Hardware-Software Interface Specification for Verification in Accelerator-Rich Platforms

Sharad Malik
Princeton University
LATTE ’21, April 15, 2021

This work was supported in part by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA; by the DARPA POSH and SSITH programs; and by NSF XPS Grant No. 1628926.
ILA Team + Collaborators

Sharad Malik  Aarti Gupta  Margaret Martonosi  Pramod Subramanyan  Bo-Yuan Huang

Hongce Zhang  Yue Xing  Yi Li  Huaixi Lu  Yu Zeng

Yakir Vizel  Weikun Yang  Grigory Fedyukovich  Caroline Trippel  Yatin A. Manerkar
Hardware-Software Interface

Software

Abstraction for software

Instruction-Set Architecture

Hardware

Specification for hardware

Single-core uniprocessor

Pentium® III Processor
Source: pdcfaculty.org
Hardware-Software Interface

Instruction-Set Architecture + Memory Consistency Model

ISA + MCM

Software

Abstraction for software

Hardware

Specification for hardware

Multi-core uniprocessor

Intel Skylake

Source: wikichip.org
Hardware-Software Interface

Apple M1 Die Photo
Source: AnandTech

https://www.anandtech.com/show/16226/apple-silicon-m1-a14-deep-dive
Accelerator Interface

// Load instruction

```
1 uint32_t status = *ADDR_STATUS; // mmio read
2 if (((status >> 8) == INIT)
3    for(int i=0; i<KEY_SIZE; i++)
4        *(ADDR_KEY+i) = KEY[i]; // mmio write
5    status |= 1; // set lock bit
6    *ADDR_STATUS = status; // mmio write & lock
7    *ADDR_ENABLE = 1; // mmio write & enable
```

// Store instruction

Firmware C code

```
Accelerator

<table>
<thead>
<tr>
<th>Address</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xff00</td>
<td>status</td>
</tr>
<tr>
<td>0xff02</td>
<td>enable</td>
</tr>
</tbody>
</table>

Accessing registers

Triggering operations

```

On-chip Interconnect

CPU

GPU

Flash

DMA

MMU+DRAM

HW accelerators

Microcontroller + Firmware

Memory

NoC interface

// Load instruction

// Store instruction
SoC Verification

- Accelerator implementation verification
  - Formal
  - Simulation-based

- Firmware hardware co-verification
  - Formal
  - Simulation-based

- Shared memory accesses and memory consistency
Instruction-Level Abstraction (ILA)

Merits (similar to ISA)
- Software-visible “architectural” state variables
- Modular: set of instructions
  - Per-instruction state-update

More than ISA
- Formal
- Hierarchical
- Generalizes ISA to include accelerators

Interface Commands \text{def} Instructions

<table>
<thead>
<tr>
<th>MMIO accesses</th>
<th>Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write, 0xff00, 0x1</td>
<td>START_ENCRYPT</td>
</tr>
<tr>
<td>Write, 0xff02, data</td>
<td>WRITE_ADDRESS</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

AES Block Encryption

Interconnect

Visible State
- Address
- Length
- Key
- Counter
- State
Instruction-Level Abstraction (ILA)

(a) Accelerator RTL implementation snippet.

```
module Top_rtl (
    clk, rst, interrupt,
    if_axi_rd_r_msg, if_axi_rd_r_rdy,
    if_axi_wr_w_msg, if_axi_wr_w_rdy
    // other AXI interface ports
);

assign and_192 = and_6& (fsm_out[2]);
assign or_117 = (fsm_out[2]) | (fsm_out[5]);

always @(posedge clk or negedge rst_bar) begin
    if (~rst_bar) begin
        addrBound_4_1_equal <= 1'b0;
    end
    else if (run_w_wen & (~while_case_0)) begin
        addrBound_4_1_equal <= addrBound_1_1_tmp_7;
    end
end
```

(b) Accelerator instruction set (partial). Each HW operation triggered by an MMIO access is modeled as an individual instruction.

```
{ 
    WR 0x33000100 val (SetWeightAddr val),
    WR 0x33000120 val (SetTensorSize val),
    WR 0x33000130 val (SetLSTMConfig val),
    WR 0x3300200 0x1 (StartLSTM), ...
}
```

