2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Automate and Accelerate RISC-V Verification by Compositional Formal Methods

Yean-Ru Chen, Department of Electrical Engineering, National Cheng Kung University, Tainan, Taiwan (R.O.C) (*chenyr@mail.ncku.edu.tw*)

Cheng-Ting Kao, Department of Electrical Engineering, National Cheng Kung University, Tainan, Taiwan (R.O.C) (*N26061775@mail.ncku.edu.tw*)

Yi-Chun Kao, Department of Electrical Engineering, National Cheng Kung University, Tainan, Taiwan (R.O.C) (*N26061513@mail.ncku.edu.tw*)

Tien-Yin Cheng, Department of Electrical Engineering, National Cheng Kung University, Tainan, Taiwan (R.O.C) (*N26061741@mail.ncku.edu.tw*)

Chun-Sheng Ke, Department of Electrical Engineering, National Cheng Kung University, Tainan, Taiwan (R.O.C) (*N26061814@mail.ncku.edu.tw*)

Chia-Hao Hsu, Department of Electrical Engineering, National Cheng Kung University, Tainan, Taiwan (R.O.C) (*N26061335@mail.ncku.edu.tw*)

RISC-V is an open-source instruction set architecture (ISA) initialized by UC-Berkeley. We borrow the concept of ARM's ISA-Formal and extend a RISC-V interface called RVFI to verify the correctness of RISC-V's implementations. Our proposed work can not only automatically generate the RISC-V RV32I properties for model checking but also accelerate the verification tasks to be convergent efficiently with compositional formal methods. The experimental results show that RISC-V formal properties are completely auto-generated and correctly verified by Cadence JasperGold. We not only detect the *sra*, *srai* and *jalr* defects in the latest version of Vscale in 7.1 hours but also find an error in *csrrwi* in the latest version of RV12.

I. INTRODUCTION

Central processing unit (CPU) is one of the most important components in computers. Basically, it performs to fetch and decode the instructions of computer programs and then execute the corresponding operations including arithmetic computation, logic controls, input/output (I/O) access and etc. With the needs of extra requirements increase, including less delay, more throughput and etc., there are many optimizations in modern microprocessor designs. These optimizations are generally achieved by improving some mechanisms like pipelining, forwarding, issuing more instructions per cycle, and so on. Each optimization, however, is highly possible to affect the original functionalities. Moreover, such behavior changes are usually hard to be detected by conventional simulation technique because they may only be triggered by some rare and special input stimulus. In 2016, A. Reid et al., the authors working in ARM, have proposed a framework called "ISA-Formal" [1]. It is an end-to-end verification to detect bugs in the datapath, pipeline control and forwarding/stall logic of processors. Similar to ARM's idea, C. Wolf proposed another formal verification framework [2] for RISC-V [3] based processors. This is also an end-to-end formal ISA verification. One of the main contributions is that they propose an interface called RISC-V formal interface (RVFI) which can benefit the design integration and verification reusability.

Figure 1 shows ARM's ISA-Formal flow. Original specification written in natural language is first manually translated into a kind of machine readable architecture specification. Then this specification is automatically interpreted by Python script into their proposed language called Architecture Specification Language (ASL). ASL is more close to register transfer language (RTL) design than machine readable architecture specification descriptions and it also benefits for translating specification descriptions to SystemVerilog Assertions as its syntax is similar to SystemVerilog language.

C. Wolf's proposed verification flow for RISC-V ISA is shown in Figure 2. The details of C. Wolf's work and the differences among the two work and ours are described below.

