ILAng: A Modeling and Verification Platform for SoCs using Instruction-Level Abstractions [⋆]

Bo-Yuan Huang, Hongce Zhang, Aarti Gupta, and Sharad Malik

Princeton University
{byhuang,hongcez,aartig,sharad}@princeton.edu

Abstract. We present ILAng, a platform for modeling and verification of systems-on-chip (SoCs) using Instruction-Level Abstractions (ILA). The ILA formal model targeting the hardware-software interface enables a clean separation of concerns between software and hardware through a unified model for heterogeneous processors and accelerators. Top-down it provides a specification for functional verification of hardware, and bottom-up it provides an abstraction for software/hardware co-verification. ILAng provides a programming interface for (i) constructing ILA models (ii) synthesizing ILA models from templates using program synthesis techniques (iii) verifying properties on ILA models (iv) behavioral equivalence checking between different ILA models, and between an ILA specification and an implementation. It also provides for translating models and properties into various languages (e.g., Verilog and SMT LIB2) for different verification settings and use of third-party verification tools. This paper demonstrates selected capabilities of the platform through case studies.

1 Introduction

Modern computing platforms are increasingly heterogeneous, having both programmable processors and application-specific accelerators. These accelerator-rich platforms pose two distinct verification challenges. The first challenge is constructing meaningful specifications for accelerators that can be used to verify the implementation. Higher-level executable models used in early design stages are not suitable for this task. The second challenge is to reason about hardware-software interactions from the viewpoint of software. The traditional approach in system-level/hardware modeling using detailed models, e.g., Register-Transfer Level (RTL) descriptions, is not amenable to scalable formal analysis.

The Instruction-Level Abstraction (ILA) has been proposed to address these challenges [4]. The ILA model is a uniform formal abstraction for processors

[⋆] This work was supported by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA. This research is also funded in part by NSF award number 1628926, XPS: FULL: Hardware Software Abstractions: Addressing Specification and Verification Gaps in Accelerator-Oriented Parallelism, and the DARPA POSH Program Project: Upscale: Scaling up formal tools for POSH Open Source Hardware.

and accelerators that captures their software-visible functionality as a set of instructions. It facilitates behavioral equivalence checking of an implementation against its ILA specification. This, in turn, supports accelerator upgrades using the notion of ILA compatibility similar to that of ISA compatibility for processors [4]. It also enables firmware/hardware co-verification [3]. Further, it enables reasoning about memory consistency models for system-wide properties [8].

In this paper, we present ILAng, a platform for Systems-on-chip (SoC) verification where ILAs are used as the formal model for processors and accelerators. ILAng provides for (i) ILA modeling and synthesis, (ii) ILA model verification, and (iii) behavioral equivalence checking between ILA models, and between an ILA specification and an implementation. The tool is open source and available online on https://github.com/Bo-Yuan-Huang/ILAng.

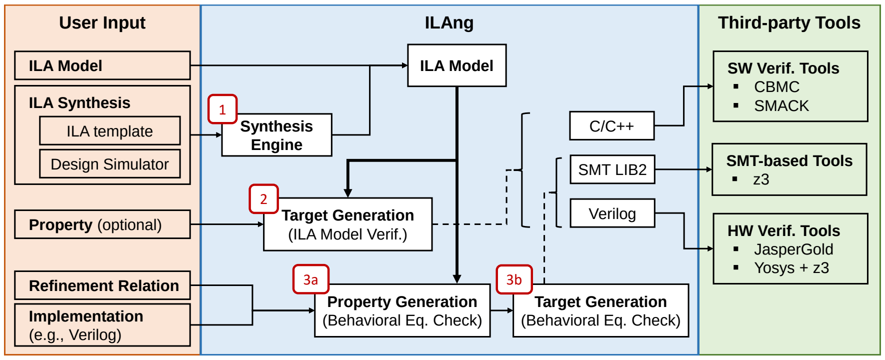

Fig. 1. ILAng Work Flow

2 The ILAng Platform

Figure 1 illustrates the modeling and verification capabilities of ILAng.

1. It allows constructing ILA formal models using a programming interface. It also allows semi-automated ILA synthesis using program synthesis techniques [5] and a template language [6].
2. It provides a set of utilities for ILA model verification, such as SMT-based transition unrolling and bounded model checking. Further, ILAng is capable of translating ILA formal models into various languages for verification, including Verilog, C, C++, and SMT LIB2, targeting other off-the-shelf verification tools and platforms.
3. It supports behavioral equivalence checking between ILA models and between an ILA specification and an implementation based on the standard commutative diagram approach [1].

2.1 Background

ILA The ILA is a formal, uniform, modular, and hierarchical model for modeling both programmable processors and accelerators [4].

