state, captured in the register `stateReg` which is initialized to `fetch` (and equal to `0`). Such states are encoded in TLA$^+$ as a sequence of `LOCAL` declarations with explicit assignments corresponding to their position in the list, while initializing `stateReg` to `fetch` in `LipsiInit`. In a similar way, an exit flag (corresponding to the instruction `exit` in Lipsi ISA) is set accordingly in both Chisel and TLA$^+$.

The control logic of the FSM of Lipsi is handled by a `switch` statement over the current value of `stateReg`. When `is(fetch)` is true, the next state is by default set to `execute` (i.e., `stateReg := execute`) for the next cycle. However, any subsequent modification of `stateReg`, in the same clock cycle, overrides this modification, a behavior common to any HDL language. For example, if the instruction is an `io` (i.e., the opcode is equal to `0xf0.U`), `stateReg` is reset to `fetch` for the next cycle (while copying the value of the accumulator, `accuReg` to `outReg` and setting `enaIoReg` to true). The timing model of the `io` instruction is thus one clock cycle. The other states are used to store the output of the decode when an instruction is fetched. For instance, `is(stind)` enables the memory write, i.e. the signal `wrEna` is set to true, while resetting `stateReg` to `fetch`. The timing model of a store instruction is thus one clock cycle, as the write occurs at the next cycle while the next instruction is fetched. Similarly, `is(execute)` resets `stateReg`, implying that `execute` represents the last stage in the timing model of some instructions. Finally, `is(exit)` signals the end of the program execution, represented in Lipsi by a specialized register `exitReg`. Therefore, `exit` takes three clock cycles. At the first cycle, the instruction is identified during `is(fetch)` based on its opcode `0xff.U` and the next state of Lipsi set to `exit`. At the second clock cycle, `exitReg` is set to true, a change that would be visible at the third clock cycle. Contrary to other instructions, no further instruction can be fetched at this third clock cycle that we thus count in the timing behavior of the `exit` instruction.

The TLA$^+$ encoding of this code fragment is driven by a cascades of conditions on both the value of `stateReg` (e.g., `cond_stateReg_eq_`) and the instruction opcode (e.g., `cond_rdData_eq_Bits`). For example, `exitReg'` is set to true when the corresponding condition on `stateReg` is `exit` and `outReg'` is updated with the value in the accumulator, `accuReg` only when `stateReg` is `fetch` and the opcode is `0xf0` (i.e., `cond_rdData_eq_Bits0xf0` is true). Otherwise it stays unmodified, as `outReg`. In a similar way, `stateReg'` is updated with the corresponding value, capturing, in this way part of the timing model that we prove in the next section.

5 Evaluation results

We first report the discrepancies we found between the SW and the HW models of Lipsi, while applying our approach. These discrepancies concern not only the functional semantic but also on the timing semantic, justifying the need for formalization and verification of timing models. Finally, we present a use-case for such built timing models to detect interferences in a multi-core context.

Semantic discrepancies. We identified several semantics discrepancies when we performed the trace equivalence between traces from formal SW model and the simulators and the circuit (steps (2) and (3) on the SW side of our approach, see section 3.2). First the instructions `sh` and `brl`, specified in the ISA of Lipsi, are not implemented in the Chisel hardware design of Lipsi. Next, the instructions `adc` and `sbb` gives output that are equivalent to respectively the `add` and the `sub` instructions. However, these are known unfinished implementation of the ISA by the author of the Lipsi circuit.

A more interesting discrepancy concerns the i/o instruction. Its specification allows the first 4 bits of its encoding to be used to specify i/o ports. However, the hardware implementation of Lipsi only uses a single i/o port, the one with value index 0. Any other index value than 0 leads to a silent drop of the next instruction ($PC + 1$), i.e. the execution continues at $PC + 2$. Even if this unprocessed instruction leads to be interpreted as an ALU operation (default decoding), the value of the accumulator is not modified as it is guarded by a boolean value, not set by default. Note that contrary to the previous discrepancies, this difference was not explicitly documented in the hardware implementation of Lipsi.

