Instruction-Level Abstraction (ILA): A Uniform Specification for System-on-Chip (SoC) Verification

10

BO-YUAN HUANG and HONGCE ZHANG, Princeton University, USA
PRAMOD SUBRAMANYAN, Indian Institute of Technology Kanpur, India
YAKIR VIZEL, Technion Israel Institute of Technology, Israel
AARTI GUPTA and SHARAD MALIK, Princeton University, USA

Modern Systems-on-Chip (SoC) designs are increasingly heterogeneous and contain specialized semi-programmable accelerators in addition to programmable processors. In contrast to the pre-accelerator era, when the ISA played an important role in verification by enabling a clean separation of concerns between software and hardware, verification of these "accelerator-rich" SoCs presents new challenges. From the perspective of hardware designers, there is a lack of a common framework for formal functional specification of accelerator behavior. From the perspective of software developers, there exists no unified framework for reasoning about software/hardware interactions of programs that interact with accelerators.

This article addresses these challenges by providing a formal specification and high-level abstraction for accelerator functional behavior. It formalizes the concept of an Instruction Level Abstraction (ILA), developed informally in our previous work, and shows its application in modeling and verification of accelerators. This formal ILA extends the familiar notion of instructions to accelerators and provides a uniform, modular, and hierarchical abstraction for modeling software-visible behavior of both accelerators and programmable processors. We demonstrate the applicability of the ILA through several case studies of accelerators (for image processing, machine learning, and cryptography), and a general-purpose processor (RISC-V). We show how the ILA model facilitates equivalence checking between two ILAs, and between an ILA and its hardware finite-state machine (FSM) implementation. Further, this equivalence checking supports accelerator upgrades using the notion of ILA compatibility, similar to processor upgrades using ISA compatibility.

CCS Concepts: • **Computer systems organization → Architectures**; • **Hardware → Application-specific VLSI designs**; **Functional verification**; *Electronic design automation*;

Additional Key Words and Phrases: System on chip, hardware specification, application-specific accelerator, architecture, instruction-level abstraction, formal verification, equivalence checking

This work was supported by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

Authors' addresses: B.-Y. Huang and H. Zhang, Princeton University, Princeton, 1 Nassau Hall, Princeton, New Jersey, 08544, USA; emails: {byhuang, hongcez}@princeton.edu; P. Subramanyan, Indian Institute of Technology Kanpur, Nankari, Kalyanpur, Kanpur, Uttar Pradesh 208016, India; email: spramod@cse.iitk.ac.in; Y. Vizel, Technion Israel Institute of Technology, Haifa, Viazman 87, Technion City, Haifa, Haifa District 3200003, Israel; email: yvizel@cs.technion.ac.il; A. Gupta and Sharad Malik, Princeton University, Princeton, 1 Nassau Hall, Princeton, New Jersey, 08544, USA; emails: aartig@cs.princeton.edu, sharad@princeton.edu.

1084-4309/2018/12-ART10 $15.00
https://doi.org/10.1145/3282444

ACM Reference format:
Bo-Yuan Huang, Hongce Zhang, Pramod Subramanyan, Yakir Vizel, Aarti Gupta, and Sharad Malik. 2018. Instruction-Level Abstraction (ILA): A Uniform Specification for System-on-Chip (SoC) Verification. *ACM Trans. Des. Autom. Electron. Syst.* 24, 1, Article 10 (December 2018), 24 pages.
https://doi.org/10.1145/3282444

1 INTRODUCTION

Today's computing platforms are increasingly heterogeneous, a trend that is expected to continue into the foreseeable future as per the International Technology Roadmap for Semiconductors [22]. In addition to programmable processors—both general purpose and domain specific such as Graphics Processing Units (GPUs)—today's platforms contain dedicated accelerators in order to meet the power-performance requirements posed by emerging applications. These accelerators may be tightly coupled, i.e., part of the processor pipeline, or loosely coupled, interacting with the processor through shared memory [27]. The latter form is dominant and the focus of this article. Apple's `A` series of processors illustrate this growth in accelerators; the `A8` processor has 30 accelerators [61] while the `A10` has 40.

Accelerator-rich platforms pose two distinct verification challenges. The first challenge is constructing meaningful specifications for accelerators that describe behavior exposed at the hardware/software interface. Such specifications are important not just for correct design/verification of hardware, but are also required to drive software and firmware development, both of which must often be done before the hardware is "taped-out." Specifications are also required to reason about portability between different generations of accelerator architectures. They can mitigate the software incompatibility risk involved in the implementation of microarchitectural enhancements. Further, it is important to note that specifications must necessarily be an *abstraction* of hardware functionality. Detailed models, e.g., Register-Transfer Level (RTL) descriptions, expose cycle-level behavior that is not part of the hardware/software interface and thus are not suitable as specifications. In addition, RTL descriptions are also undesirable as specifications because the detailed nature of these models means they are not amenable to scalable formal analysis.

The second challenge is reasoning about hardware-software interactions from the perspective of software. For software that runs exclusively on a programmable processor, its execution semantics are defined by the processor's instruction set architecture (ISA) specification. Thus, the ISA serves as a suitable abstraction of the underlying processor hardware for software verification. However, similar abstractions of hardware for reasoning about software interacting with accelerators are lacking. Software typically accesses accelerators through memory-mapped input-output (MMIO) instructions that map memory and registers inside the accelerators to specific addresses. From the perspective of the ISA, accelerator interactions appear to be just loads/stores of these addresses. However, these loads/stores trigger specific functionality implemented by the accelerator logic not modeled by the processor's load/store instruction semantics. Further, the accelerator may access some memory shared with the processor, and potentially interrupt the processor on completion of specific functions. These aspects make the ISA incomplete for modeling accelerator interactions. As a result, reasoning about software that interacts with accelerators, an increasingly important task in today's SoCs, is usually done through ad-hoc abstractions/modeling techniques that compose ISA-level models with FSM models of accelerators (e.g., in Verilog/VHDL). This results in an *abstraction gap* between the ISA and the low-level hardware FSM, making software/hardware co-verification with accelerators very challenging.

In this work, we propose a uniform and formal abstraction for processors and accelerators that captures their software-visible functionality. This abstraction is called an Instruction-Level Abstraction (ILA) and is based on the familiar notion of computation triggered by "instructions." For

a processor, the ILA is based on the ISA. For an accelerator, the insight is that commands at its interface are akin to instructions in a processor. Thus, just as the ISA models processor behavior through specifying state changes resulting from each instruction, the ILA models accelerator behavior by specifying state changes resulting from each of its "instructions," i.e., its commands. Further, as with ISAs, this modeling can distinguish the state that is persistent between instructions (architectural state), from implementation state (micro-architectural state). Top-down this modeling provides a specification for functional verification of hardware, and bottom-up it provides an abstraction for software/hardware co-verification.

The ILA, like an ISA, has the following useful attributes. It provides

(i) a modular functional specification as a set of instructions;
(ii) a meaningful state abstraction in terms of architectural state, i.e., a state that is persistent between instructions, while abstracting away an implementation state; and
(iii) a specification for each instruction in the form of state update functions for architectural state.

In modeling designs with complex instructions, it is sometimes easier to describe the architectural state update function as a sequence of steps, i.e., an algorithm. These steps may be required of all implementations, in which case they are considered part of the specification, or may only indicate a possible implementation. The ILA model allows this sequencing to be expressed through hierarchy in instructions, where an instruction can itself be modeled as a *sequence* of two different kinds of *child* instructions.

This work builds on [62, 64] which introduced an informal notion of the Instruction-Level Abstraction (ILA). That work viewed an ILA as a finite state system and focused on synthesizing ILAs using program synthesis techniques [3, 41]. The focus of this work is on formalizing the ILA as an instruction-centric operational model, well-suited as an interface between sequential software and the underlying hardware. To treat processors and accelerators uniformly, the ILA model explicitly includes functions that perform the fetch-decode-execute of instructions. This is especially useful in reasoning about a system of interacting ILA models, one ILA per processing unit, where the decode function (dependent on the fetch function) captures the condition whether an instruction is enabled to execute or not, and the execute part actually performs the update of the software-visible state. Note that the earlier finite state model could capture only the execute part. Furthermore, we have introduced hierarchy into the ILA model, via the notions of child (sub- and micro-) instructions, where an instruction at a higher level can be represented as a sequence of child instructions at a lower level. Thus, the granularity of ILA instructions can vary, ranging from processor instructions to software functions, but the focus is on modeling software-visible states and their updates. Finally, this work showcases the usefulness of the formal ILA model and its applications in verification through a set of rich case studies comprising accelerators from diverse application domains (advanced encryption, image processing, machine learning) and a processor (RISC-V Rocket Core). The earlier papers had focused only on an accelerator for encryption.

Note that while we describe the verification applications using ILAs in detail, we do not claim the verification techniques to be our central contribution—indeed, we have used standard verification techniques and commercial off-the-shelf verification tools in our case studies. The point to note is that the ILA model enables application of these techniques in a compositional manner, where the set of instructions naturally provides an instruction-based decomposition into simpler verification tasks.

Contributions of this Article

Overall this article makes the following contributions:

Table 1. Comparison of Hardware and System-Level Modeling Frameworks

Modeling Language/Framework	Level of Abstraction					Formal Semantics
	Alg.	Func.	CA	RTL	GL	
Verilog/VHDL			✓	✓	✓	Yes
Design Specific Models in C/C++ and so forth (e.g., [5, 16, 60])	✓	✓	✓			No
Chisel, PyMTL [10, 45]		✓	✓	✓		No
System-Level Modeling Frameworks [7, 9, 14, 34, 36, 51, 53]	✓	✓				Yes
ILA (this work)		✓	✓	✓		Yes

Column labels are Algorithmic (Alg.), Functional (Func.), Cycle Accurate (CA), Register Transfer Level (RTL), and Gate Level (GL).

