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)

accellera SYSTEMS INITIATIVE

2019 DESIGN AND VERIFICATION DVCON CONFERENCE AND EXHIBITION EUROPE

Outline

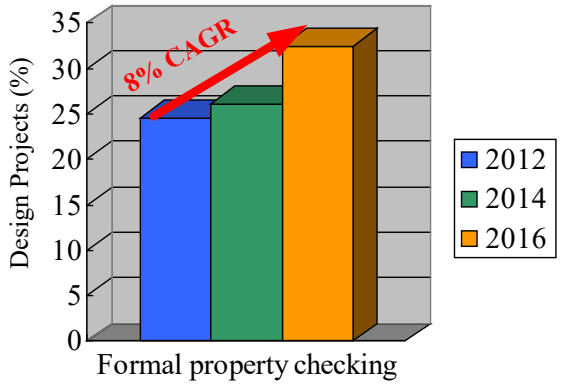
- Introduction
- Application
- Experimental results
- Conclusions

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

Cite : Alastair Reid et al., "End-to-end verification of processors with ISA-formal," International Conference on Computer Aided Verification, 2016.

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

Cite : Clifford Wolf. RISC-V Formal Verification Framework. https://github.com/cliffordwolf/riscv-formal, 2016.

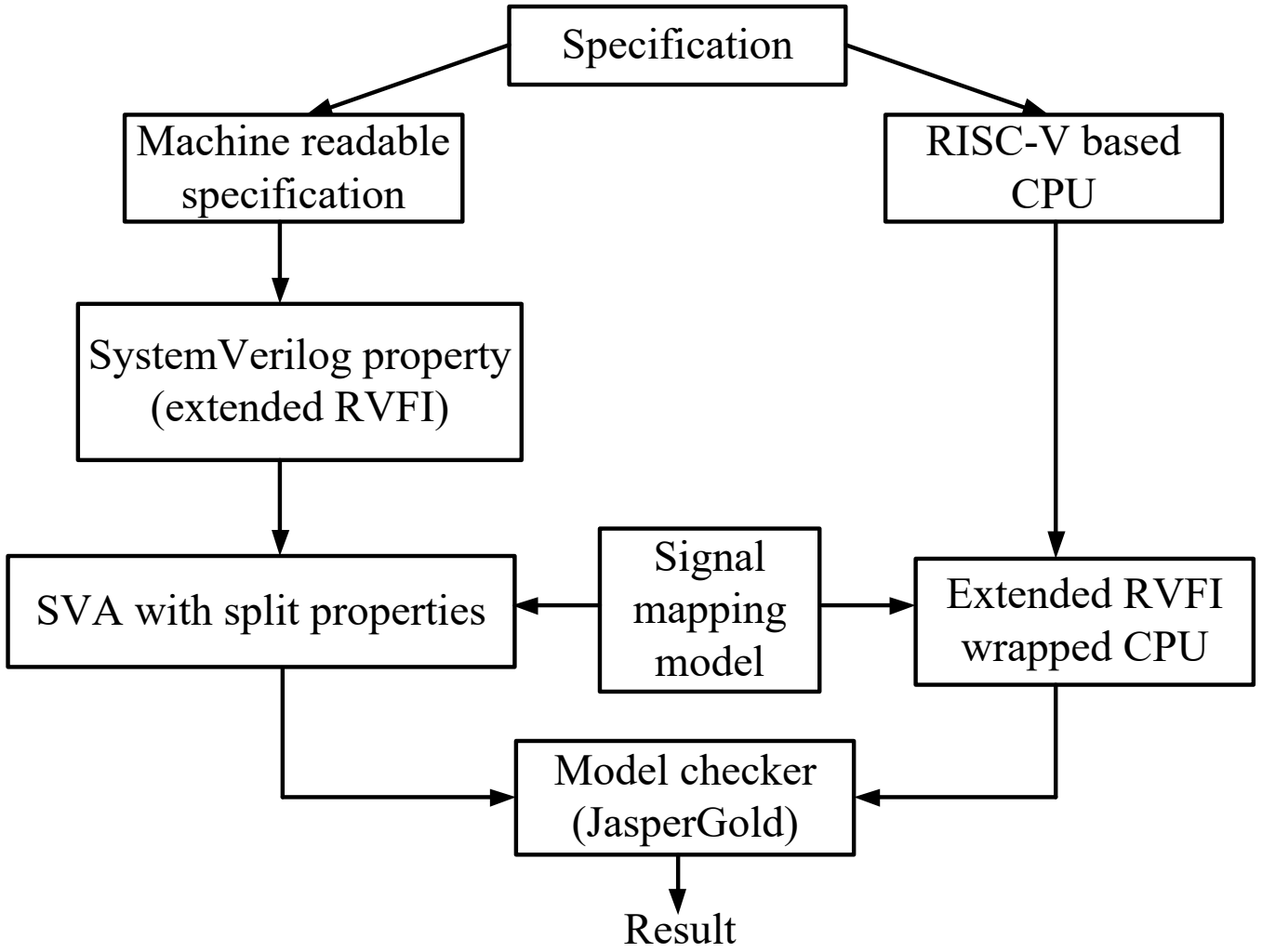
Deficiencies of Related Works

- 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

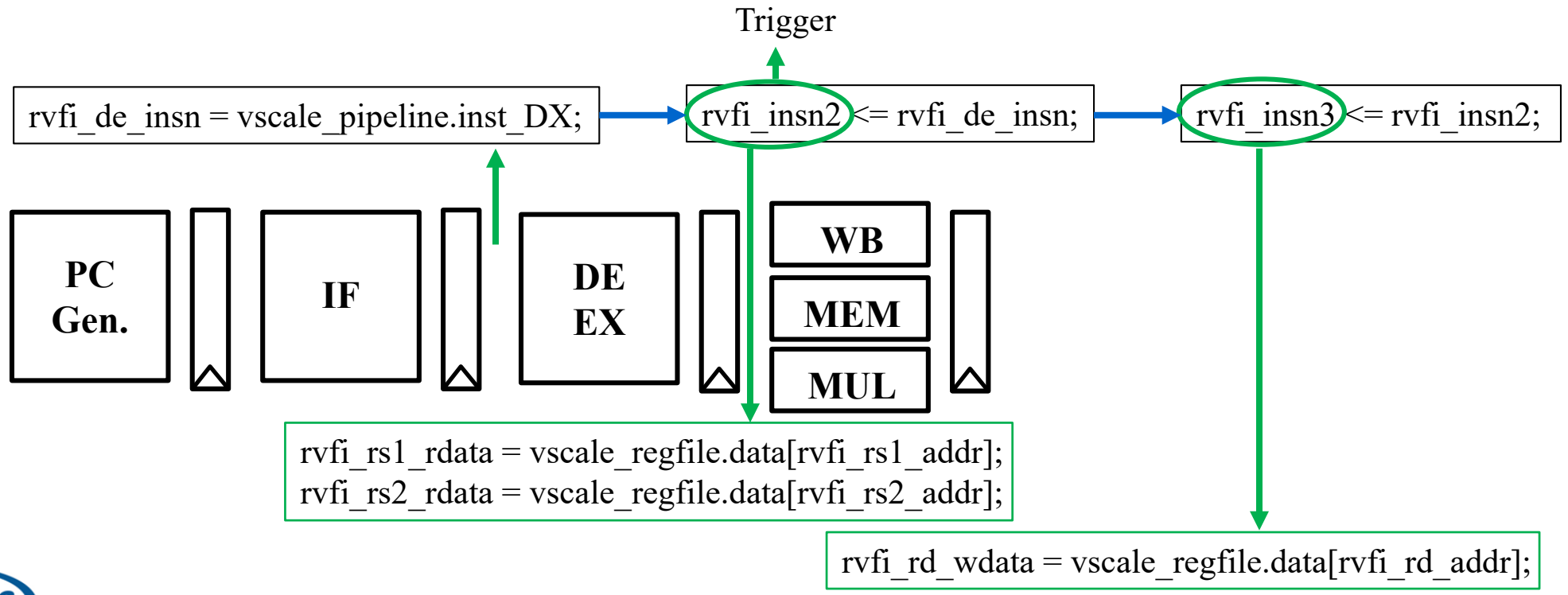
- Introduction
- Application
- Experimental results
- Conclusions

