Leveraging Processor Modeling and Verification for General Hardware Modules

Yue Xing, Huaixi Lu, Aarti Gupta, and Sharad Malik
Princeton University, Princeton, USA
{yuex, huaixil, aartig, sharad}@princeton.edu

***Abstract*—For processors, an instruction-set-architecture (ISA) provides a complete functional specification that can be used to formally verify an implementation. There has been recent work in specifying accelerators using formal instruction sets, referred to as Instruction-Level Abstractions (ILAs), and using them to formally verify their implementations by leveraging processor verification techniques. In this paper, we generalize ILAs for specification of general hardware modules and formal verification of their RTL implementations. This includes automated generation of a complete set of functional (not including timing) specification properties using the ILA instructions. We address the challenges posed by this generalization and provide several case studies to demonstrate the applicability of this technique, including all the modules in an open-source 8051 micro-controller. This verification identified three bugs and completed in reasonable time.**

I. Introduction

For processors the **Instruction Set Architecture (ISA)** provides a complete functional specification that can be used for implementation verification. This high-level specification abstracts away low-level implementation details by focusing on the **architectural states**.[1] These are states that are persistent across instructions, i.e., states needed for future instructions. The ISA defines instructions in terms of updates to the architectural states. Formal techniques for verifying processor implementations against ISA specifications have been well-studied [1], [2], and successfully applied, e.g., to automatically verify an ARM core [3]. As the implementation has more detail (e.g., pipelining) than the specification, this verification is not a cycle-by-cycle equivalence check, but rather a **refinement check** that checks the equivalence of the ISA architectural states with their corresponding implementation states at specific times, e.g., at instruction completion.

Recent work [4] has extended the notion of ISA to accelerators which are increasingly used in modern System-on-Chips (SoCs). These accelerators are accessed by software/firmware running on processors using memory-mapped-input-output (MMIO) load/store instructions that provide commands to the accelerators. These commands have been modeled as *accelerator instructions* and the corresponding instruction set is referred as an **Instruction-Level Abstraction (ILA)** of the accelerator. Like ISA instructions, the high-level ILA instructions abstract away the low-level implementation details and model the updates to the accelerator's **architectural states**. For example, an Advanced Encryption Standard (AES) accelerator was modeled as an AES ILA [4] with architectural states `encryption_key`, `address`, other control states and data memory. The AES ILA has instructions `READ/WRITE_KEY` to read/write these control states, and a `START_ENCRYPTION` instruction that performs the encryption operation. The ILA methodology [4], [5] leverages processor verification techniques to formally verify accelerator implementations, where the instruction-by-instruction **refinement checking** provides *key benefits of abstraction and modularity*.

Unlike processors and accelerators, general hardware modules are not thought of as having instruction-like commands at their interface. Thus they are specified in alternate forms. Common techniques include property-based specification [6]–[9] and transaction-level models (TLM) [10]. The property-based specification is typically done with a set of manually crafted properties that the implementation should satisfy. Unlike the ISA/ILA which provide a complete functional specification, it is hard to determine the completeness of the set of properties. *This is a major gap in this form of specification.* TLM uses a set of functions to specify the behavior of different components and is often used in system exploration or hardware-software co-simulation. However, it is hard to identify the equivalence relation between a TLM specification and its hardware implementation, especially when they are designed separately.

We address these gaps in modeling and verifying general hardware modules by extending the ILA methodology to them. We focus on synchronous single clock domain modules as they form the majority of designs. Our goal is to leverage the key ILA benefits: a complete functional specification, and instruction-by-instruction modular verification. However, there are several challenges:

1. Command Interface Identification: The commands to an accelerator can be determined from its MMIO load/store interface. However, general modules may not have an MMIO interface, so their command interfaces need to be identified differently.

2. Multiple Command Interfaces: The MMIO-based accelerators typically have only one interface to accept commands, while general modules can accept multiple commands simultaneously, e.g., the Advanced eXtensible Interface (AXI) module (Fig. 2) simultaneously accepts read and write commands.

3. Shared States between Multiple Command Interfaces: Commands from different interfaces could potentially update the same state in a module, e.g., the 8051 memory interface (Fig. 3) module has separate command interfaces for ROM access and RAM access, and there is a shared state `mem_wait` updated by the commands from both interfaces. This requires careful handling due to potentially conflicting updates.

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.

[1]Consistent with common practice, we use **states** in short to refer to state variables, rather than the valuation of these variables.

We address these challenges and extend the ILA methodology to general modules through a composition of multiple ILAs for modeling multiple *command interfaces*, with potentially shared state. As with processors/accelerators, ILAs enable generation of a *complete set of properties* for verifying functional (not including timing) correctness, and leverages modular instruction-by-instruction verification using refinement checking. Note that there is *no existing methodology for automated generation of a complete set of properties* for functional verification of general modules. Thus, it is not possible to perform a head-to-head comparison with a competing tool. To provide evidence of the value of our proposed methodology, we describe 8 case studies that include all the modules of an open-source 8051 micro-controller, a RISC-V store buffer module [11] and different communication modules from AXI [12] and OpenPiton [13].