Finally, while the instruction `exit` takes 3 cycles in the Lipsi circuit, it only takes 1 cycle in the Lipsi simulator. Our goal is to detect timing discrepancies and not to point out functionality issues in the considered design. However, such findings demonstrate that we can detect any kind of semantic discrepancy between the specified ISA and implementations of it.

Proving the timing model. We exemplify next the following timing invariants $TInv_i$ in Lipsi, with $i \in \{$ `exit`, `io`, `add`, `ld` $\}$ (`io` being the i/o instruction). We recall that formal HW model of Lipsi is cycle accurate and a cycle variable is incremented every time the system makes a transition. The opcode of the fetched instruction is stored in `curr_instr`. Also, for each invariant, we use a counter variable `x_`, which is initialized to `0` and incremented at each clock cycle. Then, this value is compared with the result of the augmented formal ISA to ensure that the timing on both SW- and HW-level are the same. Note that this counter variable is not necessary, but we opt to explicitly encode it for clarity (i.e., an alternative is to directly use the processor clock cycle).

Next, we present the invariant for the instruction `exit`.

```
inv_exit (curr_instr) == curr_instr = 255 /\ exitReg /\ x_exit <= 3
```

Informally, it states that, whenever an instruction corresponds to an `exit` (its opcode is `0xff` or `255`), the register `exitReg` becomes true 3 clock cycles later. This property follows the semantic of Chisel code from Fig. 6, with `curr_instr` corresponding to `rdData`, `stateReg := exit` triggers at the next clock cycle the assignment to `exitReg` guarded by the condition `is(exit)`, assignment observed at a further next clock cycle. The timing of instruction `exit` is thus 3 clock cycles (i.e., the condition `x_exit`) and proved using the TLC model checker.

The timing invariant of the instruction `io` is presented next.

```
inv_io (curr_instr) ==
   LET val == curr_instr - 240 IN
      /\ curr_instr >= 240 /\ curr_instr <= 254
      /\ ((enaIoReg /\ val = 0) \/ not (val = 0))
      /\ io.dout = val
      /\ x_io <= 1
```

This instruction covers a range of opcodes, depending on the i/o port identifier. For example, the instruction `0xf1` is an `io` (opcode `0xf0` or `240`) and the i/o port number `1`. The variable `val` in the $TInv_{io}$ represents the port number. With respect to i/o ports, the semantics of `io` is under-specified in [37]: the port number is not used by the instruction. Therefore, the implementation in the circuit only supports the use of port `0`, i.e. `0xf0` is the only accepted opcode, as seen previously. The invariant captures the following pattern: the instruction is identified as `io` and the port identifier is calculated in `val`, the register `enaIoReg` is updated to allow the actual port output. The timing of instruction `io` is proved to be 1 clock cycle.

Next, we present $TInv_{add}$.

```
inv_add (curr_instr, rddata) ==
   LET reg == curr_instr IN
      /\ curr_instr >= 0 /\ curr_instr <= 15
      /\ \/ (not (rddata = reg) /\ (not (reg = 0)) /\ enaAccuReg
         \/ not (res = reg)
      /\ x_add <= 2
```

Similarly to `io`, the opcodes for `add` include the value to be processed via the accumulator (i.e., the range is between `0x00` and `0x0f`, according to Table 1 from [37]). The instruction `add` executes in two cycles. In the first cycle, the instruction is fetched and the flag register `enaAccuReg` is set to true. In the second cycle, the actual addition is computed, with the result deposited in `res`. We use an auxiliary variable `reg` to distinguish between two cases, depending if the operand is `0` or not (and if the `res` remains the same, even after the addition is performed). $TInv_{add}$ does not include functional correctness, i.e. that the correct addition is performed, only that the accumulator is updated after two clock cycles. However, it could also be possible to have a functionality criterion included as well, as it is the case with the next timing invariant for a memory load instruction, `ld`.