—It provides a formal model for the ILA (Section 3). This addresses critical modeling issues in both processors and accelerators including gaps in previous ISA formal models. Top-down this model provides a formal specification for use in hardware verification, and bottom-up an abstraction for use in software/hardware co-verification that is *uniform* across accelerators and processors.
—It supports hierarchy (Section 3.2) in modeling instructions which is missing from the earlier formal ISA models [59]. In particular, it makes the important distinction between hierarchy in the specification and hierarchy in the implementation.
—It demonstrates the applicability of the ILA model through several case studies on accelerators (AES, RBM, Gaussian Blur) and the RISC-V Rocket processor (Section 4).
—It demonstrates the value in verification across models—between two ILAs, and between ILA and FSM models—through successful case studies (Section 5), including finding a bug in the RISC-V Rocket processor core. Verifying FSM implementations against ILA specifications provides the basis for ILA-compatible accelerator replacement.

2 MOTIVATION AND BACKGROUND

2.1 System-Level/Hardware Modeling Frameworks

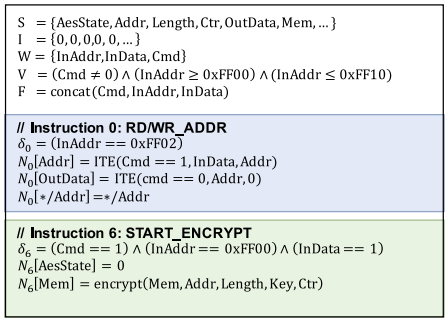
Table 1 categorizes notable system-level and hardware modeling frameworks in terms of their level of abstraction and the suitability of their models for formal analysis. The "traditional" approach to processor-based platform design uses (i) functional models of processor ISAs (typically developed in C/C++) to define architectural behavior, and (ii) cycle-accurate simulators (e.g., ESEC and gem5 [5, 16], also in C/C++) to explore the microarchitectural design space. Finally, the implementation typically uses RTL descriptions in Verilog/VHDL. This approach corresponds to the first two rows in Table 1.

Recent years have seen increased interest in system-level modeling that raises the level of abstraction for design and verification. SystemC in particular, has seen significant adoption in system/transaction-level modeling. However, RTL designs in Verilog, corresponding to SystemC transaction-level models, are usually separately constructed by hand. Ensuring that the system-level models in SystemC and the corresponding RTL are in agreement is a challenging problem. Chisel [10] and PyMTL [45] propose to address this challenge by providing unified domain-specific embedded languages in Scala and Python, respectively, for constructing functional, cycle-accurate, and RTL models. While this can mitigate some challenges in testing equivalence among these various models, bugs still slip through the cracks. In particular, these languages do not have formal precisely defined semantics which limits automated reasoning. This makes it hard to provide guarantees of equivalence between models at different levels of abstraction.

Models with formally defined operational semantics are amenable to formal analyses such as equivalence and property checking. Examples include StateCharts, SystemC, Esterel, Transaction

	Instruction	Description
0	RD/WR_ADDR	Get/set address of data to encrypt/decrypt
1	RD/WR_LENGTH	Get/set length of data to encrypt/decrypt
2	RD/WR_KEY0	Get/set key register 0
3	RD/WR_KEY1	Get/set key register 1
4	RD/WR_SELECT	Get/set key selector
5	RD/WR_CTR	Get/set counter for CTR mode
6	START_ENCRYPT	Start the encryption state machine
7	GET_STATUS	Poll for completion

(a) AES ILA instructions



(b) AES ILA definitions (without child-ILA definition)

Fig. 1. ILA for an AES accelerator.

Level Modeling (TLM), and others [1, 4, 9, 14, 31, 34–36]. A notable effort in this category is BlueSpec, a high-level specification and design language that describes hardware as sets of state change rules (guarded atomic actions) which execute atomically [7, 51]. The BlueSpec compiler synthesizes the circuits and exploits parallelism with a scheduler to choose the interleaving of rules automatically [28, 36]. BlueSpec has well-defined operational semantics and supports modular verification using SMT solvers and interactive theorem provers [29, 67].[1]

2.2 Desired Hardware Abstraction Characteristics

A given hardware design can be abstracted in many different ways. In this article, we argue for abstractions of hardware that satisfy two important properties:

- —The abstraction cleanly separates hardware and software verification concerns. This requires that the abstraction precisely codify the hardware/software interface so that software and hardware can be separately developed and verified to be conformant with the interface.
- —The abstraction treats programmable processors and accelerators uniformly. Software verification in future architectures will need to reason about accelerator interactions in addition to processor ISAs, while hardware verification will need to reason about the software interface presented by these accelerators. A uniform abstraction for these architectures is required in order to provide a common accelerator-agnostic framework for this verification.

None of the frameworks in Table 1 satisfy these properties. In this article, we take a step toward addressing this gap by introducing a uniform and hierarchical ILA: an abstraction of hardware that precisely delineates the hardware/software interface. Our notion of the ILA treats programmable processors and semi-programmable accelerators uniformly, including hierarchical modeling of microarchitecture for accelerators, similar to processors. Past work has shown how abstractions at the instruction level can be successfully used for software/hardware co-verification [63].

3 FORMAL MODELING

In this section, we formally define the ILA model and its execution semantics. A motivating example used through this section is shown in Figure 1, for an accelerator (from opencores.org) [38] that implements the Advanced Encryption Standard (AES). The derived ILA instructions are shown in Figure 1(a): six instructions read/write configuration registers, one starts encryption, and one checks the completion status. As discussed earlier, these "instructions" correspond to commands presented at the accelerator interface by the processor.

[1]See Section 7 for a detailed comparison of the ILA with BlueSpec and other related efforts.

3.1 ILA

This section defines the ILA, without considering hierarchy. An ILA A is a tuple: $\langle S, I, W, V, F, D, N\rangle$, where

S	is a vector of state variables,
I	is a vector of initial values of the state variables,
W	is a vector of input variables,
V	$: (S \times W) \rightarrow \mathbb{B}$ is the valid function, $\mathbb{B} = \{0, 1\}$,
F	$: (S \times W) \rightarrow bvec_w$ is the fetch function,
D	$= \{\delta_i : bvec_w \rightarrow \mathbb{B}\}$ is the set of decode functions,
N	$= \{N_i : (S \times W) \rightarrow S\}$ are the next state functions.

The state variables in S can be of type Boolean, bitvector, or array (representing memory). For processors, S includes architectural registers, flag bits, data and program memory. For accelerators, S includes memory-mapped registers, internal buffers, output ports to on-chip interconnect, data memory, and so forth. We refer to these state variables as "architectural state" because like an ISA's architectural state, they are persistent across instructions. In the ILA for the AES example, as shown in Figure 1(b), the architectural state variable `Addr` denotes the address of data to encrypt, and `Length` is the data length. I denotes the set of initial values of the corresponding architectural states in S. The vector of input variables W includes input ports of the hardware module, such as processor interrupt signals and accelerator command inputs. For example, input `InData` in the AES ILA is the data from the memory system for memory-mapped accesses.

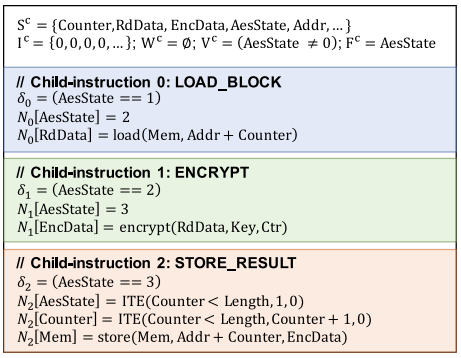
Instructions in an ILA follow the fetch/decode/execute paradigm, similar to a processor ISA. To model event-driven accelerators, we include a valid function $V : (S \times W) \rightarrow \mathbb{B}$ that indicates if an instruction is triggered based on state and input values. For example, the AES accelerator executes instructions only when `InAddr` is within a specified range, i.e., $V(S, W) \triangleq$ (`InAddr` $\geq$ `0xFF00`) $\wedge$ (`InAddr` $\leq$ `0xFF10`).

The *opcode* of the instruction is modeled as a bitvector of width w (denoted $bvec_w$). If the instruction is triggered (i.e., if V is true), then the fetch function $F : (S \times W) \rightarrow bvec_w$ indicates how it is extracted from the state and inputs. For processors, the opcode is fetched from the program memory location pointed to by the program counter, i.e., $F(S, W) \triangleq read(\texttt{IMEM}, \texttt{PC})$. If interrupt modeling is desired, F concatenates this with the interrupt signals (inputs). Similarly, accelerators extract the opcode for decoding instructions. The opcode for the AES example is the concatenation of the memory-mapped input signals, as shown in Figure 1(b).

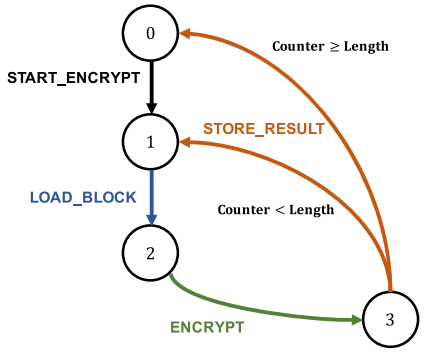
Each instruction (indexed by i) is associated with a decode function $\delta_i : bvec_w \rightarrow \mathbb{B}$, indicating whether it is *issued*. For example, as shown in Figure 1(b), the instruction `START_ENCRYPT` is issued only when it receives a "store value 1 to address 0xFF00" command at the interface. The set of all decode functions is $D = \{\delta_i | 0 \leq i < k\}$; k is the number of instructions. In an ILA, only one instruction can be issued at a time, i.e., D is one-hot encoded. Non-determinism should be modeled with explicit *choice variables* (inputs) provided by the external environment. Note the valid function V returns true if and only if one decode function returns true.

Finally, each instruction is associated with a next state function $N_i : (S \times W) \rightarrow S$, which represents the state update when the instruction is executed. The set of all next state functions in the ILA is $N = \{N_i | 0 \leq i < k\}$.