Overall, this paper makes the following contributions:

- We extend the ILA methodology to general modules through categorizing them into three classes: (i) single command interface, (ii) multiple command interfaces without shared states, and (iii) multiple command interfaces with shared states. To our knowledge, this is the first work to provide instruction-level formal functional models for general hardware modules.
- We demonstrate the use of these specification models in complete functional verification of an RTL implementation of a general hardware module, using instruction-by-instruction refinement checking.
- We demonstrate eight case studies covering a range of modules from micro-controller components to communication fabrics. *In particular, we cover all the modules from an open-source 8051 micro-controller – demonstrating the general strength of the methodology.*

II. ILA Background

The Instruction-Level Abstraction (ILA) [4] has been recently proposed for modeling MMIO-accessed accelerators at the instruction-level, similar to the ISA for processors. Effectively each MMIO Load/Store serves as a command, i.e., an "instruction," at the accelerator interface and updates its architectural states. That work [4] also defined **sub-instructions** as the visible steps in an instruction execution, i.e., a sub-instruction is a step that results in a state-update that is visible in the architectural state. [2] Note that *each sub-instruction is atomic, i.e., it executes as a unit.* For example, an instruction with a two-byte result that appears one byte at a time at the output has *two* sub-instructions, one for each of the two steps that outputs a byte. Each (sub-)instruction specifies how it updates the architectural states. This instruction-by-instruction specification is then used for modular verification of the accelerator implementation via refinement checking [5], similar to processors [1], [2]. Formally, an ILA model is a five-element tuple $\langle S, W, S_0, D, N \rangle$, where S and W denote the sets of states and inputs, respectively, and S_0 denotes the set of initial state values. The set of instructions I is defined by sets D and N, which represent the decode functions (specifying the triggered instruction) and the next state functions (describing the state update) for each instruction $i \in I$, respectively.

[2]Micro-instructions may also specify the steps in an instruction, but unlike sub-instructions, their results are not visible in the architectural state.

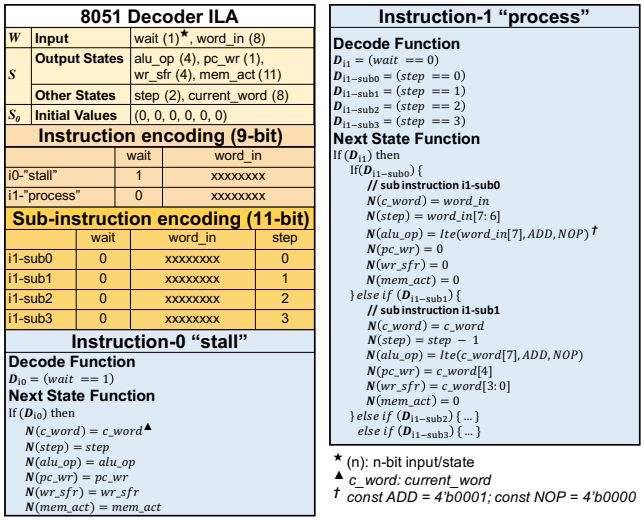

Fig. 1: 8051 decoder ILA (sketch)

III. ILA Modeling for General Modules

Based on the ILA modeling challenges for general modules (§I), we classify general modules as one of three types below.

A. Single-Command Interface Module

In the simplest case, a module has a single-command interface consisting of its input pins. The instructions are the different valid values presented at this interface. For example, Fig. 1 shows the decoder module of an 8051 micro-controller [14]. Its inputs include a one-bit pin `wait` to halt the module's execution, and an 8-bit pin `word_in` for the word to be decoded. (The clock and reset pins are abstracted, since they are general to all modules.) There are two instructions in the decoder module: `stall` and `process`, as shown in Fig. 1.

1) ILA States: The architectural states include output states such as `alu_op`, `pc_wr`, etc. There are also two non-output architectural states – `current_word` to store the word that is being decoded, and `step` to indicate the sub-instruction step in the multiple-step decoding process. These states are persistent across sub-instructions and are shown as "other states" in Fig. 1.

2) ILA Instructions: Each (sub-)instruction $i \in I$ is defined by specifying the command bit-pattern for triggering the instruction, i.e., its decode function D_i, and its state update function N_i. We use the following operational semantics – when D_i evaluates to true in the current state, then state update function N_i is applied to update the state. As shown in Fig. 1, the `stall` instruction is triggered when the input command satisfies its decode function: `(wait == 1)`. Its state update function leaves all states unchanged. The `process` instruction is triggered when the input command satisfies: `(wait == 0)`. The architectural states are updated for each of its sub-instructions as follows: when `step` is 0 (indicating that the previous multi-step word has finished), the `current_word`, `step`, and output states are updated according to the input `word_in`; when `step` is 1 (indicating the last step of the current multi-step word), `current_word` is not changed, `step` is decreased by one, and the output states (`alu_op` etc.) are updated according to `current_word`. State updates for the other two sub-instructions (when `step` is 2, 3) are defined similarly and omitted due to space limitations.

