Generalizing the ISA to the ILA: A Software/Hardware Interface for Accelerator-rich Platforms

Sharad Malik
Princeton University
DAC 2023
July 11, 2023

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 Verification Team + Collaborators

Sharad Malik  Aarti Gupta  Margaret Martonosi  Pramod Subramanyan  Bo-Yuan Huang  Jason Fung
Hongce Zhang  Yue Xing  Yi Li  Huaixi Lu  Yu Zeng  Sayak Ray
Yakir Vizel  Weikun Yang  Grigory Fedyukovich  Caroline Trippel  Yatin A. Manerkar
Architecture of the IBM System/360

Abstract: The architecture of the newly announced IBM System/360 features four innovations:

1. An approach to storage which permits and exploits very large memories, hierarchies of speeds, read-only storage for microprogram control, flexible storage protection, and simple program relocation.
2. An integrated System/360 programming system-Phase I, which provides a common programming environment for systems of various sizes and covers the fundamental aspects of the language and data processing needs.
3. The introduction of a new family of data processing systems, the IBM System/360, based on a single processor, the IBM 360 Model 30. The IBM 360 is a single-processor system designed to accommodate all levels of data processing from small to large. The architecture of the IBM System/360 is designed to provide a high degree of flexibility and adaptability to meet the needs of various users and applications.
4. An instruction-set architecture providing a high degree of flexibility and adaptability to meet the needs of various users and applications.

This paper discusses in detail the objectives of the design and the rationale for the major features of the architecture. Key points are given to the problems raised by the need for compatibility among central processing units of various sizes and by the conflicting demands of commercial, scientific, real-time, and logical information processing. A tabular summary of the architecture is shown in the Appendix.

Introduction

The design philosophy of the new general-purpose machine architecture for the IBM System/360 is discussed in this paper. In addition to showing the architecture of the new family of data processing systems, we point out the various engineering problems encountered in attempts to make the system design compatible, as the program bit level, and large and small models. The compatibility was aimed not only to models of any size but also to their various applications—scientific, commercial, real-time, and so on.

The section that follows describes the objectives of the new system design, i.e., that it serve as a base for new techniques and applications, that it be general-purpose, efficient, and simple program compatible in all models. The remainder of the paper is devoted to the design problems faced, the alternatives considered, and the decisions made for data format, data and instruction codes, storage assignments, and input/output controls.

Design objectives

The new architecture builds upon but differs from the designs that have gradually evolved since 1950. The evolution of the computer had included, besides major technological improvements, several important system concepts and developments.
Hardware-Software Interface

Intel Skylake
Source: wikichip.org

Software
Abstraction for software

Hardware
Specification for hardware

ISA + MCM
Instruction-Set Architecture + Memory Consistency Model

Chip Multiprocessor
Hardware-Software Interface

Software

???

Abstraction for software

Hardware

Specification for hardware

Heterogenous System-on-Chip

Apple M1 Die Photo
Source: AnandTech

https://www.anandtech.com/show/1622/6/apple-silicon-m1-a14-deep-dive

Specialized hardware units – aka accelerators
Software/firmware accessed/invoked
Accelerator Interface

Can we convert MMIO loads/stores to meaningful instruction-level semantics?

Firmware C code

```c
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
```

// Load instruction

// Store instruction

MMIO Instructions have load/store semantics
Opaque to hardware accelerator semantics
Instruction-Level Abstraction (ILA)

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

- Abstraction of the HW as seen at the interface
  - A disciplined lifting of the RTL to the level of software

Generalizes the ISA to include processors and accelerators

<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>
</tbody>
</table>
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 0x33000200 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
always @(posedge clk) begin
  if (!resetn) begin
    mem_la_firstword_reg <= 0;
    last_mem_valid <= 0;
  end else begin
    if (!mem_valid)
      mem_la_reg <= mem_la_firstword;
    lastm_valid <= mem_valid && !mem_ready;
  end
end
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 Map**

