Minimally Invasive Generation of RISC-V Instruction Set Simulators from Formal ISA Models

Sören Tempel[1] Tobias Brandt Christoph Lüth[1,2] Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
tempel@uni-bremen.de, tobbra91@gmail.com, christoph.lueth@dfki.de, drechsler@uni-bremen.de

***Abstract*—The development process for new embedded systems relies increasingly on simulation, e.g. to develop hardware and software components in parallel using virtual prototyping. The central component of a virtual prototype is the instruction set simulator (ISS) which implements instruction execution for a specific instruction set architecture (ISA). To avoid erroneous behavior during software simulation, it is paramount to ensure that the provided ISS implements the ISA exactly as specified, i.e. that there are no discrepancies between the hardware and the VP. In order to increase confidence in the correctness of the VP's ISS, it is advantageous to generate it automatically from a formal model of the ISA instead of implementing it manually. While a variety of formal ISA models have been proposed in prior work, they are presently not widely used in the VP domain. We attempt to ease employment of formal models for ISS generation in this domain. To this end, we reduce the integration effort through a simulator-agnostic ISS generation approach that integrates well with existing simulators and existing vendor-supplied VP components. Our approach leverages a formal RISC-V ISA model which exclusively describes instruction semantics and abstracts interactions with hardware components through an interface model, thus encapsulating interactions with simulator-specific code. As part of our experiments, we were able to generate an ISS for the popular RISC-V implementations Spike and RISC-V VP, thereby replacing their manually written implementations. Performed benchmarks indicate that the generated ISS offers the same simulation performance as a manually written one, while still passing the official RISC-V tests.**

***Index Terms*—Embedded Systems, Formal ISA Models, Simulation, RISC-V, Virtual Prototyping**

I. Introduction

As embedded systems consist of both hardware and software components, it is vital to be able to begin software development before the hardware is available to reduce the time-to-market. In order to do so, a simulator for the hardware is required. Nowadays, hardware and software components are typically developed in parallel using virtual prototypes [1, Sect. 1.3]. A virtual prototype (VP) provides a simulator for an entire hardware platform. A central component of a VP is the instruction set simulator (ISS) which is responsible for simulating instruction execution for a chosen instruction set architecture (ISA). For a VP-based development flow, it is paramount to ensure that the VP implements the ISA exactly as specified. Otherwise, the software may exhibit erroneous behavior when executed on the physical hardware, thereby negatively impacting the time-to-market of an embedded system. In order to increase confidence in the correctness of the VP, it is advantageous to derive its central part—the ISS—from a formal model of the ISA rather than implement it manually. A formal model describes ISA semantics using a well-defined formal language, instead of relying on natural language, thus achieving exactness and avoiding ambiguities in the specification. Instead of manually implementing an ISS from a specification in natural language, it is possible to automatically generate it from such a formal model, thus reducing the margin for errors.

While several formal ISA models have been presented in prior work [2, 3, 4, 5, 6], they are presently not utilized in the VP domain. We attribute this to the fact that VPs are tightly integrated with vendor-supplied components (e.g. memory implementations, bus systems, or models of hardware peripherals). These components are often modeled in SystemC, a C++ library for hardware modeling [7]. When generating an ISS for a VP from a formal model, it must be ensured that the generated ISS integrates well with these vendor-supplied SystemC components and does not require replacing or changing them. For this reason, we propose a novel approach for generating an ISS from a formal model which—contrary to prior work—is designed to be minimally invasive (i.e. allows re-using existing vendor-supplied components). This is achieved by leveraging a minimal formal model which focuses exclusively on formally describing the instruction semantics and by abstracting interactions with hardware components through a generic interface model. That is, we only generate the code implementing the instruction semantics, which is where prior work has found most bugs [8], leaving other components (e.g. those modeling the memory) as-is to ease the integration. While our proposed approach is applicable to different ISAs, we focus on the modular RISC-V [9, 10] architecture in this publication. Due to an ever-growing set of ISA extensions, simulators for this ISA benefit significantly from formal models as the specification is constantly expanding, requiring simulators to "catch up" by implementing new extensions, which is a laborious error-prone process.

The goal of our work is therefore to reduce the effort required to integrate existing RISC-V simulators with formal ISA models. Our contributions towards this goal are: (1) An enhanced version of an existing formal model for the RISC-V ISA, (2) a new simulator-agnostic C/C++ code generator for this formal model, and (3) a modified version of the popular Spike [11] and RISC-V VP [12] simulators which use a

This work was supported by the German Federal Ministry of Education and Research (BMBF) within projects Scale4Edge under grant no. 16ME0127, ECXL under grant no. 01IW22002, and VE-HEP under grant no. 16KIS1342.

generated ISS (instead of a manually written one). To the best of our knowledge, the ISS generation approach presented here is the first which is easily applicable to a variety of existing RISC-V simulators. The experiments we have conducted with Spike and RISC-V VP confirm the feasibility of our approach for this purpose. Furthermore, performed benchmarks indicate that an ISS generated using our tooling achieves similar simulation speed compared to a manually written one while still passing the official RISC-V ISA tests.

II. Preliminaries

In the following, we provide background information on formal ISA models and VPs as prerequisites for our research.

