Automate and Accelerate RISC-V Verification by Compositional Formal Methods

Yean-Ru Chen, Cheng-Ting Kao, Yi-Chun Kao, Tien-Yin Cheng, Chun-Sheng Ke and Chia-Hao Hsu

Department of Electrical Engineering, National Cheng Kung University, Tainan City, Taiwan (R.O.C)
Outline

• Introduction
• Application
• Experimental results
• Conclusions
Outline

• Introduction
• Application
• Experimental results
• Conclusions
CPU Verification

• Many optimizations applied to improve performance
  – Pipelining, forwarding, out-of-order execution, etc.

• Simulation is hard to cover the whole functionality of processor
  – Hard to think of all corner cases and harder to simulate all corner cases

• Formal verification technique has became a trend

CAGR: Compound annual growth rate

Source: The 2016 Wilson Research Group ASIC/IC and FPGA Functional Verification Study
ARM ISA-Formal

• End-to-end framework to detect bugs in the datapath, pipeline control and forwarding/stall logic of processors by model checking

• Explore all the legal different sequences of instructions and able to detect the defects mentioned above

riscv-formal

• Framework for formal end-to-end verification of RISC-V cores against the ISA specification

• Propose a RISC-V Formal Interface (RVFI) for riscv-formal

Deficiencies of Related Works

• ARM ISA formal
  – CSR instruction
  – State-space explosion problem
  – Coverage information

• riscv-formal
  – Need to pre-set checking depths
  – Environment calls/breakpoints instructions
  – Check the read/write contents of CSR but not for CSR instructions
  – Properties are manually created
  – state-space explosion problem
  – coverage information
Outline

• Introduction
• Application
• Experimental results
• Conclusions
Proposed Workflow

- Specification
  - Machine readable specification
  - SystemVerilog property (extended RVFI)
  - SVA with split properties
  - Model checker (JasperGold)

- RISC-V based CPU
  - Extended RVFI wrapped CPU
  - Signal mapping model

Result

© Accellera Systems Initiative
Extend RVFI for Property Auto-generation

- Automatically generate RV32I formal properties based on well-qualified machine readable specification
- Extend RVFI for more functions

```vhdl
rvfi_de_insn = vscale_pipeline.inst_DX;
rvfi_insn2 <= rvfi_de_insn;
rvfi_insn3 <= rvfi_insn2;
rvfi_rs1_rdata = vscale_regfile.data[rvfi_rs1_addr];
rvfi_rs2_rdata = vscale_regfile.data[rvfi_rs2_addr];
rvfi_rd_wdata = vscale_regfile.data[rvfi_rd_addr];
```
Extend RVFI for Property Auto-generation (cont’d)

- Original SVA property for checking *add* instruction

```verilog
class property [31:0] gold_add;
property ori_add;
@ (posedge clk) disable iff (reset)
addtrigger
|$= (gold_add == rvfi_rd_wdata);
endproperty
property_ori_add: assert property (ori_add);

always_ff @(posedge clk) begin
  gold_add <= rvfi_rs1_rdata + rvfi_rs2_rdata;
end
```
Verification Space Abstraction by Property Splitting

• Original SVA property for the *add* instruction is able to verify two targets:
  1. Checks the correctness of the data forwarding (red)
  2. Checks if the actual result data is correctly written back to the destination register (blue)
Split SVA Properties for Checking \textit{add} Instruction

1. Checks the correctness of the data forwarding

2. Checks if the actual result data is correctly written back to the destination register

```verilog
logic [31:0] wb_data_pipe;

always_ff @(posedge clk) begin
  wb_data <= vscale_regfile.wd;
end

property rd_wb_test;
@ (posedge clk ) disable iff (reset)
addtrigger
|$=>
(wb_data == rvfi_rd_wdata);
endproperty

property_rd_wb_test: assert property (rd_wb_test);

logic [31:0] gold_add;
property fwding_add;
@ (posedge clk ) disable iff (reset)
addtrigger
|$=>
(gold_add == wb_data);
endproperty

property_fwding_add: assert property (fwding_add);

always_ff @(posedge clk) begin
  gold_add <= rvfi_rsl_rdata + rvfi_rs2_rdata;
end
```
Compositional Formal Verification Method