READ-PORT-ILA		
W_{READ}	**Input**	rd_addr_valid, rd_addr_in, rd_length_in, rd_data_ready
S_{READ}	**Output States**	rd_addr_ready, rd_data, rd_data_valid
	Other States	tx_rd_active, tx_rd_addr, tx_rd_length
I	**Instruction Name**	**Updated States**
i0	RD_ADDR_WAIT	rd_addr_ready
i1	RD_ADDR_COMMIT	rd_addr_ready, tx_rd_active, tx_rd_addr, tx_rd_length
i1-s0	RD_DATA_PREPARE	rd_data, rd_data_valid
i1-s1	RD_DATA_COMMIT	tx_rd_addr, rd_addr_ready, tx_rd_active, tx_rd_length

WRITE-PORT-ILA		
W_{WRITE}	**Input**	wr_addr_valid, wr_addr_in, wr_length_in, wr_data_in, wr_data_valid
S_{WRITE}	**Output States**	wr_addr_ready, wr_data_ready
	Other States	tx_wr_active, tx_wr_addr, tx_wr_length
I	**Instruction Name**	**Updated States**
i0	WR_ADDR_WAIT	wr_addr_ready
i1	WR_ADDR_COMMIT	wr_addr_ready, tx_wr_active, tx_wr_addr, tx_wr_length
i1-s0	WR_DATA_PREPARE	wr_data_ready
i1-s1	WR_DATA_COMMIT	wr_data_ready, tx_wr_addr, tx_wr_active, tx_wr_length
i1-s2	WR_LAST_RESPONSE	wr_addr_ready

Fig. 2: AXI slave ILA (sketch)

3) "0"-Command Interface Modules: Modules that do not have an explicit command interface, e.g., transaction initiators and clock generators, are a special case. These modules produce outputs with no dependence on inputs, e.g., a clock generator produces an output clock signal without receiving an input command. Such modules are started by an implicit "power-on" input in hardware. Thus, we define a single "start" instruction that is triggered by a "power-on" input to model these modules.

B. Multiple-Command Interface Module

We now extend the ILA model to multiple-command interfaces, e.g., the AXI slave [12] with separate interfaces for read and write requests. An intuitive approach is to use a set of ILAs, one for the read interface, and one for the write interface. Then a *composition* of these *interface ILAs* serves as the formal specification of the module (described in §III-C).

The first step is to identify the different command interfaces for the module. We refer to a set of pins that form a single interface as the **port** for that interface. For example, in Fig. 2, the input pins for controlling the module's read (write) functionality are grouped as the `READ-port` (`WRITE-port`). The second step is to define the ILA states and instructions for each port, referred to as the **port-ILA**.

The `READ-port-ILA` and `WRITE-port-ILA` are sketched in Fig. 2. During module operation, each port is first configured and it then processes the read/write request. Thus, the architectural states include the output states (such as `rd_data`) and the non-output states ("other states" such as `tx_rd_addr`) for configuration. The `READ-port-ILA` has two instructions (`i0`, `i1`) and two sub-instructions (`i1-s0`, `i1-s1`) that wait for and commit the read operation. The `WRITE-port-ILA` has two instructions (`i0`, `i1`) and three sub-instructions (`i1-s0`, `i1-s1`, `i1-s2`) that wait for and commit the write operation. Fig. 2 shows the states updated by a (sub-)instruction (other details are omitted; github link for full models provided in §V).

C. Composition of Port ILAs

The **port-ILAs** are composed to derive the **module ILA** as follows.

1) Port ILA Composition without Shared States: For the AXI slave module, the two ports are independent with separate sets of inputs and states. In this case, the composition is the union of their ILAs, i.e., module-ILA: [`READ-port-ILA`, `WRITE-port-ILA`]. Each port-ILA accepts its command, decodes it, and updates its architectural states independently.