A. Formal ISA Model

The ISA is the interface between the hardware and the software of a system; it is therefore of central importance to the system architecture. An ISA description has many aspects: instruction semantics, memory and register behavior, interrupts, decoding, et cetera. In this publication, we are focusing exclusively on instruction semantics. These semantics have traditionally been specified in natural language. However, natural language specifications can be ambiguous, incomplete, and are not easy to work with. For this reason, we base our work on a *formal* ISA model: one which has unambiguous semantics and can be processed by electronic means (e.g. for the purpose of code generation).

Formal models come in a variety of languages (and we give a more comprehensive overview in Sect. V), but for our work we leverage an existing formal model in the general-purpose functional programming language Haskell. This model is called LibRISCV [6] and provides formal semantics for the 32-bit base instruction set of the RISC-V architecture [10, Sect. 2]. Contrary to prior work (e.g. Sail [3]), LibRISCV focuses exclusively on describing the user-level instruction semantics in isolation, e.g. without formally describing other aspects (such as the memory). Furthermore, LibRISCV does not capture microarchitecture details such as pipelining or timing. This property makes LibRISCV well suited for our approach as it eases the integration with existing simulators by allowing re-use of existing (vendor-supplied) components such as memory implementations, instead of also generating these components from the formal model.

The semantics of an individual RISC-V instruction are described formally in LibRISCV through an embedded domain-specific language (EDSL) in Haskell. Conceptually, the EDSL consists of two components: (1) primitives for describing interactions with architectural state components (e.g. the memory) and (2) an expression language for performing operations on memory/register values. Using these components (i.e. the EDSL), LibRISCV provides a formal description for each RISC-V instruction (analog to the natural language specification in the RISC-V standard). For example, the `ADD` instruction is described through the primitive `readRegister` (to obtain the register operands), the addition operation of the expression language, and the `writeRegister` primitive (for storing the result). Similar primitives are available for other operations (e.g. interactions with the memory or the program counter). In total, the LibRISCV EDSL consists of 26 primitives for formally describing instruction semantics. Formal descriptions using this EDSL can be further processed in Haskell, e.g. for the purpose of code generation by mapping the 26 primitives and the expression language to C/C++ code (see Subsect. III-D). Internally, the LibRISCV EDSL is implemented in Haskell using free monads [13]. More details regarding the utilization of free monads and LibRISCV itself are provided in an existing publication by Tempel *et al.* [6].

B. Virtual Prototypes

VPs provide an executable model of an entire hardware platform, including peripherals provided by this platform. They are a popular tool for the development of embedded systems as the early creation of VPs enables developing both—hardware and software components—for an embedded system in parallel, thereby reducing the time-to-market [1, Sect. 1.3]. This is achieved by utilizing the VP to simulate the behavior of the targeted hardware platform, thus allowing software development to begin before the physical hardware is available. For software simulation, a VP therefore provides an ISS for the architecture used by the targeted hardware platform. Our work is concerned with the generation of this component from a formal ISA model. Apart from an ISS, a VP also provides models for peripherals provided by the hardware platform. Hardware peripherals are commonly modeled using SystemC [7], a C++ class library for describing hardware components. More specifically, VPs often utilize SystemC TLM [7, Sect. 9], where hardware behavior is described based on a high-level bus abstraction. Our ISS generation approach is specifically designed to integrate well with existing SystemC components. In order to evaluate our approach, we use RISC-V VP, an existing open source SystemC-based VP for the RISC-V architecture [14].

III. Approach

In the following, we present our approach for minimally invasive generation of instruction set simulators from formal ISA models.

A. Overview

Fig. 1 provides an overview of our approach and includes an illustration of the software architecture of an ISS with some VP-specific components (e.g. a SystemC TLM bus). Components added for the application of our approach are highlighted using a dashed box. The ISS in Fig. 1 consists of different internal components and is responsible for executing a firmware image as faithful to a real processor as possible. Regarding the internal ISS components, we differentiate between the instruction execution unit (which is responsible for the execution part of the fetch-decode-execute cycle) and architectural state components (e.g. the register file) which are required for instruction execution but conceptually separate components. As motivated in Sect. I, architectural state components are commonly supplied by hardware vendors and therefore often modeled using the SystemC standard. For example, in Fig. 1 the memory component is modeled using SystemC TLM and therefore attached to a TLM bus which is accessed by the

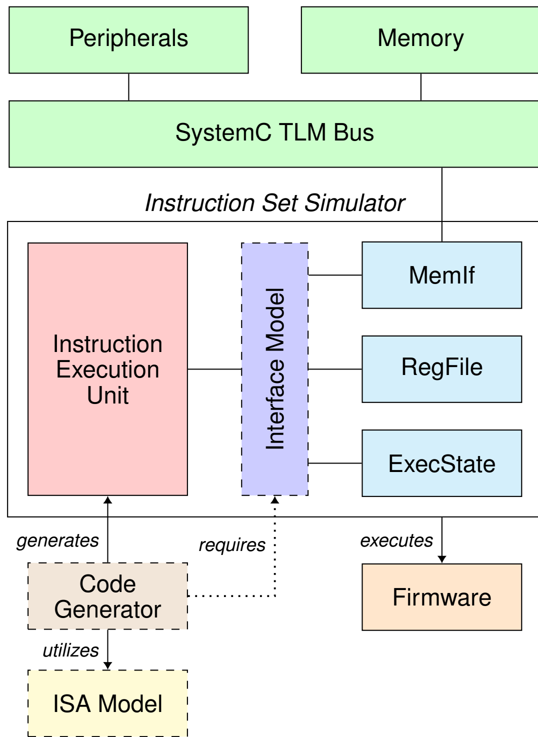

Fig. 1. Overview of our minimally invasive ISS generation approach.