• Assume-guarantee reasoning

\[
\frac{M \parallel A \models P \quad N \models A}{M \parallel N \models P}
\]

Premise

Conclusion

- M and N: Components
- A: Assumption
- P: Property
- \parallel: Composite
- \models: Satisfy

M = checking datapath of writing data to correct destination register
N = checking datapath of computing the correct write back data
P = checking whether the correct data is calculated and sent to the correct destination register
A = assumption describing that the correctness of data forwarding is assumed valid
Outline

• Introduction
• Application
• Experimental results
• Conclusions
Design under verification

- **Vscale**
  - 32-bit 3-stage single-issue pipeline CPU

- **RV12**
  - 32/64-bit 6-stage single-issue pipeline CPU

Verification environment:
- Server running CentOS 6.10, which has 48 cores with 2.20 GHz CPU and 256 GB memory embedded
- Using Cadence JasperGold 2018.03
Comparison

• Take Vscale for example

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>Abstract</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property name</td>
<td><code>original_add</code></td>
<td><code>forward_add</code></td>
</tr>
<tr>
<td>Result</td>
<td>Pass</td>
<td>Pass</td>
</tr>
<tr>
<td>Time (sec)</td>
<td>75140.4</td>
<td>1187.3</td>
</tr>
<tr>
<td>COI coverage of pipeline module</td>
<td>93.13%</td>
<td>93.13%</td>
</tr>
<tr>
<td>Proof core coverage of pipeline module</td>
<td>48.98%</td>
<td>60.60%</td>
</tr>
</tbody>
</table>

• **Cone-of-influence (COI) coverage**: Determines the cover items in the Cone-of-influence of each assert
• **Proof core coverage**: Represents the portion of the design verified by formal engines
## Results of Vscale ISA formal verification

<table>
<thead>
<tr>
<th>Instruction type</th>
<th>Number of properties</th>
<th>Execution time (second)</th>
<th>Verification result</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>22</td>
<td>16167.3</td>
<td>PASS (except <em>sra</em> instruction)</td>
</tr>
<tr>
<td>I-type</td>
<td>18</td>
<td>23482.0</td>
<td>PASS (except <em>srai</em> instruction)</td>
</tr>
<tr>
<td>B-type</td>
<td>12</td>
<td>1624.2</td>
<td>PASS</td>
</tr>
<tr>
<td>J-type</td>
<td>8</td>
<td>15.8</td>
<td>PASS (except <em>jalr</em> instruction)</td>
</tr>
<tr>
<td>L-type</td>
<td>12</td>
<td>38.2</td>
<td>PASS</td>
</tr>
<tr>
<td>S-type</td>
<td>8</td>
<td>39.3</td>
<td>PASS</td>
</tr>
<tr>
<td>U-type</td>
<td>4</td>
<td>29.8</td>
<td>PASS</td>
</tr>
<tr>
<td>Assumption</td>
<td>4</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>Total properties</td>
<td>88</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>
Number of inconclusive instruction properties in RV12

<table>
<thead>
<tr>
<th>Instruction type</th>
<th># of properties (inconclusive/total) (without abstraction)</th>
<th># of properties (inconclusive/total) (with abstraction)</th>
<th>Improvements (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-TYPE</td>
<td>10/10</td>
<td>5/10</td>
<td>50.0%</td>
</tr>
<tr>
<td>I-TYPE</td>
<td>9/9</td>
<td>3/9</td>
<td>66.6%</td>
</tr>
<tr>
<td>J-TYPE</td>
<td>2/4</td>
<td>1/4</td>
<td>25.0%</td>
</tr>
</tbody>
</table>
Coverage information

• **Vscale**
  – Top module:
    • COI coverage : 92.80 %
    • Proof core coverage : 76.96 %
  – Core module:
    • COI coverage : 92.87 %
    • Proof core coverage : 75.92 %
    
• **RV12**
  – Top module:
    • COI coverage : 67.93 %
    • Proof core coverage : 61.17 %
  – Core module:
    • COI coverage : 87.85 %
    • Proof core coverage : 83.33 %
    
waive → 98.18 %
waive → 89.11 %
waive → 96.11 %
waive → 91.57 %
Defects found by our verification flow