To summarize, Figure 1(a) shows the description of all eight instructions of the AES accelerator. Figure 1(b) shows the ILA definitions for S, I, W, V, F, and the decode (δ_i) and state update functions (N_i) for two of the instructions.



(a) AES child-ILA definitions ($m_i = true$, i.e., micro-ILA)



(b) The loop structure in the AES child-program. (Each node shows the value of AesState.)

Fig. 2. Child-ILA for an AES accelerator.

3.2 Hierarchical ILAs

In modeling designs with complex instructions, it is often easier to describe the architectural state update function as a sequence of steps, i.e., an algorithm. These steps may be required of all implementations, in which case they are considered part of the specification, or may only indicate one possible implementation. For example, the Intel x86 architecture [39] specifies the string copy instruction `REP MOVS` as a sequence where the `MOVS` instruction is repeated until register `ECX` (count) is decremented to 0. Note that the state update performed by this instruction at the architectural level is not atomic, and this fact needs to be captured in the architecture model. Similarly, in the AES accelerator in Figure 1, the `START_ENCRYPT` instruction involves reading data, encrypting it, and writing the result. The encryption itself is also a complex operation that needs to be described as a sequence of steps.

Child ILAs. To support modeling such complex instructions, we extend the ILA definition from Section 3.1 to support hierarchy. A hierarchical ILA may contain *child-ILAs*, each of which describes the *sequence of steps* in the complex instruction. Instructions in child-ILAs, referred to as child-instructions, also follow the fetch/decode/execute paradigm. These may, in turn, contain other child-ILAs, and we refer to an ILA containing a child as a *parent-ILA*. In the AES example, `START_ENCRYPT` is modeled by a child-ILA with child-instructions for message loading, encryption, and storing results, as shown in Figure 2(a). The child-instruction `ENCRYPT` is further modeled by a child-ILA for the actual encryption algorithm. As we will show later in Section 4.1, two different child-ILAs can be used to describe two different AES implementations.

A child-ILA is defined similar to an ILA. Its state variables are denoted as S^c, some of which may be shared with the parent-ILA. W^c is the set of its input variables, and is a subset of inputs of the parent-ILA. The initial values are I^c, and the initial values of the shared state variables are the same as that of the parent-ILA. For the AES example in Figure 2(a), the child-ILA has no inputs and contains three additional state variables (*Counter*, *RdData*, and *EncData*). The other components of the child-ILA: V^c, F^c, D^c, and N^c are similarly defined in terms of S^c and W^c. The state variables shared between the child-ILA and the parent-ILA are *in lock step*, since they denote shared state, i.e., updates to the shared states are visible to the parent-ILA when a child-instruction is executed, and vice versa. For the AES example in Figure 1, the instruction `START_ENCRYPT` of its parent-ILA updates shared state `AesState` to 1 and keeps `Mem` unchanged; this starts the child-ILA with the child-instruction `LOAD_BLOCK`.

Informally, a child-ILA steps through a sequence of child-instructions, where the sequencing is implicitly determined by the state updates (using its state variables). That is, the child-instructions are sequentially composed. This can be viewed as a *child-program*. For the AES example, the child-program in Figure 2(a) models the `START_ENCRYPT` instruction, which comprises a loop and is controlled by the states `AesState` and `Counter`, as illustrated in Figure 2(b).

3.2.1 Micro-Instructions and Sub-Instructions. Child-ILAs can be used to model specifications or implemented algorithms. When modeling an implemented algorithm, their instructions serve the same role as **micro-instructions** for complex instructions in processors, which represent one possible implementation of that instruction. We distinguish this from instructions of child-ILAs when used for specification, i.e., when they specify behavior that must hold for *all* implementations. In the latter case, we call them **sub-instructions**. For example, in `REP MOVS`, the steps of the instruction are part of the specification, and thus, these sub-instructions are required of *all* implementations. Therefore, child-ILAs will be referred to as sub- or micro-ILAs depending on whether they have sub- or micro-instructions, respectively.

The distinction between micro-instructions (implemented algorithm) and sub-instructions (specifications) is important. For example, the ARM Cortex-M3 user guide [6] says that load-multiple (`LDM`) and store-multiple instructions (`STM`) access memory "in order of increasing register numbers." Another shared memory interacting processor would expect to see this order. Further, these instructions are interruptible, thus the intermediate values of the architectural states are visible to the interrupt handler. This order of accesses is therefore desired in the formal abstraction when verifying systems with multiple interacting hardware components. However, while the user guide only describes one particular implementation, the ARM architecture specification does not impose this ordering requirement. This is reflected in the previous work on the ARM ISA formal specification and verification [59]. While they state that "some load instructions may be split into multiple micro-ops" and account for it by updating the architectural state when each micro-op completes, when verifying this instruction they check the state only "when the last micro-op completes." We emphasize that it is important to treat these split accesses as *micro-instructions* (i.e., as implemented algorithms) and not *sub-instructions*.

Due to the differing roles of specifications and implementations in verification, we impose some restrictions on hierarchical ILAs. A sub-ILA may contain sub-ILAs or micro-ILAs. However, a micro-ILA can only contain micro-ILAs, as an implementation cannot contain a specification.

3.2.2 Definition of Hierarchical ILAs. A hierarchical ILA A is defined as $\langle S, I, W, V, F, D, N, C\rangle$. The new component $C = \{(A_1^c, m_1), \dots\}$ is a set of tuples consisting of child-ILAs and a Boolean flag that denotes whether the particular child-ILA is a micro-ILA (m_i = `true`) or a sub-ILA (m_i = `false`).

Figure 1(b) shows the ILA definitions for S, I, W, V, F, and the decode (δ_i) and state update functions (N_i) for two of the instructions in the AES example. Figure 2(a) shows the definitions for a child-ILA that models the `START_ENCRYPT` instruction.

3.3 ILA Execution Semantics

An ILA model is essentially a labeled state transition system that emphasizes modularity through a set of instructions. The semantics of execution of an ILA instruction is as follows:

$$\frac{V(S,W) \quad \delta_i(F(S,W)) \quad S' = N_i(S,W)}{S \overset{i}{\rightsquigarrow} S'} \tag{1}$$

Rule (1) says that an ILA can transition from state S to S' if the following conditions are satisfied:

—An instruction is triggered: $V(S, W)$ is $true$.
—The i-th instruction is issued: $\delta_i(F(S, W))$ is $true$.
—State update of the vector S' is according to $N_i(S, W)$.

Execution of a child-instruction in a child-ILA is similar:

$$\frac{V^c(S^c, W^c) \quad \delta_j^c(F^c(S^c, W^c)) \quad S'^c = N_j^c(S^c, W^c)}{S^c \overset{j}{\rightsquigarrow} S'^c} \tag{2}$$

State updates in instructions at the lowest level of an ILA hierarchy are considered to be *atomic*, i.e., indivisible. This enables reasoning about concurrency with multiple ILAs.

The focus of this article is on using an ILA to model the behavior of a single processor/accelerator core using instructions. This is useful for capturing a *sequential programming model* for the core's operation as it processes a *sequence* of instructions. Although the hardware may operate on instructions in parallel (similar to pipelined processors), the programming abstraction for software is that of a single sequential thread of control (similar to the ISA programming model). The value of the ILA is that this sequential programming model is now extended uniformly from processors to hardware accelerators. *We believe this abstraction from parallel hardware in accelerators to a single sequential programming model is a key enabler for system design and verification, and a central contribution of the ILA methodology.*

Further, once we have ILAs, each of which represents a single thread of control that updates shared architectural state, we can use them to model a system of concurrent cores with shared memory. Specifically, instructions are sequentially composed within an ILA, whereas concurrency and interleaving models are handled outside of ILAs. Analogous to ISAs for processors, we can use techniques for modeling multi-thread concurrency and memory consistency with multiple ILAs. This is discussed briefly in Section 6.1 later—case studies and applications with concurrent cores are outside the scope of this article.

4 CASE STUDIES: MODELING

In this section, we evaluate the ILA's modeling abilities using four case studies: application-specific accelerators for image processing, machine learning, and cryptography; and the Rocket processor core based on the RISC-V ISA. With designs from different application domains, the ILA is shown to be a uniform model usable across heterogeneous accelerators and processors. Verification for these case studies is described in the next section.

We create the ILA for each design based either on an informal English specification or a high-level reference model. These ILAs are synthesized using template-driven program synthesis [62], or in some cases manually written in Python using our ILA library API. [2] Table 2 provides information about each case study. Columns 2–5 give the reference model type, and sizes of the reference model and RTL, respectively. The RTL descriptions are either generated by high-level synthesis or taken from OpenCores.org. Columns 6 and 7 provide the number of instructions/child-instructions in the ILA, and ILA size (in lines of Python code). We now discuss salient aspects of each case study.

4.1 Application-Specific Accelerators

We consider two types of accelerators: (i) those using local memory for computation and direct memory access (DMA) to load/store data into their local memory buffers and (ii) those streaming input and output data. The commands at the interface relate to (i) the interface protocol and (ii) the computation tasks. In the AES example, the interface protocol refers to setting configurations

[2] All models and templates are available on GitHub: https://github.com/PrincetonUniversity/ILA-Modeling-Verification.

Table 2. ILA Modeling Case Studies

Design Name	Design Statistics				ILA	
	Reference	Ref. Lang.	Ref. Size	RTL Size	# of insts. (parent/child)	ILA Size (Python LoC)
RBM	System-level design [55]	SystemC	1,211†	10,578	3/14	1,009
GB (High-level)	Halide description [56]	C++	288†	6,935	2/2	538
GB (Low-level)	HLS input [56]	C++	1,718†	6,935	2/4	1561
AES (table)	RTL simulator [38]	C++	1,905	1,105	8/5	435★
AES (logic)	Software simulator [38]	C	328	-	8/7	337★
RISC-V Rocket	Chisel description [8]	Chisel	3,488‡	18,252	43	1,672★

★ILA synthesis template size. †Excluding shared library. ‡Processor core only.

and querying the status, and the computation task is the block encryption operation modeled in the `START_ENCRYPT` instruction.