execution unit over a memory interface (MemIf). Our proposed ISS generation approach is specifically designed to integrate well with existing vendor-supplied components. Therefore, we focus on generating the execution unit of an ISS from a formal ISA model. The execution unit implements the instruction semantics and is the component in which prior work on automated testing has found the most implementation errors in existing simulators [8]. In order to generate this component, the code generation tool needs to be able to emit code that interacts with the architectural state components to generate code which implements the instruction semantics (e.g. to write a register). Since the API of these components is highly simulator- and vendor-specific, we leverage a custom interface model for our approach. This interface model provides a generic API for common operations (e.g. writing/reading registers or accessing memory). The generic API itself is a set of C function prototypes which define a simulator-agnostic interface for performing these common operations (see Subsect. III-C). These functions need to be implemented on a per-simulator basis by mapping them to the internal interfaces provided by the simulator. Since simulator-specific code is abstracted through the generic API, the code generation tool is itself applicable to different RISC-V simulators (see Subsect. IV-A). While we believe the outlined approach to be practical for different ISAs, we focus on the RISC-V architecture in this publication as it is highly modular and therefore well suited for an application of our approach. In the next subsection, we therefore present a significantly enhanced version of an existing formal RISC-V ISA model which has been tailored to our use case.

```
1 semantics LBInst{rd=dest, rs1=reg, imm=off} = do
2   base <- readRegister reg
3   byte <- loadByte (base `Add` off)
4   writeRegister dest (SExtByte byte)
```

Fig. 2. Simplified description of LB instruction semantics in LIBRISCV.

B. ISA Model

As discussed in Subsect. II-A, we are using the existing LIBRISCV formal ISA model for our approach [6]. We choose this model because—in contrast to prior work—it describes instructions semantics in isolation without providing a formal description of other ISA aspects such as memory behavior or decoding. This allows us to only generate the code implementing instruction semantics (the instruction execution unit) from the formal specification while retaining other parts as-is, thereby making our approach minimally invasive and easing the integration with existing simulators. As per Subsect. II-A, LIBRISCV leverages a Haskell EDSL for the formal description of RISC-V instruction semantics. This EDSL consists of two components: (1) primitives for describing interactions with architectural state components (e.g. the memory) and (2) an expression language for performing operations on memory/register values. As an example, the formal description of the RISC-V `LB` instruction in this EDSL is provided in Fig. 2. The semantics of this instruction are described in terms of the `readRegister` (Line 2), `loadByte` (Line 3), and `writeRegister` (Line 4) primitives which correspond to changes of the architectural state. Furthermore, operations on retrieved register/memory values are modeled using the aforementioned expression language, i.e. the `Add` and `SExtByte` constructors in Fig. 2. For the purpose of code generation, we need to map constructors of the LIBRISCV expression language to C/C++ expressions. Additionally, we need to map the `readRegister`, `writeRegister`, etc. primitives to functions provided by our interface model.

In order to do so, we further enhanced the existing ISA model for code generation purposes. The original version of LIBRISCV as presented by Tempel *et al.* [6] was intended for building custom ISA interpreters directly in Haskell. For this reason, it separates instruction decoding from instruction execution (i.e. the decoding is not part of the formal model). This can be illustrated by considering the formal semantics for the `LB` instruction in Fig. 2 again. The semantics of this instruction are defined over a record type constructor (`LBInst`) in Line 1 which represents a decoded `LB` instruction. The different members of this record type are assigned to variables; the values of these variables correspond directly to integer values (e.g. 15 for accessing register `x15`) and are hence not captured by the formal description. To overcome this limitation, we added additional primitives to LIBRISCV to express decoding operations as part of the instruction semantics descriptions. The resulting, enhanced description of the `LB` instruction is shown in Fig. 3. Contrary to the description from Fig. 2, this version is only parameterized over the instruction opcode (`LBOpcode`) and then uses the new primitives `decodeRD`, `decodeRS1`, and `decodeImmI` to obtain additional information about the current instruction

```
semantics LBOpcode = do
  dest <- decodeRD
  base <- decodeRS1 >>= readRegister
  off  <- decodeImmI

  byte <- loadByte (base `Add` off)
  writeRegister dest (SExtByte byte)
```

Fig. 3. Description of the LB instruction with our LibRISCV changes.

```
semantics LBOpcode = do
  (dest, base, off) <- decodeAndReadIType
  byte <- loadByte (base `Add` off)
  writeRegister dest (SExtByte byte)
```

Fig. 4. Final refinement of LB instruction semantics in LibRISCV.

(Line 2 - 4). Since the description is now more verbose, we added an abstraction to define the instruction type, as mandated by the RISC-V specification [9, Sect. 2.2], as part of the formal description. The actual description of the LB instruction—using our enhanced version of LibRISCV—is therefore less verbose and depicted in Fig. 4. Notably, it has the same length as the original description.

The new instruction decoding primitives that we have added to the existing LibRISCV ISA model allow us to map these to decoding functions provided by RISC-V simulators using our interface model. More details on interface modeling will be provided in the next subsection.