The opcodes of `ld` are also in a range of values, the difference with `0x70` (or `112`) being the target memory address. This address is represented by `val`. This instruction takes two cycles. The classical fetch first, followed with the actual memory read, deposited in the accumulator, `accuReg`.

```
inv_ld (curr_instr) ==
   LET val == curr_instr - 112 IN
       /\ curr_instr >= 112 /\ curr_instr <= 127
       /\ accuReg = mem[val]
       /\ x_ld <= 2
```

We could also express $TInv_{ld}$ as $TInv_{add}$, i.e. without any functional checking, by replacing the actual functionality by only a check that the memory is accessed. The other timing invariants are expressed in a similar fashion and proved using the TLC model checker.

Detection of interferences. We briefly report a illustration of the possible use of the built timing model: the detection of memory interferences. We thus consider three different input binaries to be run in our formal SW model of Lipsi, in order to represent a multi-core setting of Lipsi. Each Lipsi core is identified by an index. Besides, as each Lipsi has its own private memory, we rely on the i/o instruction to emulate the access to a shared device. Note that we only emulate the access to a shared device, not the device itself nor its arbitration policy. If two i/o accesses occur at the same time, we thus assume that the one coming from the Lipsi with the smallest index wins the access. The other Lipsi cores continue their execution as if their accesses were valid, as we are mostly interested in timing properties of programs not in their functional correctness.

For the input binaries, we reuse the same memory model for the spacing of memory accesses as in [15]. In this model, tasks or programs are represented as a sequence of memory requests separated by a given number of processor clock cycles, representing the amount of computation that is performed between two memory accesses. We assume a composable computer architecture [14], which ensures that the distance between requests is independent from the execution of other tasks. The only interference between the independent tasks thus stems from accesses to the emulated shared device.

The sequences of memory requests of our three input programs are: (A: $2, 24, 12$), (B: $14, 4, 2$) and (C: $26, 6$). Program A is made of 2 loops and thus generates i/o accesses at the (absolute) times 2, 26 and 38. Program B is made of a single loop and generates i/o accesses at the (absolute) times 14, 18 and 20. Finally, program C is also made of a single loop and generates i/o accesses at the absolute (times) 26 and 32. It is then trivial to detect that an interference is going to occur at time 26 between program A and C. Our next step is to modify the input binaries, by adding appropriate `nop` instructions (ALU operations that do not change the current value of A) or by introducing a `delay` instruction, as in Patmos or PRET, in order to space out i/o accesses. While a straightforward algorithm can solve this problem for programs with a single path, the presence of multiple paths in input programs lead to an interesting optimization problem of minimizing the number of interferences.

6 Related Work

Traditionally, HW/SW co-verification methods [30, 29, 23, 19] consider SW to be represented in higher-level languages than our low-level approach (which is characteristic to worst-case timing reasoning [40]) and HW to be based on HDL languages. These works use model checking techniques and focus mostly on proving functionality properties, while we propose advancements on the timing properties, also using model checking. For example, C code and Verilog designs are verified together via bounded model checking, in [30], similarly with the co-verification technique from [29], where model checking is used on C code and hardware abstractions based on push-down systems. High-level code is co-verified with HDL designs, in [19], by a combination of BDD-based model checking for HW and partial order reduction for SW. All these approaches consider an implicit interface between hardware and software (i.e., we presented this interface under the name of hardware patterns). An explicit interface is expressed in [13] and integrated in the HW/SW co-verification procedure based on bounded model checking (i.e., as a side note, the application is considered at binary level).

Our approach increases the confidence in the formal semantics of ISA with the help of an ISS, providing a de facto procedure to actually verify an ISS. There are several works [5, 18], centered on the verification of ISS and its use in HW/SW co-verification. The ISS presented in [5] is symbolic and addresses both functional and timing properties in processors using assertions in a similar way with our procedure as another symbolic ISS, which is constructed over an instruction-level abstraction [18].