4.1.1 Restricted Boltzmann Machine. Restricted Boltzmann Machine (RBM) is a stochastic neural network commonly used in recommendation systems. We model the RBM accelerator from the Columbia System Level Design Group [55]. It is implemented in SystemC and synthesized to Verilog. The accelerator supports both prediction and training, and uses the contrastive divergence learning algorithm. It exchanges data with shared memory via DMA.

We manually constructed the ILA of the RBM accelerator. The ILA captures both the interface protocol and the computation. It models the interface activities where the accelerator autonomously initiates DMA transactions to load and store training/testing datasets after receiving an initial configuration. It contains three instructions, *ConfDone*, *ReadGrant*, and *WriteGrant*, which set the configuration and grant DMA read and write transactions, respectively. The complexity of computation and DMA interaction is managed by five child-ILAs for loading, storing, coordination, training, and prediction, respectively, comprising a total of 14 child instructions. The training and predicting child-ILAs, in turn, have child-ILAs that model their computation. The computation iteratively updates two regions of private local memory for the hidden layer and visible layer in a fixed order. This order is maintained by control registers in the implementation, using child-ILA states. Recall that child-ILA states are updated by a child instruction, which activates the decode function of a subsequent child instruction.

This case study illustrates handling of both protocol and computation, the value of hierarchical ILAs, and how order is captured by the state update and decode functions of the child-ILA.

4.1.2 Gaussian Blur. The image processing accelerator performing the Gaussian Blur (GB) operation is from the Stanford VLSI Research Group [56]. Its behavior is described in Halide [57], a domain-specific language for developing high-performance image processing applications. Halide descriptions can be compiled into C++, which can then be synthesized to a Verilog implementation through high-level synthesis (HLS). The GB accelerator takes an image as streaming input, and utilizes a two-dimensional *line buffer* to collect one part of the image at a time for the GB kernel function computation. It then streams out the result for each part as soon as it is ready.

We manually construct two ILAs, GB_H and GB_L, from design descriptions at two different levels. GB_H is derived from the high-level Halide description, and models the specification. GB_L is derived from the lower-level C++ code compiled from the Halide description and models micro-architectural details. GB_H captures the size of input and output images, the streaming pattern (row-major traversal), data *source* for the kernel function, and *when* the result is ready. GB_H does

not specify how streamed data is buffered, whereas GB_L additionally includes a specific *line buffering* mechanism [56].

In this case study, we focus on specifying the streaming data interface and the output image accumulation. The kernel computation is modeled as an uninterpreted function, a standard practice in verification to allow decoupling of control verification from data-intensive computations that can be verified separately. (This is supported by standard SMT solvers, described in Section 5.1.) Both GB_H and GB_L have two instructions, *WRITE* and *READ*, that represent sending and receiving a pixel to and from the I/O boundary, respectively. The two ILAs have the same instruction set, i.e., the same hardware interface, but have different levels of abstraction. The extra complexity of GB_L in modeling the two-dimensional line buffer and stream buffers is captured by its child-ILAs; child-instructions model data movement between different components.

This case study serves to illustrate the ability of the ILA to model (i) streaming I/O and (ii) different levels of abstraction for the same instruction set through additional micro-architectural detail.

4.1.3 Advanced Encryption Standard. This case study, introduced in Section 3, considers a cryptographic engine from OpenCores [38] implementing the Advanced Encryption Standard (AES). The accelerator receives configurations via memory-mapped I/O and uses DMA to exchange data with shared memory. The configuration includes the encryption key, initial counter value, plaintext location, and length, which are stored in registers mapped to the memory address space. This accelerator works in AES-CTR mode [44], where the plaintext is fetched from the shared memory starting from the location pointed to by the *plaintext location*. The accelerator operates in the following sequence: fetch one block from memory, apply exclusive-OR operation between plaintext and the AES encrypted counter to get the ciphertext, and then store the block back into the same location. Each block has 128 bits and the complete encryption operation has 10 rounds. The ILA model uses child-ILAs for modeling the encryption function.

We synthesize two different ILAs using template-driven synthesis [62]. These ILAs, AES_C and AES_V, are based on C and Verilog implementations, respectively. They have the same architectural instruction set, but with differences in the block-level and round-level implementations in their micro-instructions.

The instructions on the interface has been shown in Figure 1. Only `START_ENCRYPT` instruction has child instructions, depicting block-level encryption. At the block level, AES_V has more child state variables (mostly counters and control signals), and its memory access is modeled at a finer granularity than AES_C. At the round level, there is one micro-instruction for each round. AES_V uses a table look-up, while AES_C uses logical operations. We capture these differences in their micro-ILAs. This case study illustrates the ability of the ILA to describe two different implemented algorithms for the same set of instructions.

4.2 General-Purpose Processors

The ILA of a general-purpose processor is based on its ISA, and the ILA has the same instructions and semantics. However, in contrast to existing formal ISA models (e.g., ISA-Formal [59]), our model has a uniform treatment of interrupts (and possibly other input signals) and instructions, rather than treating interrupts as a special case. Further, it supports hierarchy and distinguishes sub-instructions from micro-instructions; this is missing in previous work.

4.2.1 RISC-V. RISC-V is a free and open ISA with increasing adoption in industry and academic research. It has a base ISA with several extensions for advanced functionality. We synthesize the ILA of the base integer ISA RV32I with the `DefaultRV32Config` of Rocket—a single-issue in-order five-stage pipeline implementation (part of the Rocket Chip SoC generator) [8].

The ILA covers (1) user-level base integer registers and instructions, (2) machine-level control status registers (CSRs), (3) environment call/trap return instructions, (4) the address translation and the memory-management fence, and (5) interrupt and hardware interrupt handling. The semantics of each instruction are as follows: if an interrupt occurs, the next state is updated as the result of the interrupt. Otherwise, the state update is performed according to the instruction word. This case study demonstrates modeling interrupts and instructions uniformly. The RISC-V ISA exposes the synchronization between the memory hierarchy and the translation lookahead buffer (TLB) through the `SFENCE.VMA` instruction. The lack of synchronization could result in stale page table entry (PTE) references. The TLB in the RISC-V ISA is software visible, and we include it in our ILA model as an architectural state variable. However, its size, associativity, and other parameters are not specified by the ISA specification, so we model it as a ghost TLB, which can potentially hold any PTE that has been referred to but has not been explicitly flushed out. As a 32-bit RISC-V model, it only models the `Sv32` virtual addressing in addition to `Bare` mode. Memory consistency issues are beyond the scope of the current case study and thus not modeled. (Memory consistency is briefly discussed in Section 6.1.)

4.3 Summary

From these case studies and the data in Table 2, we make the following observations:

- —Accelerator ILAs tend to have a small number of instructions/child-instructions. That is, most accelerators can be specified by just a handful of instructions.
- —The same design can be modeled using ILAs at differing levels of detail. (In the next section we show how these different models are checked for equivalence.)
- —The ILA model (or template, when the ILA was synthesized) has size comparable to a reference design in C/SystemC/C++/Chisel. Thus, the value of its *formal* model comes at no additional cost, in terms of the size of a reference description.
- —The ILA model (or template) is *significantly smaller* than the final RTL implementation, making this an attractive entry point for verification and validation.

5 CASE STUDIES: VERIFICATION

The ILA model can represent specifications or implementations of hardware modules. In this article, we focus on using ILAs for hardware verification to check that implementations of accelerators/processors match their ILA architectural specifications. This also enables checking that different implementations of an accelerator have the same behavior at their interface specified by an ILA, thereby proving their architecture-level equivalence.

We briefly touch on the underlying formal verification techniques, then discuss ILA-based verification, and finally describe their evaluation on our case studies.

5.1 Underlying Formal Verification Techniques

SMT solvers [17, 70] provide decision procedures for first-order logic formulas in background theories, and have found numerous applications in verification. In this work, we use quantifier-free formulas that use the theories of arrays, uninterpreted functions and bitvectors (`QF_AUFBV` in the SMTLIB standard [11]).

Model checking is a verification technique to check correctness properties for a finite state transition system [25, 48]. Unbounded model checking explores all reachable states of the transition system while bounded model checking (BMC) [15] restricts the search to all states reachable within the first k transitions of the system. k is referred to as the *bound* and is typically set by the

verification engineer. BMC alone cannot prove the absence of property violations; however, it is very effective for bug finding in practice [26].[3]

5.2 ILA-Based Verification

As described in Section 3, the ILA model is a labeled state transition system, but one that emphasizes modularity and hierarchy. These features simplify verification through decomposition along (child-)instructions and architectural state elements. We consider two main settings for ILA-based verification: (i) ILA vs. ILA and (ii) ILA vs. FSM. The equivalence of these models is based on bisimulation relations on the underlying labeled state transition systems [49]. (It is also straightforward to consider stuttering in addition, or extend our discussion to model refinement by using simulation relations and containment checks instead.)

5.2.1 ILA vs. ILA Verification. As the GB and AES case studies described in Section 4.1 illustrate, we can construct ILAs for designs with differing implementations, or even at different levels of abstraction. A natural application is to check these ILAs for equivalence. In this setting, we compare two ILAs with the same instructions and sub-instructions, but with possibly different micro-instructions in the implementation. For ILAs, instruction-based modularity provides the basis for establishing correspondence between two models, i.e., we check that the behavior of the ILAs is the same for each instruction and sub-instruction.

Consider first the case where we do not have micro-instructions (implementations) in the ILA models. Given ILAs X and Y, we check that the issuing condition and the next-state transition updates for each instruction and sub-instruction are equivalent in the two models. Specifically, the equivalence for (sub-)instruction i is verified by checking

(i) equivalence of the valid function: $\forall S, W.(V^X(S,W) \leftrightarrow V^Y(S,W))$;
(ii) equivalence of the decode function: $\forall S, W.(\delta_i^X(S,W) \leftrightarrow \delta_i^Y(S,W))$; and
(iii) equivalence of state updates: $\forall S, W.(\delta_i^X(S,W) \wedge \delta_i^Y(S,W) \rightarrow (N_i^X(S,W) = N_i^Y(S,W)))$.