An ILA model is **modular** (state updates defined per instruction) and can be viewed as a generalization of the processor ISA in the heterogeneous context, where the commands on an accelerator's interface are defined as instructions. Generally, these appear as memory-mapped IO (MMIO) instructions in programs. The MMIO instruction lacks the actual semantics of what the instruction is doing in the accelerator; the ILA instruction captures exactly that.

It is an operational model that captures updates to *program-visible states* (i.e., states that are accessible or observable by programs). For processors, these include the architectural registers, flag bits, data, and program memory. An `ADD` instruction ($R1 = R2 + R3$), for example, of a processor defines the update to the $R1$ program-visible register. For accelerators, the program visible state includes memory-mapped registers, internal buffers, output ports to on-chip interconnect, etc. An ILA instruction for a crypto-accelerator could define the setting of an encryption key or even the encryption of an entire block using this key.

An ILA model is **hierarchical**, where an instruction at a high level can be expressed as a program of *child-instructions* (like micro-instructions). For example, the `START_ENCRYPT` instruction of a crypto-accelerator can be described as a program of reading data, encrypting it, and writing the result.

2.2 Constructing ILAs

ILAng provides a programming interface to define ILA models. For each component, the program-visible (aka architectural) states are specified. For each instruction, the state updates are specified independently as transition relations. Currently, ILAng supports state variables of type Boolean, bit-vector, and array. Uninterpreted functions are also supported for modeling complex operations.

Synthesis Capability. To alleviate the manual effort in constructing ILAs, ILAng provides a synthesis engine for synthesizing ILAs from a partial description called a template [6] using program synthesis techniques [5]. This is shown as Block 1 in Figure 1. The synthesis algorithm requires two user inputs: an ILA template, and a design simulator/emulator. A template is a partially defined ILA model that specifies the program-visible states, the set of instructions, and also the *possible* operations for each instruction. The simulator can be a software/hardware prototype and is used as an oracle during ILA synthesis. The synthesis algorithm fills in the missing parts in the set of instructions.

2.3 Verification using ILAs

The ILA formal model is a modular (per instruction) transition system, enabling the use of verification techniques such as model checking. We now discuss a selected list of verification capabilities provided by ILAng.

ILA Model Verification. As shown in Block 2 in Figure 1, ILAng supports verification of user-provided safety properties on ILAs. It generates verification targets (including the design and property assertions) into different languages, as discussed in Section 2.4.

Behavioral Equivalence Checking. The modularity and hierarchy in the ILA models simplify behavioral equivalence checking through decomposition. Based on the standard commutative diagram approach [1], behavioral equivalence checking in ILAng supports two main settings: (i) ILA vs. ILA, and (ii) ILA vs. finite state machine (FSM) model (e.g., RTL implementation). As shown in Blocks $3a$ and $3b$ in Figure 1, ILAng takes as input the two models (two ILA models, or an ILA model and an implementation model) and the user-provided refinement relation. The refinement relation specifies:

1. how to apply an instruction (e.g., instruction decode),
2. how to flush the state machine if required (e.g., stalling), and
3. how to identify if an instruction has completed (e.g., commit point).

The refinement map is then used with the two models to generate the property assertions using the standard commutative diagram approach [1]. The verification targets (the design and property assertions) are then generated in Verilog or SMT LIB2 for further reasoning, as described in Section 2.4.

SMT-based Verification Utilities.

Unrolling. Given an ILA, ILAng provides utilities to unroll the transition relation up to a finite bound, with options for different encodings and simplifications. Users can unroll a sequence of given instructions with a fixed instruction ordering. They can also unroll all possible instructions as a monolithic state machine.

Bounded Model Checking (BMC). With the unrolling utility, ILAng supports BMC of safety properties using an SMT solver. Users can specify initial conditions, invariants, and the properties. They can use a fixed bound for BMC or use on-the-fly BMC that iteratively increases the bound.

2.4 Generating Verification Targets

To utilize off-the-shelf verification tools, ILAng can export ILA models into different languages, including Verilog, C, C++, and SMT-LIB2.

Verilog. Many hardware implementations use RTL descriptions in Verilog. To support ILA vs. FSM equivalence checking, ILAng supports exporting ILA to Verilog. This also allows the use of third-party verification tools, since most such tools support Verilog. The memory module in the exported ILA can be configured as internal registers or external memory modules for different requirements.

C/C++. Given an ILA model, ILAng can generate a hardware simulator (in C or C++) for use in system/software development. This simulator can be verified against an implementation to check that it is a reliable executable model.

SMT LIB2. The ILAng platform is integrated with the SMT solver z3 [2]. It can generate SMT formulas for the transition relation and other verification conditions using the utilities described in Section 2.3.