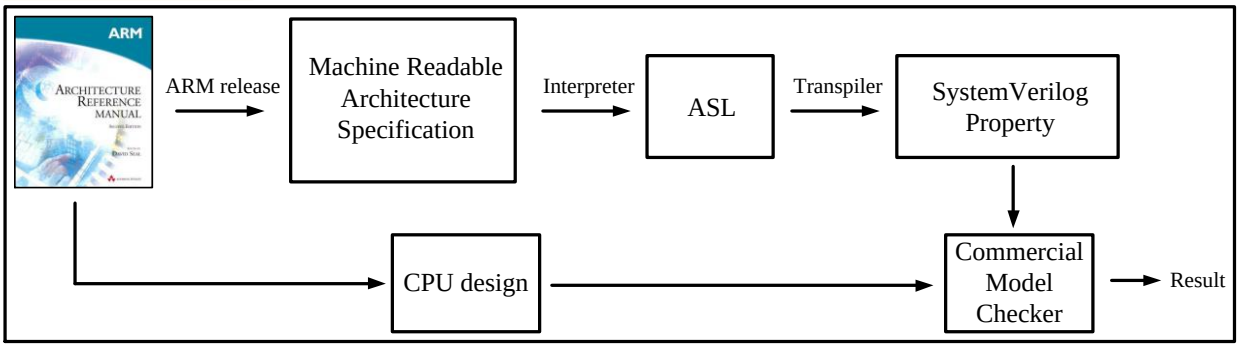

Figure 1 : ARM ISA-Formal flow

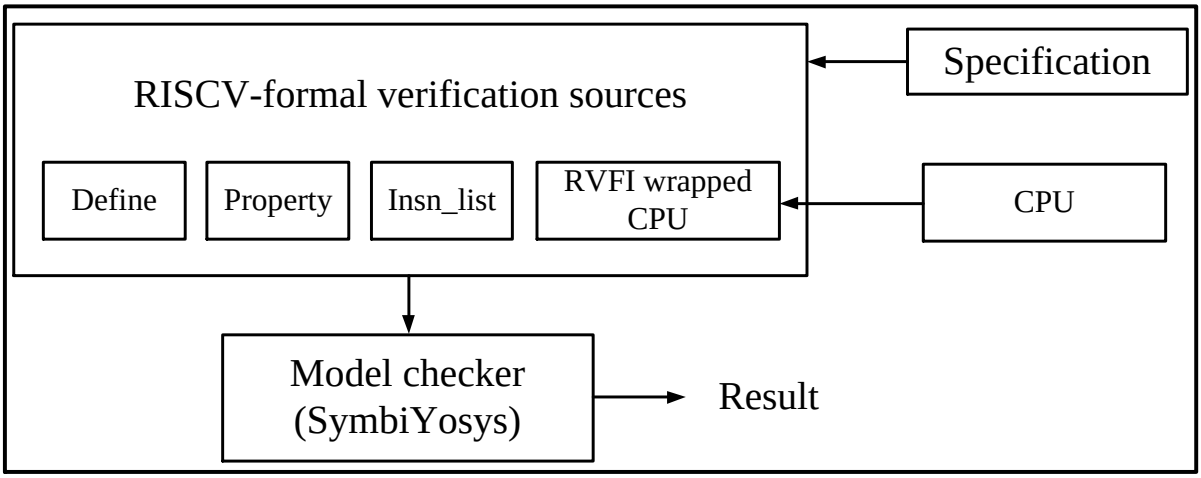

Figure 2 : Clifford Wolf's RISC-V formal verification flow

We have the same target as what ARM's ISA-Formal and C. Wolf's RVFI do; however, we have more enhancements than the two works. First, ARM's ISA-Formal mainly focuses on ISA properties and does not discuss how to formally verify the control status register (CSR) instructions with the concerns of the privilege modes. In C. Wolf's work, they do not provide formal property checking for the environment calls/breakpoints instructions. They check the read/write contents of CSRs but not for CSR instructions. Our work covers not only the verification tasks of whole RV32I instructions, including ISA properties, environment calls/breakpoints and CSR instructions with the concerns of the privilege modes (user mode, supervisor mode and machine mode), but also the formal property checkings of the exception bits and the correctness of the program counter values after the exceptions occur. We show the results in the experimental case named RV12 [4] in the section III. Second, the properties used in C. Wolf's work are manually created by formal verification experts, while our properties are automatically generated from a set of well-qualified and widely-adopted machine readable specification called riscv-semantics [5] with the help of our extended RVFI. Moreover, C. Wolf's work is bounded model checking RISC-V ISA formal properties; that is, it tackles the verification inconclusive issue by setting the verification bound before the property checking and try to explore each further step of the verification space by some manual improvements. Our work targets at full proof and thus we provide an embedded abstraction method called property splitting to explore the verification bounds as deeper as possible. Basically, our split properties are made for the verification space abstraction and they can be created automatically as long as we fill up some simple glue logics for the signal binding descriptions.