Proposed Workflow



Extend RVFI for Property Auto-generation

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



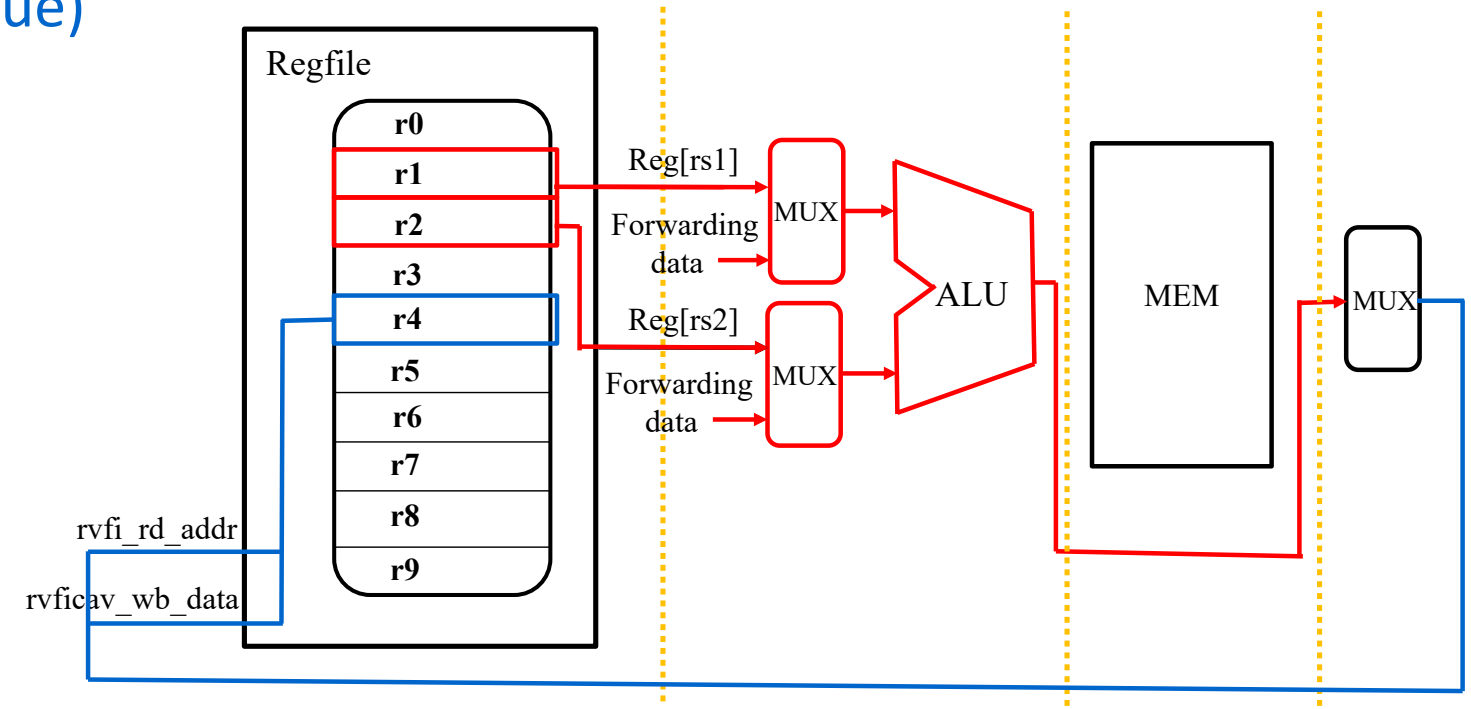
Extend RVFI for Property Auto-generation (cont'd)

- Original SVA property for checking *add* instruction

```
1   logic [31:0] gold_add;
2   property ori_add;
3   @(posedge clk )disable iff(reset)
4   addtrigger
5   |=>
6   (gold_add == rvfi_rd_wdata );
7   endproperty
8   property_ori_add:assert property (ori_add);
9
10  always_ff@(posedge clk) begin
11   gold_add <= rvfi_rs1_rdata + rvfi_rs2_rdata;
12  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 *add* Instruction

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

1. Checks the correctness of the data forwarding