3 Tutorial Case Studies

We demonstrate the applicability of ILAng through two case studies discussed in this section.[1] Table 1 provides information for each case study, including implementation size, ILA (template) size, synthesis time, and verification time (Ubuntu 18.04 VM on Intel i5-8300H with 2 GB memory). Note that these case studies are for demonstration, ILAng is capable of handling designs with significant complexity, as discussed and referenced in Section 4.

Table 1. Experimental Results

	Design Statistics		**ILA**			**Verif.**
	Reference	Ref. Size	# of Insts. (parent/child)	ILA Size	Synth. Time (s)	Time (s)
AES_V	RTL Impl. (Verilog)	1756	8/5	324†	110	63
AES_C	Software Model (C)	328	8/7	292†	63	
Pipe_I	RTL Impl. (Verilog)	218	4/0	78	-	25

† ILA synthesis template. Note: sizes are LoC w.r.t the corresponding language.

3.1 Advanced Encryption Standard (AES)

In this case study, we showcase the synthesis engine (§ 2.2) and the verification utilities (§ 2.3) for ILA vs. ILA behavioral equivalence checking.

We synthesized two ILAs for the AES crypto-engine, AES_C and AES_V, based on C and Verilog implementations respectively. They have the same instruction set, but with differences in block-level and round-level algorithms. As shown in Table 1, the sizes of ILAs (synthesis templates) are *significantly smaller* than the final RTL implementation, making this an attractive entry point for verification.

The equivalence between AES_C and AES_V is checked modularly, i.e., per instruction. Table 1 shows the verification time for checking behavioral equivalence using the SMT solver z3.

3.2 A Simple Instruction Execution Pipeline

In this case study, we demonstrate the steps in manually defining an ILA (§ 2.2) and exporting it in Verilog (§ 2.4) for ILA vs. FSM behavioral equivalence checking using existing hardware verification tools.

[1] All tutorial case studies are available in the submitted artifact and on GitHub.

This pipeline case study is a simple version of the back-end of a pipelined processor. We manually define an ILA model Pipe_I as the specification of the design. This specification can be verified against a detailed RTL implementation, using a given refinement relation. We exported the Verilog model (including Pipe_I and property assertions) and utilized Yosys and z3 for hardware verification. The equivalence is checked modularly per instruction, and took 22 seconds in total for all four instructions, as shown in Table 1.

4 Other ILAng Applications

Firmware/Hardware Co-Verification. The ILA models program-visible hardware behavior while abstracting out lower-level implementation details. This enables scalable firmware/hardware co-verification, as demonstrated in our previous work on security verification of firmware in industrial SoCs using ILAs [3].

Reasoning about Concurrency and Memory Consistency. The ILA model is an operational model that captures program-visible state updates. When integrated with axiomatic memory consistency models that specify orderings between memory operations, the transition relation defined in ILAs (§ 2.3) can be used to reason about concurrent interactions between heterogeneous components [8].

Data Race Checking of GPU Programs. Besides general-purpose processors and accelerators, an ILA model has been synthesized for the nVidia GPU PTX instruction set using the synthesis engine (§ 2.2) [7]. This model has then been used for data race checking for GPU programs using the BMC utility (§ 2.3).

References

1. Burch, J.R., Dill, D.L.: Automated Verification of Pipelined Microprocessor Control. In: CAV. pp. 68–84 (1994)
2. De Moura, L., Bjørner, N.: Z3: An Effcient SMT Solver. In: TACAS (2008)
3. Huang, B.Y., Ray, S., Gupta, A., Fung, J.M., Malik, S.: Formal Security Verification of Concurrent Firmware in SoCs using Instruction-Level Abstraction for Hardware. In: DAC. pp. 1–6 (2018)
4. Huang, B.Y., Zhang, H., Subramanyan, P., Vizel, Y., Gupta, A., Malik, S.: Instruction-Level Abstraction (ILA): A Uniform Specification for System-on-Chip (SoC) Verification. ACM TODAES **24**(1), 10 (2018)
5. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-Guided Component-Based Program Synthesis. In: ICSE. pp. 215–224 (2010)
6. Subramanyan, P., Huang, B.Y., Vizel, Y., Gupta, A., Malik, S.: Template-Based Parameterized Synthesis of Uniform Instruction-Level Abstractions for SoC Verification. IEEE TCAD **37**(8), 1692–1705 (8 2018)
7. Xing, Y., Huang, B.Y., Gupta, A., Malik, S.: A Formal Instruction-Level GPU Model for Scalable Verification. In: ICCAD. pp. 130–135 (2018)
8. Zhang, H., Trippel, C., Manerkar, Y.A., Gupta, A., Martonosi, M., Malik, S.: ILA-MCM: Integrating Memory Consistency Models with Instruction-Level Abstractions for Heterogeneous System-on-Chip Verification. In: FMCAD (2018)