There are two more related research achievements proposed by U. Kühne et al. in 2010 [6] and R. Baranowski et al. in 2016 [7], respectively. However, U. Kühne's work is on architectural level, which is an "upper" and more "abstract" level compared with the circuit design RTL implementation level, and thus the formal properties used in this work are not for RTL designs. Moreover, they did not discuss how to guarantee the completeness and correctness of their generated properties with respect to the aspect of verification coverage. The formal properties of R. Baranowski work are for verifying automotive DSPs, while ours are for RISC-V processors verification.

Their properties are automatically generated from a given tabular specification; while our properties are automatically generated from the well-qualified and widely-adopted riscv-semantics. As we have just described above, we further extend RVFI, embed a property splitting technique for the properties related to the large register files, and we also work on the formal verification of the environment calls/breakpoints instructions, privilege modes and control status registers (CSR).

II. APPLICATION

A. *Proposed Workflow*

Our workflow is shown in Figure 3. Based on RISC-V ISA specifications, we extract the essential instruction behavior descriptions from a machine readable specification called riscv-semantics to be parts of the inputs of our parser which is a preprocessor of property generation. Our work also needs a few manual efforts on glue logic writings, such as providing signal mapping model to bind the extended RVFI to the corresponding signals of the design under verification (DUV), and specifying the signals of writing back data to the register files for the split properties generation. Then we can take the extended RVFI-wrapped CPU implementation as DUV and the generated SVA properties (split properties are included) as two inputs of JasperGold [8] (a commercial RTL verification tool from Cadence Design Systems, Inc.).

B. *Extend RVFI for Property Auto-generation*

We develop a parser to read the machine readable specifications written in Haskell [9] and the signal mapping model. The signal mapping model describes the binding information of the DUT signals and the (extended) RVFI, which is a kind of glue logics for verification tasks. Take the instruction "*add rd rs1 rs2*" as an example. We create an extended RVFI called *rvfi_insnX* for keeping the decode information, *rvfi_de_insn*, to a later stage because it should be used to set up the verification triggering condition. Note that *X* is the number of pipeline stage. Signals *rvfi_rs1_rdata* and *rvfi_rs2_rdata* are used for reading the data stored in the source registers *rs1* and *rs2*, respectively. The *rvfi_rd_wdata* is for reading the data stored in the destination register.