Note, X and Y are shown with the same state variables here, but this can be generalized to a mapping between their variables.

Now consider the case where we have micro-instructions in the ILA model(s), to represent micro-architectural implementation choices. We do not enforce equivalence at the micro-instruction level. Instead, we check the equivalence of each instruction and sub-instruction, where each may be implemented using a sequence of micro-instructions. Here, we check equivalence *after* the sequence of micro-instructions that implements an instruction/sub-instruction is completed.

To check the equivalence for each instruction, we may need additional abstraction/refinement mappings to establish "corresponding" states between the two models. Thus, the equivalence check essentially says that if we start in corresponding states and apply an instruction, then we end in corresponding states. Here, we leverage well-studied processor verification techniques [19, 42] that propose and use such mappings. In addition, we use invariants to prune some unreachable micro-architectural states, such as the invalid combination of the horizontal/vertical frame pointers of an image in the GB case study. These are often needed to prove the correspondence checks.

5.2.2 ILA vs. FSM Verification. In this setting, we are interested in verifying that a hardware implementation available as an FSM model (e.g., RTL) corresponds to its ILA specification. As before, the equivalence between an ILA model and an FSM model is checked for each instruction and sub-instruction in the ILA. However, unlike the ILA that has a clear set of instructions, an

[3]The success of BMC is often ascribed to the "small world hypothesis": bugs (inadvertent mistakes, as opposed to maliciously introduced design flaws) are likely reachable through *some* short sequence of steps from the initial state.

Table 3. ILA Verification Experiments

Category	Designs	Models	Tools	Strength of Proof	Time
ILA vs. ILA	GB	GB_H vs. GB_L	JasperGold	complete	2h 27m
	AES	AES_C vs. AES_V	ILA lib+JasperGold†	complete	15m
ILA vs. FSM	GB	GB_H vs. Verilog	JasperGold	complete	2h 50m
		GB_L vs. Verilog	JasperGold	complete	16h 12m
	RBM	ILA vs. SystemC	CBMC	complete	2h 7m
		ILA vs. Verilog	JasperGold	complete	6h 54m
	RISC-V Rocket	ILA vs. Verilog	JasperGold	complete (invariants)	5h 40m
				complete (interrupt)	8m
				BMC to 40 cycles (instructions)	86h 5m

†ILA library (using Z3) for block-level ILA equivalence, JasperGold for round-level equivalence.

FSM model is generally a monolithic transition system without a separation between the parts implementing different instructions/sub-instructions.

Again, we leverage well-studied processor verification techniques [19, 42], and use refinement mappings to relate the FSM states to the ILA states for each (sub-)instruction. Invariants are also used to prune unreachable micro-architectural states in the FSM model, e.g., the invariant on a one-hot encoded counter in the RTL implementation.

Note that ILAs enable a discipline for accelerator implementation verification that is based on established methodology for processor verification. This is in contrast to customizing general hardware verification techniques for this task, since determining what/when to check is itself a challenge and in practice woefully incomplete.

5.3 Experimental Evaluation

We have implemented the ILA-based verification techniques described above, on top of off-the-shelf verification tools (Z3 [30], JasperGold [20], and CBMC [24]). The correspondence checks on instructions are expressed as verifying the assertions where the two models end in corresponding states, given the assumptions that they start in corresponding states and apply the same instruction. For example, for the Gaussian-Blur accelerator, we check the correspondence of frame pointers by assuming the two models initially have an equal horizontal/vertical frame pointer pair. Then, we verify that their frame pointers are equal after the *WRITE* instruction, regardless of the pixel accumulating and buffering mechanism. Our ILA library supports translation of the ILA models into formats supported by these tools. Verification results are summarized in Table 3.

5.3.1 ILA vs. ILA Verification.

GB Accelerator: Recall that we constructed two ILAs (see Section 4.1) for the GB accelerator. The ILA GB_H follows the high-level Halide code, and GB_L follows the lower-level C++ code. The two ILAs have the same instruction set, but GB_L has additional micro-instructions to describe stream buffer and pixel accumulation operations. We check equivalence of each instruction using the Burch-Dill approach [19], where we use a "flushing" function to relate corresponding states in the two ILAs. This is needed to abstract away intermediate micro-architectural states in GB_L that are not visible in GB_H. Specifically, the checked instruction starts in a state in GB_L where there is no buffered intermediate data. Thus, for each instruction, we check that the architectural states (IO ports, image frame, pixel pointers, etc.) are equal at the end, whenever the ILAs start in corresponding initial states. Verification completed in about 2.5 hours using JasperGold.

AES Accelerators: The two ILAs of the AES accelerator were described in Section 4.1. They have the same instructions at the top level, but different micro-instructions due to different implementations of the encryption algorithm. We leveraged the hierarchy in ILA models to decompose equivalence checking into block-level and round-level equivalence checks.

The AES encryption is a 10-round operation. As both models have one micro-instruction for each round, we first check the equivalence of such micro-instructions. At the round level, we check that the generated round keys and ciphertexts are matched after the execution, given their round keys and cipherstates are matched before the execution. The micro-instructions and the verification conditions for checking are automatically converted into Verilog to take advantage of hardware verification tools, which are better at reasoning logic operations in AES encryption. Based on the equivalence of round-level micro-instructions, we check the equivalence of the 10-round AES operation by modeling the round-level encryption as an uninterpreted function.

The block-level operations involve fetching plaintext, encrypting data, storing ciphertext, and maintaining encryption states, e.g., the counter. We check that after processing one block, the two models should have the same ending state (including shared memory and registers in the accelerator) if they start from the same state. By proving the equivalence of the micro-instructions performing block-level operations, the equivalence of `START_ENCRYPT` instruction, which processes series of blocks, can be guaranteed. We used our ILA library, which in turn uses the Z3 SMT solver [30], for checking block-level equivalence, and use JasperGold for checking round-level equivalence. The total verification time was about 15 minutes.

These two case studies show that ILA equivalence checking can be applied to bridge the gap between models at different abstraction levels associated with design languages (Halide vs. C++, C vs. Verilog).

5.3.2 ILA vs. FSM Verification. We consider FSM models at the register transfer level (e.g., in Verilog) or system level (e.g., in C/SystemC). We check ILA vs. FSM equivalence for two accelerators and a general-purpose processor. All FSM models are provided independently by other groups, and not synthesized from ILAs: RBM-SystemC model by the Carloni-Columbia group [55]; Gaussian-Blur-Halide/C++ model by the Horowitz-Stanford group [56]; AES-C/RTL implementation from OpenCores.org [38]; RISC-V implementation from Berkeley's Rocket-chip generator [8]. Our previous work [62, 64] has discussed the verification of the 8051 micro-controller and SHA accelerator, where eight bugs were found in the RTL model in 8,051.

GB Accelerator: We performed equivalence checking between the RTL implementation (generated by HLS) and each of the two ILAs, GB_H and GB_L, separately. GB_L models more detailed behavior such as buffering and pixel accumulation, which is similar to the RTL implementation. We provided invariants to establish corresponding states, and successfully completed verification against each ILA model. As expected, the verification of RTL against the more detailed GB_L took more time than against GB_H (≈16h vs. 3 h).

Restricted Boltzmann Machine: We exploited the structural similarity between SystemC, Verilog, and the ILA models to expedite equivalence checking through modular checking. We replaced some functions in the computation, e.g., the *sigmoid* function, with uninterpreted functions. (Verification of these functions can be addressed separately.) We successfully completed verification of the ILA vs. SystemC (≈2h), as well as the ILA vs. Verilog (≈7h). This example demonstrates that a single ILA can be matched against multiple FSM models with implementation-specific differences.

ILA for RV32I vs. Rocket: We synthesized the ILA for the RISC-V specification and verified this against Verilog of the Rocket processor core generated from a Chisel description [8]. The verification settings can be found in directory `RISC-V/ILAVerif` in our GitHub repository. Our focus was

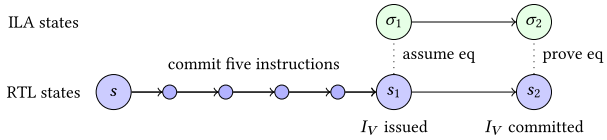


Fig. 3. ILA vs. FSM verification of instruction execution in Rocket.

on the processor core, and we separate it from the memory system and the branch predictor. We abstract the branch predictor by constraining the interface of the processor core where any valid prediction can arrive in any cycle.

Our verification had three main steps.

(1) First, as discussed in Section 5.2.2, we use implementation invariants to prune unreachable states in per-instruction equivalence checking. The first category of invariants targets the correctness of the bypassing network. That is, for each general-purpose register, the value bypassed to the decode stage must be the same as the corresponding when the instruction at that stage commits. The second category ensures the multiplication/division unit and co-processors do not generate valid response signals when executing integer instructions. These invariants are verified using the unbounded model checking engines of JasperGold.
(2) Next, we verified interrupt handling. The RTL handles interrupts by inserting dummy instructions in the pipeline, corresponding to the interrupt instruction in the ILA. We proved that RTL and ILA states match when the interrupt commits (using JasperGold).
(3) Finally, we checked equivalence on ordinary instructions using the inductive proof strategy shown in Figure 3. The processor is started in an arbitrary state s constrained by the invariants described in Step (1). Five[4] instructions are issued, leading to a state s_1 where we assume that they have been correctly committed, i.e., the ILA state σ_1 and RTL state s_1 are equal. Then, a new instruction I_V is issued, and we check whether ILA state σ_2 and RTL state s_2 match when I_V commits. We were unable to complete an unbounded proof of this property. However, except for the bug discussed below, there was no violation up to a bound of 40 cycles using BMC (from s to s_2).

    Note that the latency of an instruction depends on the response latency from modules like the data cache. Therefore, it is possible that 40 cycles are not sufficient to *guarantee* that I_V commits correctly. Future work will build a memory model that can prove full correctness to avoid this limitation.