**What to compare**

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

**When to compare**

```
“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: $S' = F(S)$

Symbolically execute the instruction
Use standard property checkers

Contrast with ad-hoc set of properties
always @(posedge clk) begin
    if (!resetn) begin
        mem_la_firstword_reg <= 0;
        last_mem_valid <= 0;
    end else begin
        if (!mem_valid)
            mem_la_reg <= mem_la_firstword;
        lastm_valid <= mem_valid && !mem_ready;
    end
end
Application of ILA: Hardware Verification (cont’d)

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

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

Legend:
- Tool’s input
- Generated Executable
- ILA toolchain
- e.g. Verilator
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

Automated - Generalized to Processors + Accelerators
always @ (posedge clk) begin
    if (!resetn) begin
        mem_la_firstword_reg <= 0;
        last_mem_valid <= 0;
    end else begin
        if (!mem_valid)
            mem_la_reg <= mem_la_firstword;
        lastm_valid <= mem_valid && !mem_ready;
    end
end
FW/HW co-verification in SoCs

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

- FW/HW interaction
  - Hardware functions not captured
  - RTL models not practical (too complex)
  - SW/HW level of abstraction gap

```c
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
always @(posedge clk) begin
    if (!resetn) begin
        mem_la_firstword_reg <= 0;
        last_mem_valid <= 0;
    end else begin
        if (!mem_valid)
            mem_la_reg <= mem_la_firstword;
        lastm_valid <= mem_valid && !mem_ready;
    end
end
Hardware-Software Co-Verification (cont’d)

- Simulation-based
  - hardware-software co-simulation using ILA as verified abstraction
Summary: ILA Based SoC Verification

- Accelerator implementation verification
  - Formal
  - Simulation-based

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

- Shared memory accesses and memory consistency

ILA-MCM: Memory Consistency Models for Acclerator-rich SoC Platforms
ILA-based Compilation for Accelerators

1. Provide compiler IR-accelerator mapping by using ILA as a verified lifting.
2. Verify the mapping correctness.
3. Pattern match the compiler IR pattern provided in the mapping.
4. Rewrite compute graph and lower to the MMIO accesses during code generation.

Applications provided in DSLs

Compiler IR (compute graph)

Verified mapping using ILA
Pattern matching & code generation

Programs exploiting custom accelerators

Heterogeneous backends

- CPU instructions
  - CMP r0, r1
  - SUBGT r0, r0, r1
  - BNE loop

- Invoke accel 1 (MMIO)
  - STR r2, 0xffff0000
  - LDR r3, 0xffff0010

- Invoke accel 2 (MMIO)
  - STR r4, 0xffff0100
  - LDR r5, 0xffff0110

- CPU instructions
  - MOV r3, r2
  - SUBGT r0, r0, r1
  - B lr

Accel 1 func A:
- STR r2, 0xffff0000
- LDR r3, 0xffff0010

Accel 1 func B:
- STR r2, 0xffffaa00
- LDR r3, 0xffffaaab

Accel 2 func C:
- STR r4, 0xffff0100
- LDR r5, 0xffff0110

Simulation-based
Proof-based

General purpose
- VTA
- ARM
- x86

Application specific
- RISC-V
- FlexASR
- HLSCNN
- NVDLA

Relay
Compiler Team (Princeton, UW, Harvard)

Bo-Yuan Huang  Thierry Tambe  Yi Li  Mike He  Gus Smith

Gu-Yeon Wei  Aarti Gupta  Sharad Malik  Zachary Tatlock
Selected Bibliography

Primary Papers

- Generalizing Tandem Simulation: Connecting High-level and RTL Simulation Models [ASPDAC 21]
- 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)

Additional Papers

- Leveraging Processor Modeling and Verification for General Hardware Modules. [DATE21] (Best Paper Award)
- Automatic Generation of Architecture-Level Models from RTL Designs for Processors and Accelerators [DATE 22]