WCET analysis tools require a clear and explicit specification of timing models of architectures to estimate WCET [40]. One approach is to rely on a product of timed automata to model the timing behavior of hardware elements, such as pipelines, caches, etc [6, 9]. WCET estimates are then obtained using the UPPAAL model-checker. However, in both cases, the timing accuracy of the hand-made models, which are difficult to design, are unclear and simplified compared to the underlying micro-architecture. Another approach is to automatically extract timing information from HDL processor designs, either using static analysis, as in [36] or aiming for patterns of pipelining using abstract simulation, as in [32]. Both approach work directly on the processor code (in this case VHDL) as our approach, however we differ in the formal technique: static analysis and respectively model-checking.

Another approach is to extend classical ISA-level Architecture Description Languages (ADLs) [28], such as ArchC or Sim-nML, to include the specification of timing models at the micro-architecture level. The OTAWA WCET analysis framework relies on an extended Sim-nML to specify timing models of pipelines by describing resource allocations of instructions over pipeline stages, functional units and buffers [16]. In [24, 25], the ADL Expression language is used to describe both ISA but also contention and parallelism relations at the micro-architecture, i.e. the timing model. Execution graphs are then generated on which WCET analysis are performed. The LISA ADL language [31] also enables the assignment of operations to pipeline stages, potentially with delays. Compared to timed automata models, the ADL-based approach enables the specification of timing models at a high-level description, from which low-level network of timed automata can be generated. However, they still have no link with the timing model at the HDL level and timing anomalies are ignored.

Designing retargetable compilers has led to the design on processor models in order to extract instruction sets [8, 21, 33]. However, they rely on the use of reservation table to describe pipeline operations which is not timing-accurate to model pipeline hazards.

In a different area, [17] proposes a timing-abstract behavioral model of pipelines to increase IP reuse and reduce bugs when optimizing implementations, for instance when performing logic retiming. Cycle-level details, such as pipeline staging, are generated from a concise specification through a process called timing augmentation.

7 Conclusion and future work

We proposed a general methodology to reason about timing properties of CPSs by taking into account the interplay between HW and SW, both formally specified and verified. We also reported on how our methodology is applied on a simple processor called Lipsi towards defining and proving its timing model. Due to the formal nature of our approach we also discovered several semantic inconsistencies, both functional and temporal, between the specification and the implementation of Lipsi. Our methodology is demonstrated using a formal semantics of Lipsi ISA using the Sail language and a formal specification of the corresponding hardware implementation in Chisel code, using the TLA$^+$ reasoning framework.

We are currently pursuing several lines of research. First, we aim to automatically generate formal (TLA$^+$) models and the necessary timing invariants directly for the HDL code. Second, Lipsi is a simple processor, we have ongoing work on popular RISC-V designs, with complicated timing models due to pipelining, multi-level caches and speculation mechanism. Research communities on the WCET analysis [41] or synchronous languages [7] rely on traceability to transfer high-level semantics to low-level code [22, 34]. We are interested in leveraging high-level timing properties and traceability enhancements within our HW and SW timing models. Finally, we plan to address timing models for the communication part of the CPSs, which are currently abstracted away in our current proposition.

References

1. Armstrong, A., Bauereiss, T., Campbell, B., Reid, A., Gray, K.E., Norton, R.M., Mundkur, P., Wassell, M., French, J., Pulte, C., Flur, S., Stark, I., Krishnaswami,

N., Sewell, P.: ISA semantics for ARMv8-a, RISC-V, and CHERI-MIPS. PACMPL **3**(POPL), 71:1–71:31 (2019)