2) Port ILA Composition with Shared States: When the port-ILAs have shared state, they are not independent. Consider the

ROM-PORT-ILA (port-1)		
W_{p1}	**Input**	rom_data_in, rom_addr_in rom_data_valid,
	Output States	rom_addr, rom_data
S_{p1}	**Other States**	***mem_wait***
I_{p1}	**Instr. Name**	**Updated States**
$i0_{p1}$	ROM_REQ	rom_addr, ***mem_wait***
$i1_{p1}$	ROM_RESP	rom_data
$i2_{p1}$	ROM_IDLE	***mem_wait***

RAM-PORT-ILA (port-2)		
W_{p2}	**Input**	ram_addr_in, ram_data_valid, ram_data_in
	Output States	ram_addr, ram_data
S_{p2}	**Other States**	***mem_wait***
I_{p2}	**Instr. Name**	**Updated States**
$i0_{p2}$	RAM_REQ	ram_addr, ram_data, ***mem_wait***
$i1_{p2}$	RAM_RESP	ram_data
$i2_{p2}$	RAM_IDLE	***mem_wait***

integrate

Integrated ROM-RAM-PORT-ILA		
W_c	**Input**	input(ROM-PORT) ∪ input(RAM-PORT)
S_c	**Output States**	output_states(ROM-PORT) ∪ output_states(RAM-PORT)
	Other States	other_states(ROM-PORT) ∪ other_states(RAM-PORT)
I_c	**Instr. Name**	**Updated States**
($i0_{p1}$, $i0_{p2}$)	ROM_REQ & RAM_REQ	rom_addr, ***mem_wait,*** ram_addr, ram_data
($i0_{p1}$, $i1_{p2}$)	ROM_REQ & RAM_RESP	rom_addr, ***mem_wait,*** ram_data
($i0_{p1}$, $i2_{p2}$)	ROM_REQ & RAM_IDLE	rom_addr, ***mem_wait***
($i1_{p1}$, $i0_{p2}$)	ROM_RESP & RAM_REQ	rom_data, ram_addr, ram_data, ***mem_wait***
($i1_{p1}$, $i1_{p2}$)	ROM_RESP & RAM_RESP	rom_data, ram_data
($i1_{p1}$, $i2_{p2}$)	ROM_RESP & RAM_IDLE	rom_data, ***mem_wait***
($i2_{p1}$, $i0_{p2}$)	ROM_IDLE & RAM_REQ	***mem_wait,*** ram_addr, ram_data
($i2_{p1}$, $i1_{p2}$)	ROM_IDLE & RAM_RESP	***mem_wait,*** ram_data
($i2_{p1}$, $i2_{p2}$)	ROM_IDLE & RAM_IDLE	***mem_wait***

(a) Integrated ROM-RAM-port-ILA

PC-PORT-ILA		
W_{pc}	**Input**	pc_jmp, pc_target, instr_in
S_{pc}	**Output States**	imm_data0, imm_data1, operand0, operand1
	Other States	pc, instr_buff
I_{pc}	**Instruction Name**	**Updated States**
i0	LOAD_INST	instr_buff
i1	PC_UPDATE	pc, imm_data0, imm_data1, operand0, operand1
i2	PC_KEEP	imm_data0, imm_data1, operand0, operand1

(b) PC-port-ILA

Fig. 3: 8051 memory interface ILA (sketch)

ILA model for the 8051 memory interface module as shown in Fig. 3. It has three ports, and two of them – the `ROM-port` and the `RAM-port` (Fig. 3a) – have a shared state `mem_wait`. The third port `PC-port` is independent of the other two.

The architectural states of `ROM-port-ILA` and `RAM-port-ILA` include the output states, such as the address (`rom_addr`, `ram_addr`) and data (`rom_data`, `ram_data`). The non-output shared architectural state `mem_wait` indicates an ongoing memory access. The `ROM-port-ILA` and `RAM-port-ILA` each have three instructions, for requesting the memory, receiving the response, and staying idle. Further, they both have instructions that update the `mem_wait` state. Specifically, the `ROM_REQ` and `RAM_REQ` instructions update `mem_wait` to `1`, and the `ROM_IDLE` and `RAM_IDLE` instructions update `mem_wait` to `0`. Note that if `RAM_REQ` and `ROM_IDLE` are *both* triggered, then state updates to `mem_wait` would be conflicting. The informal specification [14] of the memory interface module clarifies that when both ports update `mem_wait`, an update to value `1` has higher priority than update to value `0`. Thus, `mem_wait` should be updated to `1` by the `RAM_REQ` instruction. To capture this interaction of instructions from different port-ILAs, *we integrate the port-ILAs that share states into a single port-ILA*, e.g., the integrated `ROM-RAM-port-ILA` shown on the bottom of Fig. 3(a). For the integrated ILA, its input pins (W_c) and state variables (S_c) are the union of the input pins (W_{p1}, W_{p2}) and state variables (S_{p1}, S_{p2}) of the port-ILAs, respectively. Formally, $W_c = W_{p1} \cup W_{p2}$ and $S_c = S_{p1} \cup S_{p2}$. Its instruction set is a cross-product of the instruction sets of the two ports, i.e., $I_c = I_{p1} \times I_{p2}$. *Note that this cross-product is taken at the sub-instruction level, since that is the atomic unit for each instruction.* This ensures that each possible combination of steps in the different port instructions is considered. Each instruction/sub-instruction in the integrated port is triggered only when its two component instructions/sub-instructions are