The two main challenges in verifying the Rocket core are finding a sufficiently strong set of invariants so that the inductive proof (Step (3)) above succeeds and specifying the refinement relation.

Deriving Pipeline Invariants for Rocket Verification: We derived the "strengthening" invariants using a counter-example guided approach. Initially, we attempted the inductive check for instruction equivalence starting with an unconstrained state (i.e., no invariants). This resulted in spurious counter-examples where the inductive proof failed when starting from unreachable states. Analysis of these states helped us formulate the set of invariants described in Step (1). These invariants were checked using the unbounded model checking and then used to constrain the starting state

[4]In our experiments, five instructions led to an over-approximation of the reachable states that is "strong" enough to prove equivalence.

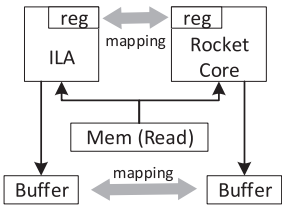


Fig. 4. The compound transition system used in ILA vs. Rocket processor core equivalence checking.

for the inductive proof described in Step (3) above. The base case of the inductive proof: ensuring that five instructions commit correctly starting from the reset state was verified separately.

Deriving Refinement Relations for Rocket Verification: As discussed in Section 5.2, the equivalence in Figure 3 is defined with respect to refinement, which consists of a mapping between the states in Rocket implementation and our ILA model. The states here involve general-purpose registers, CSRs, the program counter, and memory. The refinement relations for each of these state variables are as follows:

(1) General-purpose registers and CSRs: This refinement relation specifies that the register values in the ILA model and the Rocket implementation must be equal after each instruction commits.
(2) Program counter: The program counter's refinement relation is a little trickier due to branch prediction and speculative execution. In the Rocket implementation, every pipeline stage except the fetch stage possesses a program counter variable corresponding to the current instruction in that stage. The refinement relation for the PC specifies that the program counter value of the commit stage of the Rocket implementation and the corresponding program counter value in the ILA model should be equal after each instruction commits.
(3) Memory: Instead of modeling two individual memories and checking value equivalence over all addresses, we use a shared memory for all memory read operations, and store all the memory write operations separately for comparison. Equivalence requires that the changes to memory should be the same when I_V commits. We abstract the memory for read operations by returning arbitrary values for irrelevant read requests, and only enforcing the equivalence on the requests from I_V.

To track the stages where current instruction I_V resides, we use a sequence monitor to store the corresponding stages and use these in specifying the refinement relations.

The compound transition system in Figure 4 shows how we show that the Rocket implementation refines the ILA model. This compound transition system has more than 4k bits of flip-flops and 320k gates (reported by JasperGold). JasperGold uses both bounded and ubounded model checking techniques on this transition system and any trace violating the refinement relations indicates the two models are not equivalent.

Rocket Implementation Bug: We found a bug where the Rocket core incorrectly implements the trap return instructions. According to the specification [69] these instructions should set the *xPIE* bits in *mstatus* register to 1. However, the implementation sets them to 0. We reported this bug, and it has been fixed since. This case study illustrates the usefulness of our approach on real processors.

5.4 Summary of Verification Experiments

From Table 3, we observe that most verification experiments can derive a complete proof where "complete" refers to either unbounded proofs, or running BMC to the upper bounds of the instructions' latencies. Instructions on the Rocket core are checked up to a bound of 40 cycles, which is incomplete, but does provide a significant level of assurance.

Overall, our evaluation confirms the viability of equivalence checking using ILAs, where we leverage the ILA modularity and hierarchy on top of existing verification tools and processor verification methodology, to successfully verify a range of accelerators and processors. Our case studies cover the verification of computation (AES and RBM) as well as processor/accelerator interfaces (GB and RBM), which is important for accelerator verification.

6 DISCUSSION AND FUTURE WORK

While this article focuses on the application of the ILA model in verification of a single compute engine (processor or accelerator), the ILA has other applications as discussed below.

6.1 Modeling Concurrency and Memory Consistency

The ILA model views compute engines (processors and accelerators) as processing a sequence of instructions. Although the underlying hardware may operate on these instructions in parallel (similar to pipelined microarchitectures for processors), the programming abstraction it provides is that of a single sequential thread of control (similar to the ISA-based programming model).

As a next step, we believe that individual ILA models can be composed to perform reasoning over a *concurrent* system of multiple accelerator/processor cores. Here, the large body of work on concurrent programs and multiprocessor systems can be leveraged, and potentially extended to accelerator-rich systems using ILAs. One natural application is to use concurrent program verification techniques for checking correctness properties at the system level. This would include use of well-known methods and tools, such as software model checking with partial order reduction [25, 37] and compositional frameworks for thread-modular reasoning [33, 52].

Another promising application is in verification of memory consistency models, which capture rules about operations on shared memory. The ISA plays a central role in many efforts related to verification of memory consistency—correctness of compiler mappings for higher-level languages [12, 13], correctness of microarchitecture implementations (including coherence and virtual memory subsystems) with respect to ISA and microarchitecture specifications [46, 47], and more recently at the trisection of software, hardware, and ISA [66]. Furthermore, there have been recent advances in automatic methods for verification [2] and synthesis [18] of axiomatic memory models.

Note that these techniques and tools are not currently directly applicable to accelerators, where the hardware is described with low-level FSM models (e.g., RTL Verilog). More importantly, since the accelerator memory operations are generally not visible to the processor, ignoring these interactions with shared memory can have adverse consequences for checking correctness or security of the overall system. Modeling accelerator behavior as an ILA allows application (and potential extensions) of these known ISA-based techniques. We are currently working on memory consistency modeling for a general shared memory system with multiple processors and accelerators.

Admittedly, this approach does not *yet* address the challenging issues that currently pervade memory consistency verification using ISAs. However, it allows some separation of concerns, whereby good solutions for ISAs can be adapted for ILAs to extend their reach to accelerators.

6.2 Accelerator Code Generation

Accelerators provide efficient hardware implementations of functions that can be offloaded from programmable processors. When accelerators are deployed, an important and error-prone task is to program the accelerator to invoke these functions. As discussed in Section 1, the processor-accelerator interactions often useMMIO. Even when a single ILA instruction implements a significant function (e.g., block encryption in the AES accelerator example), other instructions must precede this encryption instruction to set up the encryption key, address of the block, size of the block, and so forth. Thus, a sequence of instructions is needed to completely implement this function. This sequence is often referred to as the accelerator driver code and is typically provided as library code with the accelerator. For this code to be correct—the instructions that set up the accelerator must be correct, as must the main accelerator function itself. The ILA model enables correct code synthesis using well-known program synthesis techniques (e.g., [3, 41]). In this setup, program synthesis seeks a program with k ILA instructions that is equivalent to the software function f it is replacing. Function f serves as an oracle to guide the search, and the ILA model provides the accelerator instruction semantics for use in the SMT solver based search for the program with k-instructions. While this has not yet been implemented, the fact that these driver programs are short (i.e., k is small) suggests promise for this useful application of the ILA model.

6.3 Reliable Simulator Generation

Given an ILA model, a reliable hardware simulator can be automatically generated for use in system/software development. The ILA model specifies the state update functions of the architectural state variables. Through hierarchy, it may optionally provide additional micro-architectural detail. These functions can be used to construct an executable model (i.e. a simulator) in almost any programming language (we currently use C++). As this simulator is generated from a formal specification that can be verified against the detailed RTL hardware model, this makes it a *reliable* executable model.

Mismatches between a simulator and the hardware it models is a common problem for software (especially OS) developers. This problem can be addressed through generation of reliable simulators. As an illustrative example, we note that a previous version of the seL4 RISC-V port makes no use of the supervisor memory-management fence (`SFENCE.VMA`) instruction, but still executes correctly on the spike ISA simulator. The simulator flushes the translation look-ahead buffer more frequently than either the Rocket implementation or the RISC-V specification's minimum requirement.

We checked if the missing fence instruction would cause a problem. We removed the gratuitous TLB flushes in the simulator and embedded an address translation monitor to check whether any address translation uses a stale page table entry. The OS crashed on this modified simulator, and stale page table references were observed. This illustrates that the missing `SFENCE.VMA` could crash on a seL4 RISC-V port with a hardware implementation that conforms only to the minimum requirement in the specification. This mismatched behavior between the simulator and the hardware would be a problem if the OS were later ported to run on real hardware. Although the missing fence instruction has been added by the seL4 developers in a newer release, the simulator behavior of gratuitous TLB flushes has not been changed. The RISC-V community knows that the spike ISA simulator represents only one possible implementation of RISC-V, and that this might be different from a hardware implementation. However, we believe that it is useful to have an ISA-level simulator that represents the specification or matches a specific hardware implementation, so that software developers can be more confident about test results with the simulator.

7 RELATED WORK

To the best of our knowledge, this is the first work to formally model accelerator interfaces using the notion of instructions similar to ISAs for processors. Our previous work in [62, 64] did introduce the notion of instructions as accelerator abstractions, but did not provide a formal model of execution. Instead, its focus was on template-based synthesis of these abstractions. Further, these abstractions were defined as finite state transition systems, with no notion of hierarchy and no applicability to processors. In this work, we introduce the formal ILA model, with hierarchy (sub- and micro-instructions), that can be used uniformly across processors and accelerators. In addition, we provide an extensive evaluation of its modeling and verification capabilities on a diverse set of accelerator and processor designs. Past work [63] has also shown how abstractions can be used for hardware/software co-verification. In contrast, the focus of this article is on verification of hardware implementations against ILA specifications.

Formal machine-readable and precise specifications [32, 58] of ARM and x86 processors have been developed. ISA-Formal [59] is a framework aimed at verification of ARM processors against ISA specifications [58]. However, as discussed earlier, this does not distinguish between different forms of hierarchy (sub-instructions vs. micro-instructions) needed for correct verification. Further, interrupts require special handling in their instruction semantics. Others [19, 42] have targeted verification of processor microarchitecture w.r.t. the ISA. These works target general-purpose processors and do not address verification of accelerators. As discussed, we build on these techniques for verifying accelerator implementations against their ILA specifications.