(c) A sequence of accelerator instructions performing an LSTM layer.

```
SetWeightAddr 0xff100100
SetDataAddr 0xff100200
SetOutputAddr 0xff100300
SetTensorSize 0x20
SetNumTimestep 0x10
SetLSTMConfig 0x02cd87f9
StartLSTM
```

FlexASR Accelerator (Speech Recognition)
Harvard Wei Group
SoC Verification

- Accelerator implementation verification
  - Formal
    - Simulation-based

- Firmware hardware co-verification
  - Formal
  - Simulation-based

- Shared memory accesses and memory consistency
Application of ILA: Hardware Verification

- Formal verification of RTL implementation

Processor or accelerator RTL

ILA specification

Add: $r_d = r_{s1} + r_{s2}$
$pc += 4, \ldots$

Jump:
$pc = r_{s1} + \text{imm}, \ldots$

SetLength:
if !busy: length = axi_wdata

SetKey:
if !busy: key = axi_wdata

(for processors or accelerators)

Refinement Mapping

```
always @(posedge clk) begin
  if ((resetn) begin
    mem_le_firstword_reg <= 0;
    last_mem_valid <= 0;
  end else begin
    if (!last_mem_valid)
      mem_le_firstword_reg <= mem_le_fir;
    last_mem_valid <= mem_valid & !mem_ready;
  end
end
```

```
“state_mapping”: {
  “aes_address” : “RTL.aes_reg_opaddr_i.reg_out”,
  “aes_length” : “RTL.aes_reg_oplen_i.reg_out”,
  ... 
}

“instructions”: [{
  “instruction” : “WR_ADDR”,
  “ready_signal” : “RTL.xram_ack_delay_1”,
  “max_bound” : 20 }, ...]
```
Application of ILA: Hardware Verification (cont’d)

• Formal verification of RTL implementation
  • Auto-generate complete formal properties for each instruction

ILA specification
Instruction i: \( S' = F(S) \)

- **State S**
- **Instruction i**
- **Assume:** match
- **Implementation**
- **Symbolically execute the instruction**
- **State S’**
- **Check:** match?
- **Instruction i**
Application of ILA: Hardware Verification (cont’d)

• Formal verification of RTL implementation
  • Modular verification — per-instruction checking
  • Automating modular verification

For each instruction, check:
• in all valid RTL starting states (environment)
Application of ILA: Hardware Verification (cont’d)

• Formal verification of RTL implementation
  • Modular verification — per-instruction checking
  • Automating modular verification

For each instruction, check:
• in all valid RTL starting states (environment)
• Environment constraint auto-generation
Application of ILA: Hardware Verification (cont’d)

- Formal verification of RTL implementation

<table>
<thead>
<tr>
<th>Category</th>
<th>Designs</th>
<th>Design Size</th>
<th>ILA Size</th>
<th>RTL size vs. ILA size (ratio of LoC)</th>
<th>Proof Coverage (in 1 hour on double Xeon 5222 + 256GB RAM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processors</td>
<td>8051 micro-controller</td>
<td>8.9k</td>
<td>0.6k (2.6k)</td>
<td>0.6k</td>
<td>256</td>
</tr>
<tr>
<td></td>
<td>Rocket</td>
<td>18.2k</td>
<td>4.6k</td>
<td>0.7k</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>Piccolo</td>
<td>11.4k</td>
<td>7.8k (79.2k)</td>
<td>0.7k</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>Nibbler</td>
<td>2.7k</td>
<td>2.7k</td>
<td>0.6k</td>
<td>15</td>
</tr>
<tr>
<td>Accelerators</td>
<td>AES encryption</td>
<td>1.1k</td>
<td>7.5k</td>
<td>0.5k</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>RBM</td>
<td>10.6k</td>
<td>6.3k (184.6k)</td>
<td>1.0k</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>Gaussian blur image processing</td>
<td>6.9k</td>
<td>36.8k</td>
<td>0.5k</td>
<td>2</td>
</tr>
<tr>
<td>Memory system</td>
<td>OpenPiton L1.5 cache</td>
<td>12.0k</td>
<td>5.9k (104.7k)</td>
<td>1.5k</td>
<td>19</td>
</tr>
<tr>
<td>Communication interfaces</td>
<td>LeWiz Ethernet MAC core (TX)</td>
<td>12.5k</td>
<td>12.9k (428.7k)</td>
<td>2.3k</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>EMesh AXI interfaces</td>
<td>0.9k</td>
<td>0.45k</td>
<td>0.5k</td>
<td>22</td>
</tr>
</tbody>
</table>
SoC Verification