2. Asavoae, M.: K semantics for assembly languages: A case study. Electr. Notes Theor. Comput. Sci. **304**, 111–125 (2014)
3. Asavoae, M., Hedia, B.B., Jan, M.: Formal executable models for automatic detection of timing anomalies. In: 18th International Workshop on Worst-Case Execution Time Analysis, WCET 2018. pp. 2:1–2:13 (2018)
4. Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, R., Wawrzynek, J., Asanović, K.: Chisel: Constructing hardware in a scala embedded language. In: Proceedings of the 49th Annual Design Automation Conference. pp. 1216–1225. DAC '12, ACM (2012)
5. Beatty, D.L., Bryant, R.E.: Formally verifying a microprocessor using a simulation methodology. In: Proceedings of the 31st Conference on Design Automation, 1994. pp. 596–602 (1994)
6. Béchennec, J., Cassez, F.: Computation of WCET using program slicing and real-time model-checking. CoRR **abs/1105.1633** (2011), `http://arxiv.org/abs/1105.1633`
7. Benveniste, A., Caspi, P., Edwards, S.A., Halbwachs, N., Guernic, P.L., de Simone, R.: The synchronous languages 12 years later. Proceedings of the IEEE **91**(1), 64–83 (2003)
8. Bradlee, D.G., Henry, R.R., Eggers, S.J.: The marion system for retargetable instruction scheduling. SIGPLAN Not. **26**(6), 229–240 (May 1991)
9. Dalsgaard, A.E., Olesen, M.C., Toft, M., Hansen, R.R., Larsen, K.G.: METAMOC: Modular Execution Time Analysis using Model Checking. In: Lisper, B. (ed.) 10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010). OpenAccess Series in Informatics (OASIcs), vol. 15, pp. 113–123 (2010)
10. Dasgupta, S., Park, D., Kasampalis, T., Adve, V.S., Rosu, G.: A complete formal semantics of x86-64 user-level instruction set architecture. In: Proceedings of the 40th PLDI 2019. pp. 1133–1148 (2019)
11. Eisinger, J., Polian, I., Becker, B., Metzner, A., Thesing, S., Wilhelm, R.: Automatic identification of timing anomalies for cycle-accurate worst-case execution time analysis. In: Proceedings of the 9th IEEE Workshop on Design & Diagnostics of Electronic Circuits & Systems (DDECS 2006), Prague, Czech Republic, April 18-21, 2006. pp. 15–20 (2006)
12. Gordon, M.J.C.: The semantic challenge of verilog HDL. In: Proceedings, 10th Annual IEEE Symposium on Logic in Computer Science, San Diego, 26-29, 1995. pp. 136–145 (1995)
13. Große, D., Kühne, U., Drechsler, R.: HW/SW co-verification of embedded systems using bounded model checking. In: Proceedings of the 16th ACM Great Lakes Symposium on VLSI 2006. pp. 43–48 (2006)
14. Hahn, S., Reineke, J., Wilhelm, R.: Towards compositionality in execution time analysis: definition and challenges. SIGBED Review **12**(1), 28–36 (2015)
15. Hebbache, F., Jan, M., Brandner, F., Pautet, L.: Shedding the shackles of time-division multiplexing. In: 2018 IEEE Real-Time Systems Symposium, RTSS, 2018. pp. 456–468 (2018)
16. Herbegue, H., Filali, M., Cassé, H.: Formal architecture specification for time analysis. In: Maehle, E., Römer, K., Karl, W., Tovar, E. (eds.) Architecture of Computing Systems – ARCS 2014. pp. 98–110. Springer International Publishing, Cham (2014)