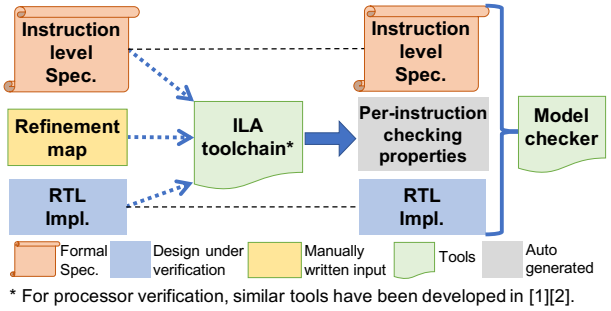

Fig. 4: ILA Verification Flow

triggered in their respective ports, i.e., $D_{c,i} = D_{p1,i_j} \wedge D_{p2,i_k}$ for $i \in I_c$ and $i = (i_j, i_k)$.

In the integrated port, the non-shared states are updated according to the update functions of the individual ports, and the shared states are updated according to how the informal specification resolves the conflict using priorities. *Note that if the informal specification does not resolve the conflict, this is flagged as a specification gap by our methodology.* For example, for the `ROM_REQ & RAM_RESP` instruction, state `rom_addr` is updated as in the `ROM-port`, since there are no conflicting updates in the `RAM_RESP` instruction. In instructions `ROM_IDLE & RAM_REQ`, the shared state `mem_wait` is updated to `1` due to priority as per the specification document.

Integrating port-ILAs as a single port-ILA results in more instructions than in the original ports (9 instructions in `ROM-RAM-port` vs. a sum of 6 instructions in `ROM-port` and `RAM-port`), but it resolves the conflicting updates as per the informal specification. Finally, the `PC-port` (Fig. 3(b)) is modeled for controlling the program counter (`pc`). It is independent of the other two ports, leading finally to the resulting ILA: [`ROM-RAM-port-ILA`, `PC-port-ILA`].

D. ILA Modeling for General Modules: Summary

Step 1 Group the input pins into different input ports. Each input port corresponds to a set of pins that provide a command to the module.
Step 2 Identify architectural states, instructions for each port.
Step 3 Integrate the port-ILAs that share states.
Step 4 The union of independent port-ILAs is the module-ILA.

IV. Verifying General Modules

In this section, we show how a module ILA is used for verification of its RTL implementation. Like the ILA methodology with accelerators [4], [5], we leverage processor verification techniques [1], [2], and briefly review them first.

A. Processor/Accelerator Verification with ILA Specifications

Fig. 4 shows the flow of verifying an RTL implementation against its instruction-level specification (ISA for processors, ILA for accelerators). Since the instruction-level specification is at a higher-level than the implementation, this verification is a refinement check and not a cycle-by-cycle sequential equivalence check. The refinement check uses a *refinement map*, typically provided by a user, to connect the specification and the implementation. It defines: (i) the implementation states that correspond to the architectural states to be checked for equivalence, and (ii) when to check this equivalence, e.g., after an instruction commits. Given an ISA/ILA, an RTL design, and a refinement map, a set of correctness properties can be generated automatically using prior techniques [1], [2]. Each property checks one specified instruction modularly, and has the following form – *Starting from corresponding states that are equivalent, after executing the specified instruction, the corresponding states will be equivalent*. A standard model checker [15] is used to automatically check the set of properties. *Thus, this methodology provides automation in generation of properties for complete functional verification, since properties are checked for each instruction in the specification. Our methodology fully leverages these benefits for general modules.*

B. General Module Verification using Module-ILAs

The verification of a given RTL implementation of a general module against its module-ILA specification similarly follows the flow in Fig. 4. For modules with multiple ports, after integrating the dependent ports, the resulting ports are independent. Thus, each independent port is verified using its port-ILA.

As an example, Fig. 5 shows a refinement map for verifying the 8051 decoder. It contains three parts: state map, interface map, and instruction map. The state map provides the correspondence between the ILA architectural states and the RTL registers. The interface map relates the inputs, so that when a command is presented to the ILA, the corresponding inputs will also be presented to the RTL. The instruction map specifies the start and finish condition of each instruction in the RTL. For example, for the `stall` instruction, the start condition is its decode function, and the finish condition (i.e., when to check) is: after one clock cycle.