C. Interface Model

The interface model is the central perquisite for generating a simulator-agnostic ISS as the generated implementation of instruction semantics will need to interface with existing components of a simulator (e.g. the register file). Since the C/C++ code—emitted by our code generation tool—should be simulator-agnostic, we introduce the interface model as an additional abstraction layer within the simulator. The interface model provides a generic C/C++ API for accessing the aforementioned components, this API is used by the code generator tool and needs to be implemented manually once for each targeted simulator. An excerpt of the generic API is shown in Fig. 5, the full API description is available separately.[1] As illustrated in this figure, the API consists of a set of C functions which are parameterized over a void pointer. These void pointers are converted to simulator-specific types internally in the implementation of these functions. We decided against utilizing C++ abstractions (such as abstract classes) for this purpose to also support RISC-V simulators that are purely written in C. Presently, the generic API consists of 19 C functions and provides an interface for the register file, the program counter, the memory, and the decoder of a RISC-V simulator. Relying solely on a functional abstraction eases implementing this generic API as an implementation is essentially a mapping of the defined generic functions to simulator-specific ones. Therefore, these functions will be inlined by the C/C++ compiler in the common case and hence the additional interface model abstraction has minimal to no impact on simulation performance (see Subsect. IV-C). We

```
/* Register file */
uint32_t read_register(void *core, unsigned idx);
void write_register(void *core,
                    unsigned idx,
                    uint32_t value);

/* Byte-addressable memory */
uint8_t load_byte(void *core, uint32_t addr);
uint16_t load_half(void *core, uint32_t addr);
uint32_t load_word(void *core, uint32_t addr);
/* ... */
```

Fig. 5. Excerpt of the generic API provided by the interface model.

will further discuss the interface model implementation for Spike and RISC-V VP in Subsect. IV-A. In the following, we will introduce our simulator-agnostic code generation tool and illustrate how this tool interacts with the interface model.

D. Code Generation

We use the previously described ISA and interface models to implement a simulator-agnostic code generation tool. As depicted in Fig. 1, the tool generates a simulator-agnostic instruction execution unit, i.e. the code implementing the RISC-V instruction semantics. For this purpose, we build on the formal description of these semantics provided by LibRISCV. As discussed in Subsect. II-A, the formal ISA model consists conceptually of two components: primitives for describing interactions with the architectural state components and an expression language for describing operations on register/memory values which were obtained through these primitives. All instruction semantics are formally described using these components. In order to automatically generate code from this formal description, we need to build a code generator in Haskell which receives these EDSL components as inputs. As discussed in the original LibRISCV paper, the code generator then acts as an interpreter for the EDSL transforming its components into the desired representation [6, Sect. 4.3]. As part of this transformation, we generate C/C++ code for all 26 primitives of the EDSL, e.g. mapping the `readRegister` primitive to C/C++ code retrieving a register value through the interface model. Therefore, the desired representation is a C/C++ abstract syntax tree (AST) in our case. The creation of this AST from the formal ISA model is illustrated in Fig. 6.

As depicted in Fig. 6, C/C++ code implementing instruction semantics is created from this generated AST using an unparser (also called a pretty printer). Conceptually, an unparser is the opposite of a parser, as shown in Fig. 6: it serializes a given AST to a chosen output format, C/C++ source code in our case [15, 16, 17]. By employing an unparser we can ensure the syntactic correctness of the generated code, compared to—for example—generating the code directly through string concatenation. This enables straightforward adjustments of the generated code and eases the application of our approach to simulators written in other programming languages. The implementation of the unparser (i.e. the translation from the AST to the C code) makes use of the existing `language-c`[2]

[1] https://github.com/agra-uni-bremen/formal-iss/tree/fdl-2023#readme

[2] https://hackage.haskell.org/package/language-c

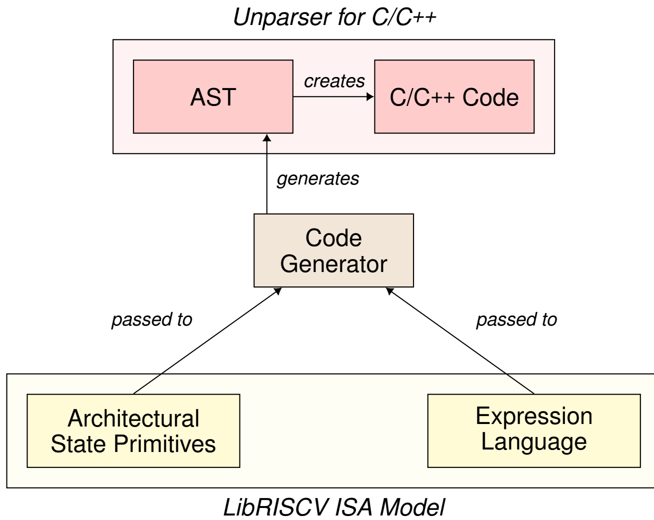

Fig. 6. Interaction between the code generator and the LIBRISCV ISA model.

```
1 static inline void exec_lb(void * core,
2                            uint32_t instrPC,
3                            void * instr)
4 {
5   write_register(core, instr_rd(instr),
6     (int32_t)(int8_t)load_byte(core,
7       read_register(core,
8         instr_rs1(instr))+instr_immI(instr)));
9 }
```

Fig. 7. Automatically generated C/C++ code for the LB instruction.

Haskell library. As shown in Fig. 6, our code generation tool is in this context responsible for generating an AST that is passed to the unparser provided by `language-c`. The generation of this AST is based on the formal instruction semantics obtained from LIBRISCV.