As discussed in Section 2, many efforts over the years have proposed the use of high-level models in design and verification. These include StateCharts, SystemC, Esterel, Transaction Level Modeling (TLM), BlueSpec, and others [9, 14, 34, 36, 53]. In particular, BlueSpec has been used as a high-level specification and design language in industry and research [7, 51]. It models hardware components as atomic rules of state transition systems and enables easy exploration of microarchitectural design space, e.g., adding a buffer in a pipeline. The commercial BlueSpec compiler synthesizes the circuit implementation, i.e., Verilog, and exploits parallelism with a scheduler determining how to interleave the atomic rules [28, 36]. BlueSpec has a well-defined operational semantics and supports modular verification using SMT solvers [29] and interactive-theorem provers [23, 67]. While the use of high-level models helps raise the level of abstraction, and hence improves scalability in design and verification, all of these models including BlueSpec lack two essential ILA features: a clean separation between hardware and software concerns, and uniform instruction-level treatment of processors and accelerators. This limits their use in hardware/software co-verification and scalable verification of systems with heterogeneous hardware components.

A number of hardware/software co-synthesis frameworks [21, 50, 54, 65] attempt to automatically generate both firmware and accelerator hardware from an algorithmic description. While these efforts may side-step the need for abstractions for co-verification through correct-by-construction claims, reasoning about their correctness will itself require a principled abstraction of hardware with the key ILA features stated above.

Property validation of hardware over Verilog/VHDL models has been advancing since the adoption of novel model checking techniques, e.g., [40, 43, 68]. These works are orthogonal to our work. Our key contribution is using ILA as a functional specification of processors/accelerators, and enabling the use of existing processor verification techniques for accelerator verification. The verification problem is to check equivalence of instruction-level vs. RTL models, and not validating individual properties in Verilog/VHDL models, which would otherwise need to be specified for capturing full functionality.

8 CONCLUSIONS

This article presents the ILA as a formal model for accelerators to address the heterogeneity challenges of emerging computing platforms. The ILA is a uniform model, usable across heterogeneous processors and accelerators. Further, it raises the level of abstraction of the accelerators to that of the processors, enabling formal software-hardware co-verification. The ILA has several valuable attributes for modeling and verification. It is modular, with functionality expressed as a set of instructions. It enables meaningful abstraction through architectural state that is persistent across instructions. It provides for portability through a more durable interface with the interacting processors. It is hierarchical, providing for multiple levels of abstraction for modeling complex instructions as a software program through sub- and micro-instructions. It enables leveraging processor verification techniques for verifying accelerator implementations. This allows for accelerator replacement using the notion of ILA compatibility similar to that of ISA compatibility.

We demonstrate the value of these attributes through modeling and verification of a range of accelerators (RBM, AES, and Gaussian Blur) and a processor (RISC-V Rocket processor core). We identify modeling gaps in previous formal modeling of ISAs (ISA Formal's lack of distinction between hierarchy in specification vs. implementation) and a bug in the implementation of the RISC-V Rocket core. Further, we demonstrate substantially complete model checking based verification for our case studies. Regarding scalability, our verification for accelerators from OpenCores (AES) and processors (Rocket Chip) are the targets over the next 4 years in the current DARPA POSH BAA. Finally, we highlight additional applications of the ILA model in reasoning about concurrency and memory consistency with accelerators, accelerator code generation, and reliable simulator generation. Overall, these results and contributions provide significant evidence of the value of ILAs in accelerator-based modeling and verification.

REFERENCES

[1] Samar Abdi and Daniel Gajski. 2006. Verification of system level model transformations. *International Journal of Parallel Programming* 34, 1 (2006), 29–59. DOI: https://doi.org/10.1007/s10766-005-0001-y

[2] Jade Alglave and Michael Tautschnig. 2014. Herding cats: Modelling, simulation, testing, and data-mining for weak memory. *ACM Transactions on Programming Languages and Systems* 36, 2 (2014), 7:1–7:74. DOI: https://doi.org/10.1145/2627752

[3] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Proceedings of the Conference on Formal Methods in Computer-Aided Design*. 1–8. DOI: https://doi.org/10.1109/FMCAD.2013.6679385

[4] Rajeev Alur and Radu Grosu. 2000. Modular refinement of hierarchic reactive machines. In *Proceedings of the Symposium on Principles of Programming Language*. 390–402. DOI: https://doi.org/10.1145/973097.973101

[5] Ehsan K. Ardestani and Jose Renau. 2013. ESESC: A fast multicore simulator using time-based sampling. In *Proceedings of the International Symposium on High-Performance Computer Architecture*. 448–459. DOI: https://doi.org/10.1109/HPCA.2013.6522340

[6] ARM Ltd. 2010. Cortex-M3 Devices Generic User Guide. Retrieved November 11, 2017 from http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0552a/BABCAEDD.html.

[7] Arvind and Xiaowei Shen. 1999. Using term rewriting systems to design and verify processors. *IEEE Micro* 19, 3 (May 1999), 36–46. DOI: https://doi.org/10.1109/40.768501

[8] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. *The Rocket Chip Generator*. Technical Report UCB/EECS-2016-17. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[9] Francine Bacchini, Daniel D. Gajski, Laurent Maillet-Contoz, Haruhisa Kashiwagi, Jack Donovan, Tommi Makelainen, Jack Greenbaum, and R. S. Nikhil. 2007. TLM: Crossing over from buzz to adoption. In *Proceedings of Design Automation Conference*. 444–445. DOI: https://doi.org/10.1109/DAC.2007.375205

[10] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing hardware in a scala embedded language. In *Proceedings of Design Automation Conference*. 1212–1221. DOI: https://doi.org/10.1145/2228360.2228584

[11] Clark Barrett, Aaron Stump, and Cesare Tinelli. 2010. The SMT-LIB standard version 2.0. In *Proceedings of the International Workshop on Satisfiability Modulo Theories*. 14–112.

[12] Mark Batty, Kayvan Memarian, Scott Owens, and Susmit Sarkar. 2012. Clarifying and compiling C/C++ concurrency: From C++ 11 to POWER. In *Proceedings of the Annual Symposium on Principles of Programming Languages*. ACM, New York, 509–520. DOI: https://doi.org/10.1145/2103656.2103717

[13] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *Proceedings of the Annual Symposium on Principles of Programming Languages*. 55–66. DOI: https://doi.org/10.1145/1925844.1926394

[14] G. Berry, M. Kishinevsky, and S. Singh. 2003. System level design and verification using a synchronous language. In *Proceedings of the International Conference on Computer-Aided Design*. 433–439. DOI: https://doi.org/10.1109/ICCAD.2003.1257813

[15] Armin Biere, Alessandro Cimatti, and Edmund M Clarke. 2003. Bounded model checking. *Advances in Computers* 58 (2003), 117–148.

[16] Nathan Binkert, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, David A. Wood, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, and Tushar Krishna. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7. DOI: https://doi.org/10.1145/2024716.2024718

[17] Nikolaj Bjorner and Leonardo De Moura. 2011. Satisfiability modulo theories: Introduction and applications. *Communications of the ACM* 54, 9 (2011), 69–77. DOI: https://doi.org/10.1145/1995376

[18] James Bornholt and Emina Torlak. 2017. Synthesizing memory models from framework sketches and litmus tests. In *Proceedings of the Conference on Programming Language Design and Implementation*. 467–481. DOI: https://doi.org/10.1145/3062341.3062353

[19] Jerry R. Burch and David L. Dill. 1994. Automated verification of pipelined microprocessor control. In *Proceedings of the International Conference on Computer Aided Verification*. 68–84. https://dl.acm.org/citation.cfm?id=735662

[20] Cadence Design Systems, Inc. 2018. JasperGold: Formal Property Verification App. Retrieved January 2, 2018 from http://www.jasper-da.com/products/jaspergold-apps/.

[21] Andrew Canis, Jongsok Choi, Mark Aldham, and Victor Zhang. 2013. LegUp: An open-source high-level synthesis tool for FPGA-Based processor/accelerator systems. *ACM Transactions on Embedded Computing Systems* 13, 2 (2013), 24:1–24:27. DOI: https://doi.org/10.1145/2514740

[22] Wei Ting Jonas Chan, Andrew B. Kahng, Siddhartha Nath, and Ichiro Yamamoto. 2014. The ITRS MPU and SoC system drivers: Calibration and implications for design-based equivalent scaling in the roadmap. In *Proceedings of the International Conference on Computer Design*. 153–160. DOI: https://doi.org/10.1109/ICCD.2014.6974675

[23] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, and Adam Chlipala. 2017. Kami: A platform for high-level parametric hardware specification and its modular verification. *Proceedings of the ACM on Programming Languages* 1, 24 (2017). DOI: https://doi.org/10.1145/3110268

[24] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. CBMC - A tool for checking ANSI-C programs. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Vol. 2988. 168–176. DOI: https://doi.org/10.1007/978-3-540-24730-2_15

[25] Edmund M. Clarke, Orna Grumberg, and Doron Peled. 1999. *Model Checking*. MIT Press.