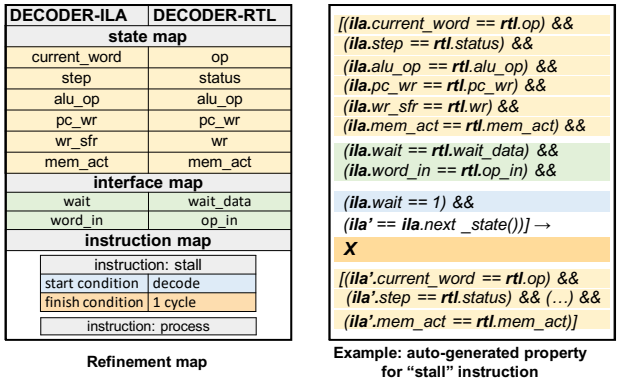

Fig. 5: Refinement Map for 8051 Decoder (sketch)

Given a refinement map, the properties for checking all instructions are generated automatically by our tool. The right side of Fig. 5 shows an example property for the `stall` instruction. It checks the following: When the ILA model and RTL implementation execute the instruction starting from corresponding equivalent states (ILA executes the instruction by applying the next_state function), under the start condition (decode function), then when the RTL finishes the instruction (here, the symbol $\rightarrow$ denotes logic implication and the temporal logic operator ***X*** denotes the next-cycle in the RTL) the ILA's architectural states (denoted as ila') should be equivalent to the corresponding RTL states. The background color shows the association between each part of the property and the given refinement map – yellow for equivalent states, green for corresponding inputs, and blue/orange for the start/finish condition of the instruction in RTL. Similar properties are automatically generated for each instruction and are formally verified using a model checker.

V. Case Studies

As mentioned earlier (§ I), there is no existing methodology for automated generation of a complete set of properties for functional verification of general hardware modules. Therefore, a direct comparison with a competing tool or approach is not possible. Instead, we provide evidence of the value of ILA-based modeling and verification of general modules through eight case studies. [3] Three of them are the decoder, memory interface, and datapath - covering all the modules of an open-source 8051 micro-controller [14]. The others are the AXI master and slave [12], an L2 cache and an NoC router from OpenPiton [13], and a store buffer from a RISC-V core [11]. As with most open-source hardware, these eight designs come only with informal text descriptions/specifications – we used these to create the ILA models.

The open-source ILAng platform [5] was used to model each port-ILA and JasperGold [16] was used as the model checker in verification. All the experiments were performed on a Dell Server with a 2.3 GHz 28-core Intel Haswell processor and 224 GB of RAM, running a RedHat Linux 5 OS. The statistics of the RTL designs and ILA models are summarized in Table I. For verification, the column "Ref-map Size" shows the size of the refinement map (described in Json format), which is much smaller than the ILA model and the RTL implementation. The column "Time (bug)" presents the time to find bugs (if any), and after fixing them the verification time ("Time") and maximum memory ("Memory Usage") are also reported.

The models for 8051 decoder, memory interface and AXI slave have been discussed in §III and thus omitted below.

A. Single-Port Modules

1) 8051 Decoder Module: Its verification completes quickly since most of the design code realizes a single map from the input `word` values to the output values.

B. Multi-Port Modules without Shared States

1) AXI Slave: A bug is identified in the `READ-port`: the update function for the output state `rd_data` uses the current value in the architectural states `tx_rd_burst`. However the implementation incorrectly uses the input `rd_burst_in`. A counter-example trace for this bug is found in 0.01s, and the design is verified in 0.11s after fixing that bug.

2) AXI Master: The AXI master receives requests from a host, translates them into AXI protocol form and interacts with the AXI slave. It also maintains two separate ports `READ-port` and `WRITE-port`. They are modeled similarly to the two ports of the slave module. The master module is verified within 1s.

3) 8051 Datapath: The datapath has two independent ports, an arithmetic logic unit port (`ALU-port`) and a data access (`data-port`) port. The `ALU-port` models 16 instructions for different computations like `add, sub`, etc. The `data-port` is for accessing an internal RAM and special functional registers (SFRs) and has 4 instructions. The verification takes 176s to finish, largely because of the 256 byte internal RAM. When abstracted as a 16-byte memory (standard small memory modeling), the verification time is reduced to 9.5s.

4) L2 Cache: The L2 cache module sits between an L1.5 cache and an NoC (network-on-chip) [13]. It has dual parallel pipelines, modeled as two independent ILA ports. The `PIPE1-port` has two instructions that handle the load and store miss from L1.5 cache. The `PIPE2-port` has six instructions that cover all six types of messages from the NoC.

A bug was found while verifying `PIPE1-port`: there is a typo in the informal document that a pipeline register `msg_flag_2` is used, whereas `msg_flag_3` is needed. In our verification, a counter-example trace exposing this bug is found in 0.7s. After fixing the typo, it is verified in 20 minutes. This case study demonstrates that our methodology scales to large modules (more than 10k LoC in RTL) in practical designs.

C. Multi-Port Modules with Shared States

1) Memory Interface: As discussed in §III, it demonstrates the integration of dependent ports and is verified in 1s.

2) Store Buffer in RISC-V core: The store buffer [11] has three command interfaces: `in-port` and `out-port` control an array of buffered stores, and a `load-port` loads the buffered store back to the processor pipeline. The `in-port` and `out-port` share some flag states and are integrated as an `in-out-port`. The `load-port` is independent. Verification identified the following bug in 0.61s: the update to the flag register is incorrect when there is traffic on both `in-port` and `out-port` and the buffer is full. After fixing it, verification proved correctness in 78s (reduced to 1.3s by abstracting the 64 byte memory as a 16 byte memory).