- Accelerator implementation verification
  - Formal
  - Simulation-based

- Firmware hardware co-verification
  - Formal
  - Simulation-based

- Shared memory accesses and memory consistency
Application of ILA: Hardware Verification (cont’d)

- Simulation-based Validation
  - ILA supports auto-generation of simulation model

ILA Spec. → ILAtor → Instruction level executable model (ILEM)
RTL Impl. → Existing RTL Compiler* → RTL executable model (RTEM)

Tool’s input
Generated Executable
ILA toolchain
e.g. Verilator
Application of ILA: Hardware Verification (cont’d)

- Simulation-based validation (traditional)
  - Activate the RTL Execution Model (RTEM) with test stimuli (a sequence of RTL inputs)
  - After the simulation, check if the RTL signal/register value matches the reference result

```
res = a/b = 3;
```

```
a = 6;  
b = 2;  
res = a/b = 3;
```

```
res = 3;
```

```
res = 2;
```

```
res = 3;
```

**Deficiency:** need to run and analyze the full test regardless of when bug is triggered – instead, should stop when bug is triggered
Application of ILA: Hardware Verification (cont’d)

- Simulation-based validation (tandem simulation)
  - After simulating each instruction, check if the RTEM Architectural Variables (RTAV) match ILEM Architectural Variables (ILAV)

Identify bugs right at the instruction that causes AV deviations
SoC Verification

- Accelerator implementation verification
  - Formal
  - Simulation-based

- Firmware hardware co-verification
  - Formal
  - Simulation-based

- Shared memory accesses and memory consistency
FW/HW co-verification in SoCs

- Communicating (heterogeneous) IPs
  - Processor
  - Firmware
  - Specialized accelerators

- Hard to design, hard to verify
  - FW/HW co-design
  - Heterogeneity
  - Security critical
Challenges

• FW/HW interaction
  • Hardware functions not captured
  • RTL models not practical (too complex)
  • HW/SW semantics gap

• Scalability
  • Bit-precise reasoning
  • Concurrency
    • message handling, synchronization

ILA for Hardware +
SW Verification Tech.

```
1  uint32_t status = *ADDR_STATUS;  // mmio read
2  if (((status >> 8) == INIT)
3      for(int i=0; i<KEY_SIZE; i++)
4          *(ADDR_KEY+i) = KEY[i];  // mmio write
5    status |= 1;                   // set lock bit
6  *ADDR_STATUS = status;         // mmio write & lock
7  *ADDR_ENABLE = 1;              // mmio write & enable
```
Co-verification methodology

- Modeling
  - ILA for specialized HW
  - Source-level (LLVM) modeling of SW

- Verification
  - Software verification techniques

![Diagram](Diagram.png)
SoC Verification

- Accelerator implementation verification
  - Formal
  - Simulation-based

- Firmware hardware co-verification
  - Formal
  - Simulation-based

- Shared memory accesses and memory consistency
Hardware-Software Co-Verification (cont’d)

• Simulation-based
  • hardware-software co-simulation using ILA as verified abstraction

ILA Model ➔ ILAtor ➔ Instruction-level simulator

Formally-verified abstraction

Verilog design ➔ Low speed simulation

High speed simulation

HW/SW co-simulation (QEMU)
SoC Verification

- Accelerator implementation verification
  - Formal
  - Simulation-based

- Firmware hardware co-verification
  - Formal
  - Simulation-based

- Shared memory accesses and memory consistency
Hardware-Software Co-Verification with MCM