```
1   logic [31:0] wb_data_pipe;
2   always_ff@(posedge clk) begin
3    wb_data <= vscale_regfile.wd;
4   end
5
6   property rd_wb_test;
7   @(posedge clk )disable iff(reset)
8   addtrigger
9   |=>
10  (wb_data == rvfi_rd_wdata );
11  endproperty
12  property_rd_wb_test:assert property (rd_wb_test);
13
14
15  logic [31:0] gold_add;
16  property fwding_add;
17  @(posedge clk )disable iff(reset)
18  addtrigger
19  |=>
20  (gold_add == wb_data );
21  endproperty
22  property_fwding_add:assert property (fwding_add);
23
24  always_ff@(posedge clk) begin
25   gold_add <= rvfi_rs1_rdata + rvfi_rs2_rdata;
26  end
```

Compositional Formal Verification Method

- Assume-guarantee reasoning

$$\frac{M \parallel A \models P \qquad N \models A}{M \parallel N \models P}$$

Premise

Conclusion

- M and N : Components
- A : Assumption
- P : Property
- ‖: Composite
- ⊨ : 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

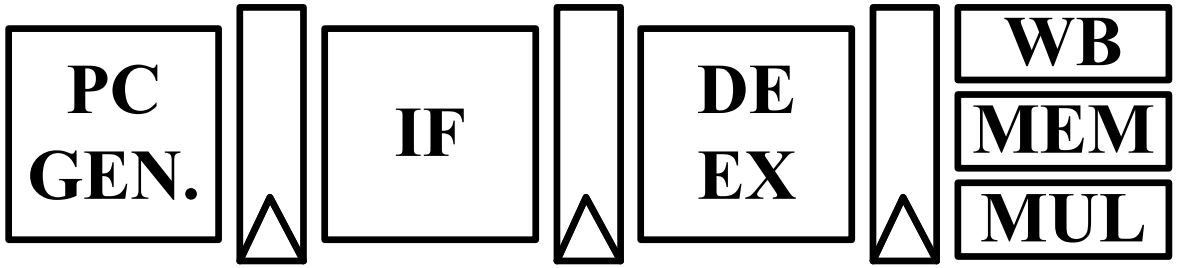
A = assumption describing that the correctness of data forwarding is assumed valid

Outline

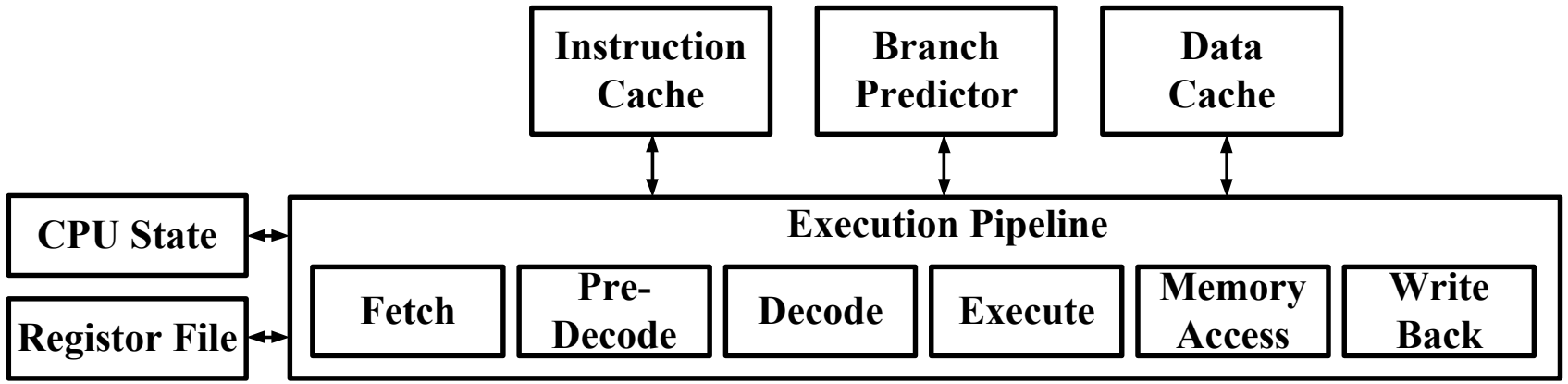
- Introduction
- Application
- Experimental results
- Conclusions

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

	Original	Abstract	
Property name	*original_add*	*forward_add*	*rd_wb_test*
Result	Pass	Pass	Pass
Time (sec)	75140.4	1187.3	6.7
COI coverage of pipeline module	93.13%	93.13%	
Proof core coverage of pipeline module	48.98%	60.60%	

- **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

Instruction type	Number of properties	Execution time (second)	Verification result
R-type	22	16167.3	PASS (except *sra* instruction)
I-type	18	23482.0	PASS (except *srai* instruction)
B-type	12	1624.2	PASS
J-type	8	15.8	PASS (except *jalr* instruction)
L-type	12	38.2	PASS
S-type	8	39.3	PASS
U-type	4	29.8	PASS
Assumption	4	---	---
Total properties	88	---	---

Number of inconclusive instruction properties in RV12

Instruction type	# of properties (inconclusive/total) (without abstraction)	# of properties (inconclusive/total) (with abstraction)	Improvements (%)
R-TYPE	10/10	5/10	50.0%
I-TYPE	9/9	3/9	66.6%
J-TYPE	2/4	1/4	25.0%

Coverage information

- Vscale
  - Top module :
    - COI coverage : 92.80 %
    - Proof core coverage : 76.96 %
  - Core module :
    - COI coverage : 92.87 % → (waive) 98.18 %
    - Proof core coverage : 75.92 % → (waive) 89.11 %
- RV12
  - Top module :
    - COI coverage : 67.93 %
    - Proof core coverage : 61.17 %
  - Core module :
    - COI coverage : 87.85 % → (waive) 96.11 %
    - Proof core coverage : 83.33 % → (waive) 91.57 %

Defects found by our verification flow

- Vscale :
  - *sra* and *srai*
  - *jalr*
- RV12 :
  - *csrrwi*

Error in *sra* and *srai* instructions (Vscale)

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

  - Vscale ALU implementation