3) NoC Router: The last case study is an OpenPiton NoC router [13], which is used for data movement among cores. This router module is connected in the four directions with other routers and with a processor core. Each connection `X` has an `IN-port-X` to receive the incoming packets and an `OUT-port-X` to send the outgoing packets, leading to a total of ten ports. The module has a dynamic routing table state, which can be updated by all five `IN-port-X`s. We integrate these five ports and resolve conflicting updates (with a round-robin algorithm according to the specification [13]). Similarly, all five `OUT-port-X`s are also integrated. Finally, we get a single `IN-port` and a single `OUT-port`, each with 32 instructions. The RTL implementation is shown equivalent to the ILA model and this verification takes around 700s.

VI. Related Work

Property specification language (PSL) [7], [8] and similarly SystemVerilog Assertions (SVA) [9] have been widely used as temporal logic based specifications in formal verification. Users specify a set of property assertions (e.g., signal A always rises before signal B), which can be verified using model checking. However, evaluating the completeness of these property specifications is difficult. This limits their value as a *complete* functional specification. Several definitions of "completeness" have been explored such as unambiguously determining the output trace [17], [18]. However, these often lead to a property specification that overly constrain the design behavior. In contrast, our proposed general module ILA provides a functional model for each command at its ports. Just like a processor ISA, the completeness of an ILA model is achieved by including all "instructions" from the command interface into the model.

Our refinement check verifies the functional equivalence of the ILA model and RTL using safety properties. We can extend our method to use other techniques to check liveness properties in RTL implementations [2] or convert liveness to safety verification [19].

[3] Source code is available on https://github.com/yuex1994/DATE2021

TABLE I: Case Studies – Statistics of RTL Designs, ILA Models, Refinement Maps and Verification Time/Memory

Design	Design Statistics		ILA Model Statistics				Verification			
	RTL Size (LoC)	# of RTL State Bits	# of ports	# of insts. (all ports)	ILA Size (LoC)	# of Arch. State Bits	Ref-map Size (LoC)	Time (bug)(s)	Time (s)	Memory Usage (MB)
Decoder	2636	30	1	5	479	30	53	-	0.21	32.9
AXI Slave	828	372	2	9	167	159	77	0.01	0.11	7.8
AXI Master	871	403	2	11	184	289	109	-	0.23	9.7
Datapath	2987	273*	2	20	861	229*	119	-	9.5 (176)	667 (2830)
L2 Cache	10924	2844*	2	8	596	340*	272	0.7	1214	2270
Mem. Interface	1096	304	3/2†	12	342	220	86	-	0.74	44.4
Store Buffer	399	93*	3/2†	6	148	45*	47	0.6	1.3 (78)	189 (243)
NoC Router	5495	1522	10/2†	64	394	465	198	-	691	3920

* Not including memory block ; † Before/after integrating ports that share states

An abstracted FSM or a high-level state machine (HLSM) is often used in hardware design [20], [21]. As with the ISA/ILA, this is a high-level functional specification and abstracts away low-level implementation details. However, unlike ISA/ILA, it has no notion of an instruction. This leads to non-modular verification when checking an RTL implementation, where establishing a monolithic refinement/equivalence relation between the two models is more difficult.

Transaction-level modeling [10] using SystemC has seen significant use in early-stage design exploration, to model system components as functional units and their communication as pre-defined abstract channels. However, the TLM model and the RTL implementation are developed separately, making it difficult to verify the implementation's correctness against the TLM. High-Level Synthesis (HLS) tools generate an RTL implementation from a high-level model, and can verify their equivalence in certain situations, e.g., with synthesis metadata [22], [23], or with certain HLS transformations [24] that rely on structural similarity. These approaches are hard to apply on non-HLS RTL designs. Similarly Verilog designs synthesized from BlueSpec (BSV) [25] have been verified against the BSV models in that restricted context. In contrast, we do not depend on structural similarity and our verification is modular, offering the benefits of improved scalability.

Instruction-level model has also been used for specific processor components. For instance, [26] models the Intel i7 processor's executor engine module at the instruction-level, with the ISA/microISA opcodes as the input instructions. This is straightforward since these modules have the same set of instructions at their interface just like the processor ISA. In the contrast, our method generalizes the instruction level models for general hardware modules with arbitrary command interfaces.

VII. Conclusions

In this paper, we generalized the notion of instruction-level abstraction (ILA) to model and verify general hardware modules. This leverages processor verification techniques with significant advantages: (i) automated generation of a complete set of verification properties for functional verification, and (ii) modular instruction-by-instruction verification. We discussed the challenges associated with this generalization and provided a classification of designs into three categories to address them. We showed eight modeling and verification case studies from these three categories (including all the modules from an 8051 micro-controller design). We found bugs in three cases and completed formal verification in all case studies in reasonable time, demonstrating the efficacy of the proposed ILA methodology for general modules.