- Shared memory accesses – memory consistency issues

Driver  Device  Crypto Accelerator

Shared Memory

System-wide Verification
Hardware-Software Co-Verification with MCM (cont’d)

- Shared memory accesses – memory consistency issues
  - Memory consistency issues: memory access reordering

Example:

Thread 1 (T1):
1:   st  [x], 1
2:   ld  r1, [y]

Thread 2 (T2):
1':  st  [y], 1
2':  ld  r2, [x]

r1 == 0 and r2 == 0

Observable under Total Store Order (TSO: adopted by x86)

Reordering visible to the other thread
Hardware-Software Co-Verification with MCM (cont’d)

- Shared memory accesses – memory consistency issues
  - ILA-MCM framework: security and correctness verification using ILA and MCM

![Diagram of hardware-software co-verification with MCM]

- Property
- System-wide Verification w. MCM
- State Update Values
- State Update Orderings
Hardware-Software Co-Verification with MCM (cont’d)

• Shared memory accesses – memory consistency issues
  • ILA: operational, state update functions (values)
  • MCM: axiomatic & relational, state update re-ordering effects (ordering)

• Integrating ILA and MCM
  • Facet and facet event
  • Captures different entities’ different observations

Shared (memory) variables

\[ x \text{ (observed by } T1) \]
\[ x \text{.T1} \]

\[ x \text{ (observed by } T2) \]
\[ x \text{.T2} \]

Observed by some entity
Denoted as \(<\text{variable.entity}>\)
Shared memory accesses – memory consistency issues
- ILA: operational, state update functions (values)
- MCM: axiomatic & relational, state update re-ordering effects (ordering)

Integrating ILA and MCM
- Facet and facet event
- Captures different entities’ different observations

Integrate values and orderings

Hardware-Software Co-Verification with MCM (cont’d)
Hardware-Software Co-Verification with MCM (cont’d)

- Verifying existing hardware-interacting programs

- Synthesize **new** programs (malicious exploits, buggy trace and etc…)

  - Program sketch
  
  - Set of ILA instructions
  - Partial ordering over instances of instructions (optional)

<table>
<thead>
<tr>
<th>Driver</th>
<th>Device</th>
<th>Crypto-Acc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>IO_WRITE 0x100, 1 (SendResetCmd)</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>memcpy FwAddr,SM? (WriteImage)</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>IO_WRITE 0x104, 1 (SendLdFwCmd)</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Hardware-Software Co-Verification with MCM (cont’d)

And example of the security exploit found by our verification:

- Okay in SC
- But TOCTOU exists **under** TSO!

Set the lock of device to prevent changes to FW image

**Core problem:**
The overwritten could be visible late due to relaxed MCM, thus it circumvents the checking

Overwritten is still allowed, but visible later
Summary: Hardware-Software Interface

Apple M1 Die Photo
Source: AnandTech

https://www.anandtech.com/show/16226/apple-silicon-m1-a14-deep-dive
Summary: ILA Based SoC Verification

- Accelerator implementation verification
  - Formal
  - Simulation-based

- Firmware hardware co-verification
  - Formal
  - Simulation-based

- Shared memory accesses and memory consistency
GitHub: https://github.com/PrincetonUniversity/ILAng
Wiki: https://bo-yuan-huang.gitbook.io/ilang/
Docker: https://hub.docker.com/r/byhuang/ilang/
Selected Bibliography

- ILAng: A Modeling and Verification Platform for SoCs using Instruction-Level Abstractions. [TACAS19]
- Integrating Memory Consistency Models with Instruction-Level Abstractions for Heterogeneous System-on-Chip Verification. [FMCAD18]
- Formal Security Verification of Concurrent Firmware in SoCs using Instruction-Level Abstraction for Hardware. [DAC18]
- Instruction-Level Abstraction (ILA): A Uniform Specification for System-on-Chip (SoC) Verification. [TODAES18] (Best Paper Award)