As part of the AST generation, we create one C function for each formally described RISC-V instruction. As an example, the C function which implements the `LB` instruction is shown in Fig. 7. Each generated function receives a void pointer to a simulator-specific processor abstraction (`core`), the program counter of the current instruction (`instrPC`), and a void pointer to a simulator-specific instruction abstraction (`instr`) as function arguments. Naturally, since the code is automatically generated, it heavily nests function calls and is not optimized for human readability. Nonetheless, it is possible to illustrate the interaction with the aforementioned generic API of the interface model using this example. The function body shown in Line 5 - 8 of Fig. 7 uses the `write_register`, `read_register`, and `load_byte` functions from the generic API (see Fig. 5) to interact with the register file and memory implementation. These functions receive the processor abstraction (`core`) as a void pointer function argument and cast this pointer to a simulation-specific type internally to implement the operation.[3] Furthermore, the generated code in Fig. 7 also obtains information about the instruction (register and immediate) using the `instr_rs1` and `instr_immI` functions of the interface model. Arithmetic operations are performed on these values by mapping the `Add` operation from LIBRISCV's expression language (see Fig. 4) to the + operator provided by C/C++. Similarly, the sign-extension from Fig. 4 (`SExtByte`) is implemented in Fig. 7 using integer type casts.

By leveraging the interface model, the code generation tool itself remains entirely simulator-agnostic. The tool is a standalone Haskell binary written in roughly 750 LOC which depends on the LIBRISCV Haskell library (for the formal RISC-V model) and the `language-c` library (for C/C++ unparsing). In the next section, we illustrate that we can easily employ this tool—and our general approach—for automatically generating an execution unit for different existing RISC-V simulators, thereby demonstrating that a minimally invasive ISS generation is feasible.

IV. EVALUATION

In the following, we evaluate our approach in terms of generalizability, conformance, and simulation performance. In this regard, we have concerned ourselves with the following research questions:

RQ1 Is the approach generalizable in the sense that it can be applied to different RISC-V simulators?

RQ2 Does the generated ISS conform to the instruction semantics mandated by the RISC-V specification?

RQ3 Does the original, manually written, ISS have better simulation performance than the generated one?

A. Generalizability

Our proposed ISS generation approach is specifically designed to be easily applicable to a variety of different RISC-V simulators. In order to evaluate the suitability of our approach for this purpose, we have employed it to generate a new ISS for the popular Spike [11] and RISC-V VP [12] simulators. The existing ISS of these simulators was manually written by the developers in C++ and was not generated from a formal specification. In the following, we provide more background information on these two simulators and describe the changes that were necessary in order to integrate them with our ISS generation approach.

Spike was one of the first simulators for the RISC-V architecture and was initially developed by the University of California. Similar to RISC-V VP, it simulates the execution of RISC-V machine code on a host system. In this regard, it focuses on achieving a high simulation speed at the cost of simulation accuracy. For this reason, it does not use a hardware modeling language like SystemC and therefore only has limited support for additional hardware peripherals. Contrary to Spike, RISC-V VP provides a full virtual prototype of common RISC-V hardware platforms, e.g. the SiFive HiFive1[4] or the SiFive HiFive Unleashed[5]. As such, RISC-V VP focuses on simulation accuracy and therefore also uses the SystemC

[3]Refer to Subsect. IV-A for more information on the simulator-specific implementation of the interface model for Spike and RISC-V VP.

[4]https://www.sifive.com/boards/hifive1

[5]https://www.sifive.com/boards/hifive-unleashed

```
// ...

static inline uint32_t
read_register(void *c, unsigned idx)
{
  return ((struct rv32::ISS*)c)->regs[idx];
}

static inline void
write_register(void *c, unsigned idx, uint32_t v)
{
  ((struct rv32::ISS*)c)->regs[idx] = v;
}

static inline uint8_t
load_byte(void *c, uint32_t addr)
{
  auto mem = ((struct rv32::ISS*)c)->mem;
  return mem->load_byte(addr);
}

// ...
```

Fig. 8. Excerpt of the interface model implementation for RISC-V VP.

hardware modeling language. The entire execution of RISC-V machine code is performed within a SystemC simulation, which eases reasoning about low-level details (e.g. timing). RISC-V VP is further described in a publication by Herdt *et al.* [12]. We choose Spike and RISC-V VP for our experiments, because they represent two ends of a spectrum (Spike focuses on simulation performance while RISC-V VP focuses on simulation accuracy) and their implementations therefore differ significantly. This demonstrates that our approach is applicable to a variety of existing simulators, from full VPs to performance-oriented simulators like Spike.

In order to employ our ISS generation approach for these simulators, we first had to manually implement an interface model for each simulator (see Subsect. III-C). As part of this implementation, we need to map the simulator-agnostic API for interacting with simulator components (e.g. the register file) to the internal simulator-specific API. An excerpt of the interface model implementation for RISC-V VP is shown in Fig. 8. As illustrated in this figure, the interface model casts a provided void pointer to a simulator-specific type for representing a RISC-V processor (`struct rv32::ISS`) and afterward calls methods of this type to implement the semantics of the interface model. The implementation presented in Fig. 8 is specific to RISC-V but the Spike interface model has a similar complexity. The complete implementation of both interface models is available as part of the publication artifacts. Apart from the interface model, we also had to connect the generated functions which implement the semantics of RISC-V instructions (see Fig. 7) with the existing fetch-decode-execute cycle implementation of Spike and RISC-V VP. Spike already generates functions for the implementation of RISC-V instruction using several scripts, which we have adjusted accordingly. RISC-V VP uses a switch/case statement to execute decoded instructions, which—similar to Spike—we now generate using a script. In total, we modified roughly 150 lines in RISC-V VP to implement the interface model and the build system changes. In Spike, we modified 200 lines for the same purpose.[6] The integration process took a programmer with domain knowledge less than a day. As such, the experiments demonstrate that minimal effort is required to apply our approach to different RISC-V simulators, thereby illustrating its generalizability.