References

[1] J. R. Burch and D. L. Dill, "Automatic verification of pipelined microprocessor control," in *CAV*, 1994, pp. 68–80.

[2] P. Manolios and S. K. Srinivasan, "A complete compositional reasoning framework for the efficient verification of pipelined machines," in *ICCAD*, 2005, pp. 863–870.

[3] A. Reid, R. Chen, A. Deligiannis, D. Gilday, D. Hoyes, W. Keen, A. Pathirane, O. Shepherd, P. Vrabel, and A. Zaidi, "End-to-end verification of ARM® processors with ISA-formal," in *CAV*, 2016, pp. 42–58.

[4] B.-Y. Huang, H. Zhang, P. Subramanyan, Y. Vizel, A. Gupta, and S. Malik, "Instruction-Level Abstraction (ILA): A Uniform Specification for System-on-Chip (SoC) Verification," *TODAES*, pp. 1–24, 2018.

[5] B. Y. Huang, H. Zhang, A. Gupta, and S. Malik, "Ilang: a modeling and verification platform for SoCs using instruction-level abstractions," in *TACAS*, 2019, pp. 351–357.

[6] C. Eisner and D. Fisman, *A practical introduction to PSL*. Springer Science & Business Media, 2007.

[7] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M. Y. Vardi, and Y. Zbar, "The ForSpec temporal logic: A new temporal property-specification language," in *TACAS*, 2002, pp. 296–311.

[8] F. Xie and H. Liu, "Unified property specification for hardware/software co-verification," in *COMPSAC*, 2007, pp. 483–490.

[9] E. Cerny, S. Dudani, J. Havlicek, and D. Korchemny, *SVA: the power of assertions in SystemVerilog*, 2015.

[10] L. Cai and D. Gajski, "Transaction level modeling: an overview," in *CODES+ISSS*, 2003, pp. 19–24.

[11] Arch Lab. in Tokyo Institute of Technology, "RISC-V Dynamic Execution CORE," 2014, [Online]. Available:https://github.com/ridecore/ridecore, accessed on: 2020-08-29.

[12] A. Olofsson, R. Trogan, F. Huettig, O. Jeppsson, and P. Saunderson, "Epiphany eLink AXI," 2016, [Online]. Available:https://github.com/parallella/oh/tree/master/src/axi, accessed on: 2020-08-29.

[13] J. Balkind, M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, A. Lavrov, M. Shahrad, A. Fuchs, S. Payne, and X. Liang, "OpenPiton: An open source manycore research framework," *ACM SIGPLAN Notices*, pp. 217–232, 2016.

[14] S. Teran and S. Jaka, "8051 micro controller," 2016, [Online]. Available: http://opencores.org/project,8051, accessed on: 2020-08-29.

[15] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model checking*. MIT press, 2018.

[16] Cadence Design Systems, Inc., "JasperGold: Formal Property Verification App." 2018, [Online]. Available:http://www.jasper-da.com/products/jaspergold-apps/, accessed on: 2020-08-29.

[17] K. Claessen, "A coverage analysis for safety property lists," in *Formal Methods in Computer Aided Design (FMCAD'07)*. IEEE, 2007, pp. 139–145.

[18] D. Große, U. Kuhne, and R. Drechsler, "Estimating functional coverage in bounded model checking," in *2007 Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 2007, pp. 1–6.

[19] A. Biere, C. Artho, and V. Schuppan, "Liveness checking as safety checking," *Electron. Notes Theor. Comput. Sci.*, pp. 160–177, 2002.

[20] A. Kuehlmann and R. A. Bergamaschi, "High-level state machine specification and synthesis," in *ICCD*, 1992, pp. 536–539.

[21] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of computer programming*, pp. 231–274, 1987.

[22] P. Urard, A. Maalej, R. Guizzetti, N. Chawla, and V. Krishnaswamy, "Leveraging sequential equivalence checking to enable system-level to RTL flows," in *DAC*, 2008, pp. 816–821.

[23] A. Mathur, M. Fujita, E. Clarke, and P. Urard, "Functional equivalence verification tools in high-level synthesis flows," *Design and Test of Computers*, pp. 88–95, 2009.

[24] S. Kundu, S. Lerner, and R. K. Gupta, "Translation validation of high-level synthesis," *TCAD*, pp. 566–579, 2010.

[25] R. Nikhil, "Bluespec System Verilog: efficient, correct RTL from high level specifications," in *MEMOCODE*. IEEE, 2004, pp. 69–70.

[26] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, and E. Reeber, "Replacing Testing with Formal Verification in Intel ® CoreTM i7 Processor Execution Engine Validation," in *CAV*, 2009, pp. 414–429.