- **Vscale:**
  - `sra` and `srai`
  - `jalr`

- **RV12:**
  - `csrrwi`
Error in \textit{sra} and \textit{srai} instructions (Vs scale)

- "Arithmetic right shifts" operator should be ">>>>", while they are implemented as ">>" which is the logical right shift operator.

\begin{verbatim}
module vscale_alu(
    input [`ALU_OP_WIDTH-1:0] op,
    input [`XPR_LEN-1:0] in1,
    input [`XPR_LEN-1:0] in2,
    output reg [`XPR_LEN-1:0] out);

    wire [`SHAMT_WIDTH-1:0] shamt;

    assign shamt = in2[`SHAMT_WIDTH-1:0];

    always @* begin
        case (op)
            `ALU_OP_ADD : out = in1 + in2;
            `ALU_OP_SLL : out = in1 << shamt;
            `ALU_OP_XOR : out = in1 ^ in2;
            `ALU_OP_OR : out = in1 | in2;
            `ALU_OP_AND : out = in1 & in2;
            `ALU_OP_SRL : out = in1 >> shamt;
            `ALU_OP_SEQ : out = {31'b0, in1 == in2};
            `ALU_OP_SNE : out = {31'b0, in1 != in2};
            `ALU_OP_SUB : out = in1 - in2;
            `ALU_OP_SRA : out = $signed(in1) >> shamt;
    endcase // case op
end
\end{verbatim}
Error in \textit{jalr} instruction (Vscale)

• Vscale directly sets the lowest bit of the immediate value to be 0 and then adding to rs1, which is different from RISC-V specification requirements

◆ Part of Vscale PC mux implementation

```verilog
wire [\texttt{XPR\_LEN}-1:0]
jalr_offset = { {23:inst\_DX[31]}, inst\_DX[30:21], \texttt{1'b0} ;

always @(*) begin
  case (PC\_src\_sel)
    `PC\_JAL\_TARGET : begin
      base = PC\_DX;
      offset = jalr\_offset;
    end
    `PC\_JALR\_TARGET : begin
      base = rs1\_data;
      offset = jalr\_offset;
    end
  end

assign PC\_PIF = base + offset;
```
Error in “csrrwi” instructions (RV12)

- **csrrs, csrrc, csrrsi** and **csrrci**
  - Have to concern whether source register is \( x0 \)

- **csrrw** and **csrrwi**
  - Shouldn’t concern whether source register is \( x0 \)

- Part of RV12 decode stage implementation

```haskell
//system
{1'b?,CSRRW}: illegal_alu_instr = illegal_csr_rd | illegal_csr_wr ;
{1'b?,CSRRS}: illegal_alu_instr = illegal_csr_rd | (!if_src1 & illegal_csr_wr);
{1'b?,CSRRC}: illegal_alu_instr = illegal_csr_rd | (!if_src1 & illegal_csr_wr);
{1'b?,CSRRI}: illegal_alu_instr = illegal_csr_rd | (!if_src1 & illegal_csr_wr);
{1'b?,CSRRWI}: illegal_alu_instr = illegal_csr_rd | (!if_src1 & illegal_csr_wr);
{1'b?,CSRRSI}: illegal_alu_instr = illegal_csr_rd | (!if_src1 & illegal_csr_wr);
{1'b?,CSRRCI}: illegal_alu_instr = illegal_csr_rd | (!if_src1 & illegal_csr_wr);

default: illegal_alu_instr = 1'b1;
```
Outline

• Introduction
• Application
• Experimental results
• Conclusions
Conclusions

• Propose a verification flow to automatically generate the formal properties for RISC-V RV32I instructions

• The properties are reliable by coverage analysis information
  – Proof coverage can average about 90% in core module after waive unconcerned module

• Using abstraction technique to mitigate state-space explosion problem

• Defect the faults in our experimental CPU
  – \textit{sra}, \textit{srai} and \textit{jalr} instructions in Vscale case
  – \textit{csrrwi} instruction in RV12 case
Thank you for your attention!

Q&A

Author Contact Information: cheny@ncku.edu.tw