[26] Fady Copty, Limor Fix, Ranan Fraer, Enrico Giunchiglia, Gila Kamhi, Armando Tacchella, and Moshe Y. Vardi. 2001. Benefits of bounded model checking at an industrial setting. In *Proceedings of the International Conference on Computer Aided Verification*. 436–453. DOI: https://doi.org/10.1007/3-540-44585-4_43

[27] Emilio G. Cota, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P. Carloni. 2015. An analysis of accelerator coupling in heterogeneous architectures. In *Proceedings of Design Automation Conference*. 202:1–202:6. DOI: https://doi.org/10.1145/2744769.2744794

[28] Nirav Dave, Arvind, and Michael Pellauer. 2007. Scheduling as rule composition. In *Proceedings of the IEEE/ACM International Conference on Formal Methods and Models for Codesign*. IEEE, 51–60. DOI: https://doi.org/10.1109/MEMCOD.2007.371249

[29] Nirav Dave, Michael Katelman, Myron King, Jose Arvind, and Jose Meseguer. 2011. Verification of microarchitectural refinements in rule-based systems. In *Proceedings of the ACM/IEEE International Conference on Formal Methods and Models for Codesign*. IEEE, 61–71. DOI: https://doi.org/10.1109/MEMCOD.2011.5970511

[30] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 337–340. DOI: https://doi.org/10.1007/978-3-540-78800-3_24

[31] Rainer Dömer, Andreas Gerstlauer, Junyu Peng, Dongwan Shin, Lukai Cai, Haobo Yu, Samar Abdi, and Daniel D. Gajski. 2008. System-on-chip environment: A SpecC-based framework for heterogeneous MPSoC design. *EURASIP Journal on Embedded Systems* (2008), 5:1–5:13. DOI: https://doi.org/10.1155/2008/647953

[32] Shilpi Goel, Warren A. Hunt, Matt Kaufmann, and Soumava Ghosh. 2014. Simulation and formal verification of x86 machine-code programs that make system calls. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*. 91–98. DOI: https://doi.org/10.1109/FMCAD.2014.6987600

[33] Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. 2011. Threader: A constraint-based verifier for multi-threaded programs. In *Proceedings of the International Conference on Computer Aided Verification*. 412–417. DOI: https://doi.org/10.1007/978-3-642-22110-1_32

[34] David Harel and Amnon Naamad. 1996. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology* 5, 4 (1996), 293–333. DOI: https://doi.org/10.1145/235321.235322

[35] Paula Herber and Sabine Glesner. 2013. A HW/SW co-verification framework for systemC. *ACM Transactions on Embedded Computing Systems* 12, 1 (2013), 61:1–61:23. DOI: https://doi.org/10.1145/2435227.2435257

[36] James C. Hoe and Arvind. 2000. Synthesis of operation-centric hardware descriptions. In *Proceedings of the International Conference on Computer-Aided Design*. 511–519. DOI: https://doi.org/10.1109/ICCAD.2000.896524

[37] Gerard J. Holzmann. 1997. The model checker SPIN. *IEEE Transactions on Software Engineering* 23, 5 (1997), 279–295. DOI: https://doi.org/10.1109/32.588521

[38] Homer Hsing. 2014. OpenCores.org: Tiny AES. Retrieved November 17, 2017 from https://opencores.org/project,tiny_aes.

[39] Intel Corporation. 2016. Intel®64 and IA-32 Architectures Software Developer Manual: Vol. 2 Instruction Set Reference. Retrieved November 17, 2017 from https://software.intel.com/en-us/articles/intel-sdm.

[40] Himanshu Jain, Daniel Kroening, Natasha Sharygina, and Edmund M. Clarke. 2005. Word level predicate abstraction and refinement for verifying RTL verilog. In *Proceedings of Design Automation Conference*. 445–450. DOI: https://doi.org/10.1145/1065579.1065697

[41] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *Proceedings of the International Conference on Software Engineering*. 215–224. DOI: https://doi.org/10.1145/1806799.1806833

[42] Ranjit Jhala and Kenneth L. McMillan. 2001. Microarchitecture verification by compositional model checking. In *Proceedings of the International Conference on Computer Aided Verification*, Vol. 2102. 396–410. DOI: https://doi.org/10.1007/3-540-44585-4_40

[43] Suho Lee and Karem A. Sakallah. 2014. Unbounded scalable verification based on approximate property-directed reachability and datapath abstraction. In *Proceedings of the International Conference on Computer Aided Verification*. 849–865. DOI: https://doi.org/10.1007/978-3-319-08867-9_56

[44] Helger Lipmaa, David Wagner, and Phillip Rogaway. 2000. Comments to NIST Concerning AES Modes of Operation: CTR-Mode Encryption. Retrieved May 5, 2018 from http://kodu.ut.ee/ lipmaa/papers/lrw00/html/ctr.html.

[45] Derek Lockhart, Gary Zibrat, and Christopher Batten. 2014. PyMTL: A unified framework for vertically integrated computer architecture research. In *Proceedings of the International Symposium on Microarchitecture*. 280–292. DOI: https://doi.org/10.1109/MICRO.2014.50

[46] Daniel Lustig, Michael Pellauer, and Margaret Martonosi. 2015. Pipe check: Specifying and verifying microarchitectural enforcement of memory consistency models. In *Proceedings of the Annual International Symposium on Microarchitecture*. 635–646. DOI: https://doi.org/10.1109/MICRO.2014.38

[47] Daniel Lustig, Geet Sethi, Margaret Martonosi, and Abhishek Bhattacharjee. 2016. COATCheck : Verifying memory ordering at the hardware-OS interface. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, Vol. 1. 233–247. DOI: https://doi.org/10.1145/2872362.2872399

[48] Kenneth L. Mcmillan. 1993. *Symbolic Model Checking*. Springer.

[49] Robin Milner. 1989. *Communication and Concurrency*. Prentice Hall.

[50] Razvan Nane, Vlad Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, Jason Anderson, and Koen Bertels. 2016. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 10 (2016), 1591–1604. DOI: https://doi.org/10.1109/TCAD.2015.2513673

[51] R. Nikhil. 2004. Bluespec system verilog: Efficient, correct RTL from high-level specifications. In *Proceedings of the International Conference on Formal Methods and Models for Co-Design*. 69–70. DOI: https://doi.org/10.1109/MEMCOD.2004.1459818

[52] Susan Owicki and David Gries. 1976. An axiomatic proof technique for parallel programs. *Acta Informatica* 6, 4 (1976), 319–340. DOI: https://doi.org/10.1007/BF00268134

[53] Preeti Ranjan Panda. 2001. SystemC - A modeling platform supporting multiple design abstractions. In *Proceedings of the International Symposium on Systems Synthesis*. 75–80. DOI: https://doi.org/10.1109/ISSS.2001.156535

[54] Christian Pilato and Fabrizio Ferrandi. 2013. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *Proceedings of the International Conference on Field Programmable Logic and Applications*. 13–16. DOI: https://doi.org/10.1109/FPL.2013.6645550

[55] Christian Pilato, Qirui Xu, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P. Carloni. 2016. On the design of scalable and reusable accelerators for big data applications. In *Proceedings of the ACM International Conference on Computing Frontiers*. 406–411. DOI: https://doi.org/10.1145/2903150.2906141

[56] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. 2016. Programming heterogeneous systems from an image processing DSL. *ACM Transactions on Architecture and Code Optimization* 14 (2016), 26:1–26:25. DOI: https://doi.org/10.1145/3107953

[57] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Frédo Durand, Connelly Barnes, and Saman Amarasinghe. 2013. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the Conference on Programming Language Design and Implementation*. 519–530. DOI: https://doi.org/10.1145/2491956.2462176

[58] Alastair Reid. 2017. Trustworthy specifications of ARM® v8-A and v8-M system level architecture. In *Proceedings of the Conference on Formal Methods in Computer-Aided Design*. 161–168. DOI: https://doi.org/10.1109/FMCAD.2016.7886675

[59] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. 2016. End-to-end verification of ARM® processors with ISA-formal. In *Proceedings of the International Conference on Computer Aided Verification*, Vol. 9780. 42–58. DOI: https://doi.org/10.1007/978-3-319-41540-6_3

[60] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. 2011. DRAMSim2: A cycle accurate memory system simulator. *IEEE Computer Architecture Letters* 10, 1 (2011), 16–19. DOI: https://doi.org/10.1109/L-CA.2011.4

[61] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. 2015. The Aladdin approach to accelerator design and modeling. *IEEE Micro* 35, 3 (2015), 58–70. DOI: https://doi.org/10.1109/MM.2015.50

[62] Pramod Subramanyan, Bo-Yuan Huang, Yakir Vizel, Aarti Gupta, and Sharad Malik. 2017. Template-based parameterized synthesis of uniform instruction-level abstractions for SoC verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 99 (2017). DOI: https://doi.org/10.1109/TCAD.2017.2764482

[63] Pramod Subramanyan, Sharad Malik, Hareesh Khattri, Abhranil Maiti, and Jason Fung. 2016. Verifying information flow properties of firmware using symbolic execution. In *Proceedings of the Conference on Design, Automation and Test in Europe*. 1393–1398. DOI: https://doi.org/10.3850/9783981537079_0793

[64] Pramod Subramanyan, Yakir Vizel, Sayak Ray, and Sharad Malik. 2017. Template-based synthesis of instruction-level abstractions for SoC verification. In *Proceedings of the Conference on Formal Methods in Computer-Aided Design*. 160–167. DOI: https://doi.org/10.1109/FMCAD.2015.7542266

[65] Impulse Accelerated Technologies. 2003. Impulse CoDeveloper C-to-FPGA Tools. Retrieved November 17, 2017 from http://www.impulseaccelerated.com/products_universal.htm.

[66] Caroline Trippel, Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. 2017. TriCheck: Memory model verification at the trisection of software, hardware, and ISA. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*. 119–133. DOI: https://doi.org/10.1145/3093337.3037719

[67] Muralidaran Vijayaraghavan, Adam Chlipala, Arvind, and Nirav Dave. 2015. Modular deductive verification of multiprocessor hardware designs. In *Proceedings of the International Conference on Computer Aided Verification*. 109–127. DOI: https://doi.org/10.1007/978-3-319-21668-3_7

[68] Yakir Vizel, Orna Grumberg, and Sharon Shoham. 2012. Lazy abstraction and SAT-based reachability in hardware model checking. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*. 173–181.

[69] Andrew Waterman, Yunsup Lee, Rimas Avizienis, David A. Patterson, and Krste Asanović. 2017. The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.9.1. Retrieved November 17, 2017 from https://riscv.org/specifications/privileged-isa/.

[70] Tjark Weber. 2004. Satisfiability modulo theories. In *Handbook of Satisfiability*. Vol. 185. 825–885. DOI: https://doi.org/10.1145/1995376.1995394

Received January 2018; revised June 2018; accepted September 2018