B. Conformance

With the modifications outlined in the previous section, our enhanced versions of Spike and RISC-V VP use an ISS that has been automatically generated from the formal LIBRISCV ISA model, instead of a manually written one. Naturally, it is possible that the ISA model does not correctly capture the RISC-V instruction semantics or that our code generation tool or interface model implementations contain bugs. Therefore, it is paramount to ensure that the generated ISS still conforms to the RISC-V specification. In order to test conformance to the specification, we utilize the official RISC-V ISA tests for the 32-bit base instruction set.[7] These tests include several test programs (one per instruction) which validate the behavior of RISC-V instruction implementations using manually written test cases. Both our modified version of Spike and RISC-V VP pass the RISC-V ISA tests for `rv32i`. This indicates that our enhanced version of the LIBRISCV ISA model still conforms to the RISC-V ISA tests and that our code generator and interface model do not introduce any severe bugs. In future work, we would like to expand our conformance tests by showing equivalence between the generated ISS and the manually written one.

C. Performance

Since our approach replaces a manually written ISS with an automatically generated one, there is the possibility that the code generation tool does not account for optimizations included in the manually written code. Simulation speed is of importance for RISC-V simulators in order to be able to execute and test complex RISC-V software in a reasonable time span. In order to evaluate the impact of our approach on simulation speed, we use our modified version of RISC-V VP (see Subsect. IV-A) and perform a simulation speed comparison with the original unmodified version of this simulator (referred to as the baseline version in the following). In prior work, performance benchmarks for RISC-V VP have been conducted using the Embench benchmark suite [14]; therefore we also use Embench for our experiments. Embench is an open source benchmark suite which is specifically tailored to the embedded domain, it consists of several benchmark programs which perform computation intensive tasks (such as checksum calculation) [18]. We conduct our experiments with Embench 1.0 on a Linux system with an Intel i7-8565U processor.

Since benchmark results for a simulator can differ depending on the workload of the host system, we executed each benchmark application 25 times with both variants of RISC-V VP. Benchmark results are presented as a grouped bar chart in Fig. 9. The absolute execution time in seconds is given on the y-axis of Fig. 9 while the x-axis lists the benchmark programs

[6]Naturally, automatically generated lines are not included in this metric, since they do not correspond to any manual integration effort.

[7]https://github.com/riscv/riscv-tests

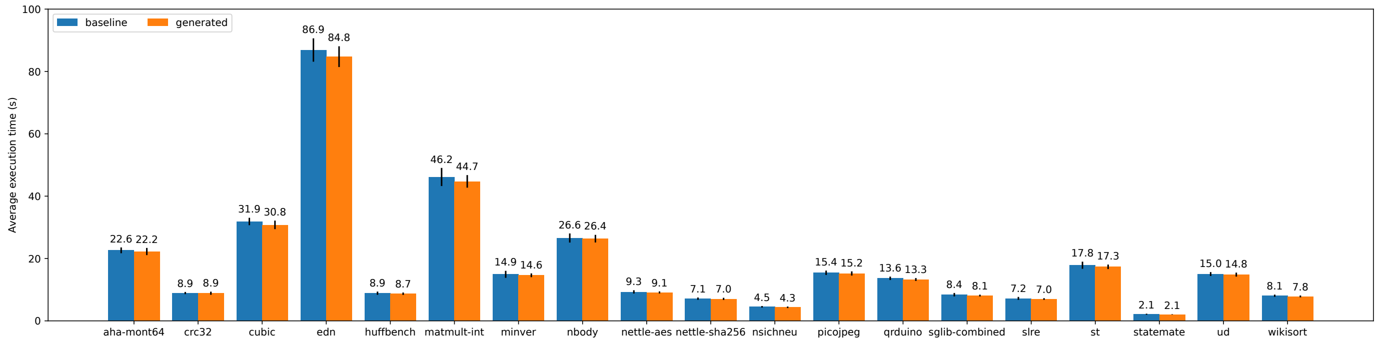

Fig. 9. Execution time benchmarks performed for an unmodified version of RISC-V VP and a version generated using our approach. Each benchmark application has been executed 25 times. The bars represent the arithmetic mean over all executions (*lower is better*); error bars indicate the standard derivation.

of the Embench suite. For each benchmark program, two bar charts are presented: the left bar chart (blue) represents the results for the baseline version, the right bar chart (orange) represents our modified version of RISC-V VP (i.e. uses an ISS generated using our approach). Both bar charts specify the arithmetic mean for the execution time of a given benchmark application over 25 executions. The error bars in Fig. 9 specify the standard derivation.

In total, 19 benchmark applications have been tested with both variants of RISC-V VP. Comparing the results for each benchmark application, execution time for the generated RISC-V VP variant is either sightly lower or the same as the execution time of the baseline version. This indicates that an ISS generated using our approach does not have worse simulation performance than a manually written one. Since the generated instruction semantics are the same for both Spike and RISC-V VP and only the interface model differs, we do not provide a comparison of Spike variants in this publication.