17. Hoover, S.F.: Timing-abstract circuit design in transaction-level verilog. In: 2017 IEEE International Conference on Computer Design (ICCD). pp. 525–532 (Nov 2017)
18. Huang, B., Zhang, H., Subramanyan, P., Vizel, Y., Gupta, A., Malik, S.: Instruction-level abstraction (ILA): A uniform specification for system-on-chip (soc) verification. ACM Trans. Design Autom. Electr. Syst. **24**(1), 10:1–10:24 (2019)
19. Kurshan, R.P., Levin, V., Minea, M., Peled, D.A., Yenigün, H.: Combining software and hardware verification techniques. Formal Methods in System Design **21**(3), 251–280 (2002)
20. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley Longman Publishing Co., Inc. (2002)
21. Leupers, R., Marwedel, P.: A bdd-based frontend for retargetable compilers. In: Proceedings the European Design and Test Conference. ED TC 1995. pp. 239–243 (March 1995)
22. Li, H., Puaut, I., Rohou, E.: Tracing flow information for tighter WCET estimation: Application to vectorization. In: 21st IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2015, Hong Kong, China, August 19-21, 2015. pp. 217–226 (2015)
23. Li, J., Xie, F., Ball, T., Levin, V., McGarvey, C.: An automata-theoretic approach to hardware/software co-verification. In: Fundamental Approaches to Software Engineering, 13th International Conference, FASE 2010. pp. 248–262 (2010)
24. Li, X., Roychoudhury, A., Mitra, T., Mishra, P., Cheng, X.: A retargetable software timing analyzer using architecture description language. In: 2007 Asia and South Pacific Design Automation Conference. pp. 396–401 (Jan 2007)
25. Li, X., Roychoudhury, A., Mitra, T.: Modeling out-of-order processors for wcet analysis. Real-Time Systems **34**(3), 195–227 (Nov 2006)
26. Liu, I., et al.: A PRET microarchitecture implementation with repeatable timing and competitive performance. In: 2012 IEEE 30th international conference on computer design (ICCD). pp. 87–93. IEEE (2012)
27. Meredith, P.O., Katelman, M., Meseguer, J., Rosu, G.: A formal executable semantics of verilog. In: 8th ACM/IEEE MEMOCODE 2010, Grenoble, France, 2010. pp. 179–188 (2010)
28. Mishra, P., Dutt, N. (eds.): Processor Description Languages, application and methodologies, vol. 1 in Systems on Silicon. Morgan Kaufman (2008)
29. Monniaux, D.: Verification of device drivers and intelligent controllers: a case study. In: Proceedings of the 7th ACM & IEEE International conference on Embedded software, EMSOFT 2007. pp. 30–36 (2007)
30. Mukherjee, R., Purandare, M., Polig, R., Kroening, D.: Formal techniques for effective co-verification of hardware/software co-designs. In: Proceedings of the 54th Annual Design Automation Conference, DAC 2017. pp. 35:1–35:6 (2017)
31. Pees, S., Hoffmann, A., Zivojnovic, V., Meyr, H.: Lisa-machine description language for cycle-accurate models of programmable dsp architectures. In: Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361). pp. 933–938 (June 1999)
32. Pister, M.: Timing model derivation: pipeline analyzer generation from hardware description languages. Ph.D. thesis, Saarland University (2012)
33. Rau, B.R., Kathail, V., Aditya, S.: Machine-description driven compilers for epic and vliw processors. Design Automation for Embedded Systems **4**(2), 71–118 (Mar 1999)
34. Raymond, P., Maiza, C., Parent-Vigouroux, C., Carrier, F., Asavoae, M.: Timing analysis enhancement for synchronous program. Real-Time Systems **51**(2), 192–220 (2015)

35. Reineke, J., Wachter, B., Thesing, S., Wilhelm, R., Polian, I., Eisinger, J., Becker, B.: A definition and classification of timing anomalies. In: 6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis (2006)
36. Schlickling, M.: Timing model derivation: static analysis of hardware description languages. Ph.D. thesis, Saarland University (2013)
37. Schoeberl, M.: Lipsi: Probably the smallest processor in the world. In: Architecture of Computing Systems - ARCS 2018 - 31st International Conference. pp. 18–30 (2018)
38. Schoeberl, M., Puffitsch, W., Hepp, S., Huber, B., Prokesch, D.: Patmos: A time-predictable microprocessor. Real-Time Systems **54(2)**, 389–423 (Apr 2018). https://doi.org/10.1007/s11241-018-9300-4
39. Thiele, L., Wilhelm, R.: Design for timing predictability. Real-Time Systems **28**(2-3), 157–177 (2004)
40. Wilhelm, R.: Formal analysis of processor timing models. In: Model Checking Software, 11th International SPIN Workshop. pp. 1–4 (2004)
41. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D.B., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P.P., Staschulat, J., Stenström, P.: The worst-case execution-time problem - overview of methods and survey of tools. ACM Trans. Embedded Comput. Syst. **7**(3), 36:1–36:53 (2008)
42. Wolf, C.: A RISC-V Formal Verification Framework. https://github.com/cliffordwolf/riscv-formal
43. Yu, Y., Manolios, P., Lamport, L.: Model checking tla+ specifications. In: 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods. pp. 54–66. CHARME '99 (1999)