```
4  module vscale_alu(
5                  input [`ALU_OP_WIDTH-1:0] op,
6                  input [`XPR_LEN-1:0]      in1,
7                  input [`XPR_LEN-1:0]      in2,
8                  output reg [`XPR_LEN-1:0] out
9                  );
10
11    wire [`SHAMT_WIDTH-1:0]                shamt;
12
13    assign shamt = in2[`SHAMT_WIDTH-1:0];
14
15    always @* begin
16       case (op)
17          `ALU_OP_ADD : out = in1 + in2;
18          `ALU_OP_SLL : out = in1 << shamt;
19          `ALU_OP_XOR : out = in1 ^ in2;
20          `ALU_OP_OR : out = in1 | in2;
21          `ALU_OP_AND : out = in1 & in2;
22          `ALU_OP_SRL : out = in1 >> shamt;
23          `ALU_OP_SEQ : out = {31'b0, in1 == in2};
24          `ALU_OP_SNE : out = {31'b0, in1 != in2};
25          `ALU_OP_SUB : out = in1 - in2;
26          `ALU_OP_SRA : out = $signed(in1) >> shamt;
27          `ALU_OP_SLT : out = {31'b0, $signed(in1) < $signed(in2)};
28          `ALU_OP_SGE : out = {31'b0, $signed(in1) >= $signed(in2)};
29          `ALU_OP_SLTU : out = {31'b0, in1 < in2};
30          `ALU_OP_SGEU : out = {31'b0, in1 >= in2};
31         default : out = 0;
32       endcase // case op
33    end
```

Error in *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

```
17    wire [`XPR_LEN-1:0]
```

```
jalr_offset = { {21{inst_DX[31]}}, inst_DX[30:21], 1'b0 };
```

```
22    always @(*) begin
23       case (PC_src_sel)
24          `PC_JAL_TARGET : begin
25             base = PC_DX;
26             offset = jal_offset;
27
28          end
29          `PC_JALR_TARGET : begin
30             base = rs1_data;
31             offset = jalr_offset;
32
33          end
```

```
64       assign PC_PIF = base + offset;
```

Error in "*csrrwi*" instructions (RV12)

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

rs = x0 → **Shall not cause illegal instruction exception**

- ***csrrw*** and ***csrrwi***
  - Shouldn't concern whether source register is *x0*

rs = x0 → **Still cause illegal instruction exception**

◆ Part of RV12 decode stage implementation

```
724                 //system
725                 {1'b?,CSRRW }: illegal_alu_instr = illegal_csr_rd |             illegal_csr_wr ;
726                 {1'b?,CSRRS }: illegal_alu_instr = illegal_csr_rd | (|if_src1 & illegal_csr_wr);
727                 {1'b?,CSRRC }: illegal_alu_instr = illegal_csr_rd | (|if_src1 & illegal_csr_wr);
728                 {1'b?,CSRRWI}: illegal_alu_instr = illegal_csr_rd | (|if_src1 & illegal_csr_wr);
729                 {1'b?,CSRRSI}: illegal_alu_instr = illegal_csr_rd | (|if_src1 & illegal_csr_wr);
730                 {1'b?,CSRRCI}: illegal_alu_instr = illegal_csr_rd | (|if_src1 & illegal_csr_wr);
731
732                 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
  - *sra*, *srai* and *jalr* instructions in Vscale case
  - *csrrwi* instruction in RV12 case

Thank you for your attention!

Q&A

Author Contact Information: chenyr@mail.ncku.edu.tw