V. Related Work

Due to their numerous advantages, formal ISA models have been subject of intense research for a while now. ARM technologies introduced a custom domain-specific language (DSL) with a formal semantics, to describe the ISA of their at that time new ARM-v8 architecture [2]. The formal language allowed deriving test suites and Verilog code, for example. Following on from this, the Sail language was developed, a DSL designed to describe ISA semantics [19]. From the Sail model of an ISA, formal descriptions in different programming and theorem proving languages can be generated (such as C, OCaml, Coq, Isabelle, or HOL4). Sail has been used in prior work to model the RISC-V, ARM-v8 and MIPS ISAs, amongst others [3]. However, such versatility comes at a price: the generated models are not as concise or convenient to work with as native ones. Moreover, Sail focuses on completeness and therefore goes beyond the description of instruction semantics and also includes formalization of additional ISA details such as address translation algorithms or instruction decoding. This contributes to the complexity of Sail and makes it difficult to integrate it into an existing RISC-V simulator. Therefore, Sail instead generates a new standalone ISA simulator [3, Sect. 5].

Specifically for RISC-V, a variety of formal ISA models exist [20]. The definitive formal model—approved by the RISC-V board—is provided in Sail [3], but models in the Coq theorem prover or the Haskell programming language exist as well. The Coq model uses the embedded Kami DSL [21] and allows reasoning about correctness—in particular, about correctness of microarchitecture implementations of the ISA, or about software running on the ISA—but as Coq is an interactive theorem prover rather than a programming platform, it is less convenient to work with in aspects other than proof (e.g. code generation). Apart from the LibRISCV model used in this publication, additional RISC-V models in Haskell are GRIFT [4] and Forvis [5]. All of these have different design goals. While Forvis was meant as an executable formal model implemented in a deliberately restricted subset of Haskell, GRIFT attempted to capture the ISA as precise as possible in Haskell's type system, and LibRISCV was designed with a focus on instruction semantics to ease building custom ISA interpreters. From these, the design goals of LibRISCV best aligned with ours as it enabled us to focus on generating the code implementing instruction semantics while retaining other parts, thereby easing the integration with existing simulators.

Related work in the electronic design automation domain leverages architecture description languages (ADLs) for processor descriptions [22, 23, 24]. Compared to formal ISA models, these ADLs focus more on microarchitectural details (such as pipelining or caching). For this reason, it is challenging to integrate them with existing simulators and vendor-supplied components. Therefore, these languages are primarily used to generate new simulators instead of aiming for an integration with existing ones. To the best of our knowledge, the generation approach presented here is the first which is easily applicable to a variety of existing simulators, from full VPs like RISC-V VP to performance-focused simulators like Spike.

VI. Discussion & Future Work

In this publication, we focused on a minimally invasive integration of formal ISA models with existing RISC-V simulators. Especially in the VP domain, formal models have not yet been used to their full potential. In order to ease usage of formal ISA models for VP generation, we focused on minimizing the integration effort. Therefore, we only employed a formal model for the 32-bit RISC-V base instruction set [10,

Sect. 2]. In future work, it would be possible to focus more on modeling aspects and expand the underlying LIBRISCV ISA model to cover additional RISC-V extensions and RISC-V variants (e.g. 64-bit RISC-V). In this context, it would be especially interesting to also support parts of the RISC-V privileged architecture specification [10], instead of focusing on the user-level ISA [9]. Such modeling aspects are also discussed in the original LIBRISCV paper [6], extending the formal model to cover more parts of the ISA (e.g. decoding) would allow our code generation approach to be more comprehensive. However, in this regard, there is a trade-off between comprehensiveness of the formal model and the effort required to integrate it with existing simulators. The premise of our work is that by focusing on code implementing the actual instruction semantics—where in accordance with prior work we assume most bugs to occur [8]—we can more easily integrate the formal model with existing simulators. If we were to capture additional parts of an ISA (e.g. memory behavior) in the formal model as well, it would complicate the integration with existing simulators. In terms of comprehensiveness, it is possible to cover the semantics of additional RISC-V extensions using our approach in future work [6, Sect. 7].

VII. Conclusion

In this paper, we have presented a novel approach for generating the ISS of RISC-V simulators from a formal ISA model. Contrary to prior work, our approach is designed to be as minimally invasive as possible through a simulator-agnostic interface model (Subsect. III-C), a self-contained formal ISA model (Subsect. III-B), and a code generator for this model (Subsect. III-D). By focusing exclusively on the code implementing the actual instruction semantics, we ease the integration with existing vendor-supplied components and increases the confidence in the correctness of the utilized ISS. Conducted experiments confirm that our approach is applicable to different RISC-V simulators (Spike and RISC-V VP) with minimal effort (Subsect. IV-A). Furthermore, we were able to show that an ISS—generated using our approach—still passes the RISC-V ISA tests (Subsect. IV-B) and offers similar simulation speed performance as a manually written one (Subsect. IV-C). In future work, we want to investigate correctness proofs (for both the formal model and the generated ISS), expand our ISA model to support additional RISC-V extensions, and consider its application to VP-based software analysis tasks. To stimulate further research in this direction, we have released our code generation tool[8] as well as our modified versions of Spike[9] and RISC-V VP[10] as open source software.

References

[1] T. D. Schutter, *Better Software. Faster!: Best Practices in Virtual Prototyping*. Synopsys Press, Mar. 2014, ISBN: 978-1-61-730013-4.

[2] A. Reid, "Trustworthy specifications of ARM® v8-A and v8-M system level architecture," in *2016 Formal Methods in Computer-Aided Design (FMCAD)*, Oct. 2016, pp. 161–168. DOI: 10.1109/FMCAD.2016.7886675.