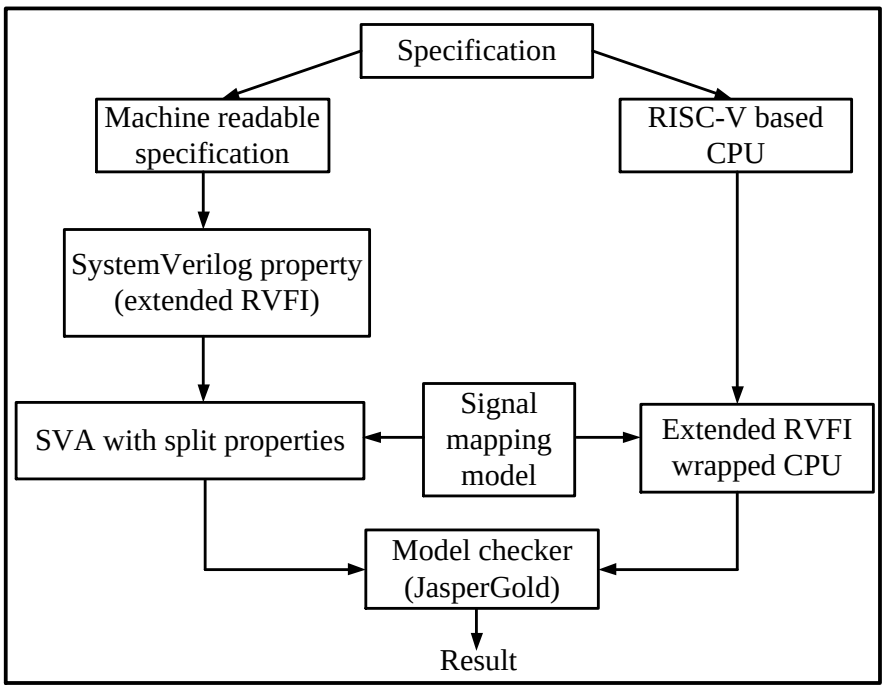

Figure 3 : Our proposed RISC-V formal verification flow

This work is supported by MOST 107-2221-E-006-202 and we thanks to Taiwan Semiconductor Research Institute for providing Cadence JasperGold license.

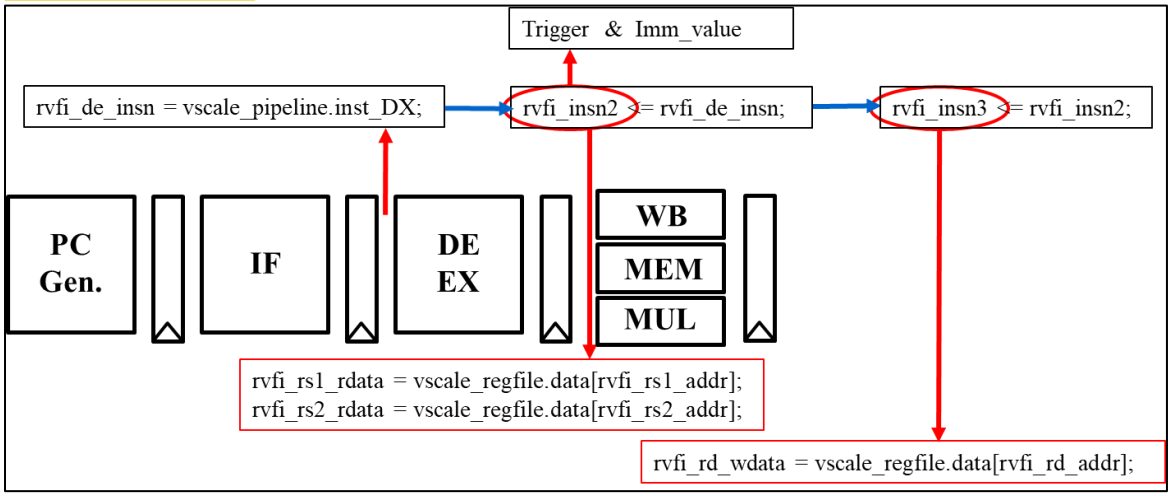

Figure 4 : Mapping model of implemented signals to extended RVFI instruction followers in Vscale

```
logic addtrigger;
assign addtrigger = (rvfi_insn2[`bit_opcode_end:`bit_opcode_start] == 7'b0110011)&&
                    (rvfi_insn2[`bit_funct3_end:`bit_funct3_start] == 3'b000)&&
                    (rvfi_insn2[`bit_funct7_end:`bit_funct7_start] == 7'b0000000)&&
                    (rvfi_insn2[`bit_rd_end:`bit_rd_start]!=5'd0);
```

Figure 5.1 : Triggering condition of *add* instruction for verification

Figure 4 shows partial signal mapping model between our extended RVFI and Vscale [10] in a pipeline stage diagram. Figure 5.1 shows pieces of codes of the triggering condition definitions. Figure 5.2 is the auto-generated original SVA property for the *add* instruction. The descriptions of the triggering condition can be auto-generated based on the "instruction decode" descriptions of riscv-semantics and the given signal mapping model. Meanwhile, the SVA property of *add* instruction is automatically generated based on the "instruction behavior" descriptions of riscv-semantics and the mapped RVFI signals. However, verifying this version of the generated SVA takes about 75140.4 sec. as shown in Table I. We try to mitigate the verification overhead with two abstraction techniques. First one is to simply reduce the verified data bits by fixing the index of data bits. Take 5-bit data stored in a register as an example, we can denote it as *data [index]*, while *index* = 0, 1, 2, 3, 4, to represent each bit of a data. We can reduce the verification complexity by fixing the *index* value to verify fewer bits of a data. This strategy is commonly used to verify the properties concerning large storage structures, such as buffer (stack), memory and queue (fifo). This reduction can be legal only when it is based on an assumption of that all the data bits are driven by the same cone-of-influence (COI) logics. Basically this assumption should be validated by analyzing the driving logics of the data bits. Once this assumptions is valid, by this assumption, it is not possible that only one data bit can pass the verification and others fail. However, sometimes reducing the verification data bits is not good enough, we further propose another method called property splitting to ease the state space explosion problem, which is described in the following paragraph.

```
logic [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
```