[8]https://github.com/agra-uni-bremen/formal-iss

[9]https://github.com/agra-uni-bremen/spike-libriscv

[10]https://github.com/agra-uni-bremen/libriscv-vp

[3] A. Armstrong, T. Bauereiss, B. Campbell, *et al.*, "ISA semantics for ARMv8-a, RISC-V, and CHERI-MIPS," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019. DOI: 10.1145/3290384.

[4] B. Selfridge, "GRIFT: A richly-typed, deeply-embedded RISC-V semantics written in Haskell," in *SpISA 2019: Workshop on Instruction Set Architecture Specification*, Portland, Oregon, Sep. 2019. [Online]. Available: https://www.cl.cam.ac.uk/~jrh13/spisa19/paper_10.pdf.

[5] Bluespec, Inc., *Forvis: A formal RISC-V ISA specification*, GitHub. [Online]. Available: https://github.com/rsnikhil/Forvis_RISCV-ISA-Spec.

[6] S. Tempel, T. Brandt, and C. Lüth, "Versatile and flexible modelling of the RISC-V instruction set architecture," in *Trends in Functional Programming*, S. Chang, Ed., Boston, MA, USA: Springer International Publishing, 2023, ISBN: 978-3-031-21314-4.

[7] System C Standardization Working Group, "IEEE standard for standard SystemC language reference manual," Tech. Rep., 2012, pp. 1–638. DOI: 10.1109/IEEESTD.2012.6134619.

[8] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Verifying instruction set simulators using coverage-guided fuzzing," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019, pp. 360–365. DOI: 10.23919/DATE.2019.8714912.

[9] RISC-V Foundation, *The RISC-V instruction set manual, volume I: User-level ISA*, ed. by A. Waterman and K. Asanović, Document Version 20191213, Dec. 2019.

[10] RISC-V Foundation, *The RISC-V instruction set manual, volume II: Privileged architecture*, ed. by A. Waterman and K. Asanović, Document Version 20190608-Priv-MSU-Ratified, Jun. 2019.

[11] University of California, *Spike, a RISC-V ISA simulator*, GitHub. [Online]. Available: https://github.com/riscv/riscv-isa-sim.

[12] V. Herdt, D. Große, P. Pieper, and R. Drechsler, "RISC-V based virtual prototype: An extensible and configurable platform for the system-level," *Journal of Systems Architecture*, vol. 109, p. 13, 2020, ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2020.101756.

[13] O. Kiselyov and H. Ishii, "Freer monads, more extensible effects," *SIGPLAN Notices*, vol. 50, no. 12, pp. 94–105, Aug. 2015, ISSN: 0362-1340. DOI: 10.1145/2887747.2804319.

[14] V. Herdt, D. Große, and R. Drechsler, "Fast and accurate performance evaluation for RISC-V using virtual prototypes," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2020, pp. 618–621. DOI: 10.23919/DATE48585.2020.9116522.

[15] J. Hughes, "The design of a pretty-printing library," in *Advanced Functional Programming*, J. Jeuring and E. Meijer, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 53–96, ISBN: 978-3-540-49270-2. DOI: 10.1007/3-540-59451-5_3.

[16] P. Wadler, "A prettier printer," in *The Fun of Programming*. Palgrave, Mar. 2003, ISBN: 0-3339-9285-7.

[17] L. Hermerschmidt, S. Kugelmann, and B. Rumpe, "Towards more security in data exchange: Defining unparsers with context-sensitive encoders for context-free grammars," in *2015 IEEE Security and Privacy Workshops*, 2015, pp. 134–141. DOI: 10.1109/SPW.2015.29.

[18] Free and Open Source Silicon Foundation, *Embench: A modern embedded benchmark suite*. [Online]. Available: https://www.embench.org/ (visited on 01/24/2023).

[19] A. Armstrong, T. Bauereiss, B. Campbell, *et al.*, "The state of sail," in *SpISA 2019: Workshop on Instruction Set Architecture Specification*, M. Fernandez, Ed., 2019. [Online]. Available: https://www.cl.cam.ac.uk/~jrh13/spisa19/paper_04.pdf.

[20] RISC-V Foundation, *ISA Formal Spec Public Review*, GitHub, Accessed 2023-03-23, 2019. [Online]. Available: https://github.com/riscvarchive/ISA_Formal_Spec_Public_Review.

[21] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, and Arvind, "Kami: A platform for high-level parametric hardware specification and its modular verification," *Proc. ACM Program. Lang.*, vol. 1, no. ICFP, Aug. 2017. DOI: 10.1145/3110268.

[22] S. Rigo, G. Araujo, M. Bartholomeu, and R. Azevedo, "ArchC: A SystemC-based architecture description language," in *16th Symposium on Computer Architecture and High Performance Computing*, 2004, pp. 66–73. DOI: 10.1109/SBAC-PAD.2004.8.

[23] A. Fauth, J. Van Praet, and M. Freericks, "Describing instruction set processors using nML," in *Proceedings the European Design and Test Conference. ED&TC 1995*, 1995, pp. 503–507. DOI: 10.1109/EDTC.1995.470354.

[24] V. Zivojnovic, S. Pees, and H. Meyr, "LISA-machine description language and generic machine model for HW/SW co-design," in *VLSI Signal Processing, IX*, 1996, pp. 127–136. DOI: 10.1109/VLSISP.1996.558311.