Figure 5.2. Original SVA property for checking *add* instruction

C. *Verification Space Abstraction by Property Splitting*

The original SVA property for the *add* instruction is able to verify two targets: *Target 1* checks the correctness of the data forwarding and *target 2* checks if the actual result data is correctly written back to the destination register. Now we split the original SVA into two properties: the property named *fwding_add* is for verifying *target 1* and the property named *rd_wb_test* is for *target 2*. Details are shown in Figure 8.

Figure 6 is the datapath of data forwarding from the source registers *r1* and *r2* of the given *add* instruction to a place where is just before the result written back to the register file. We set a signal named *wb_data* which indicates that the data is not yet written back to the destination register file but for the output along the datapath from *r1* and *r2* through ALU *add* computation.

For auto-generating the split properties, it is necessary to provide the binding information of the corresponding DUV signal to *wb_data* in the signal mapping model. Figure 7 is the datapath of *wb_data*. In target 2, the property *rd_wb_test* concerns two signals named *rvfi_rd_addr* and *wb_data*. Though *rvfi_rd_addr* is not explicitly written in the property, it is actually also in the driving logics of the signal *wb_data*.

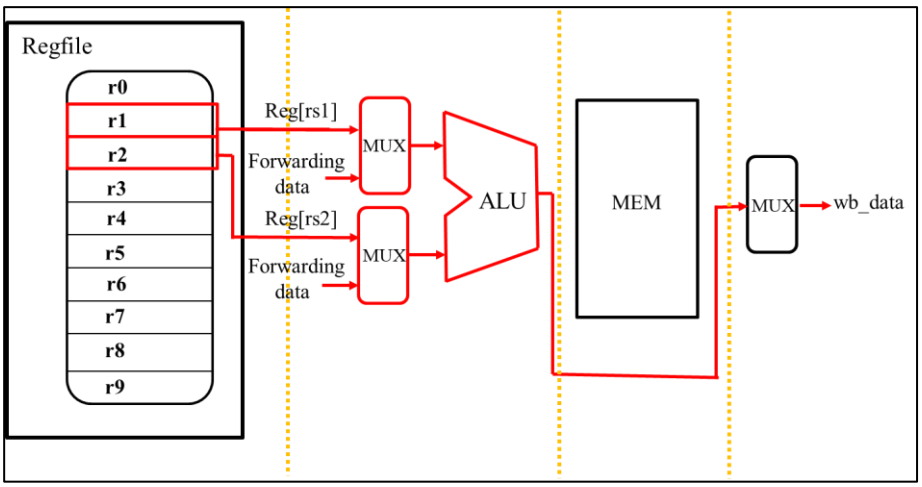

Figure 6 : Datapath of data forwarding from source registers

By splitting the property, the implementation constraints on *wb_data* can be *released*, as well as *rvfi_rd_addr*. That is, *rvfi_rd_addr* and *wb_data* can be arbitrary values when checking the property *rd_wb_test*. This abstraction brings great benefits to verification performance - we can reduce the verification time of *add* instruction from 75140.4 sec. to 1194.0 sec. as shown in Table I. Moreover, the proof core coverage of top module increases from 47.00% to 59.32% and proof core coverage of pipeline module increases from 48.98% to 60.60%.

We further use a compositional verification method called Assume-Guarantee Reasoning (AGR) [11] to not only guarantee the correctness of our property splitting method but also accelerate the verification tasks to explore deeper verification bounds. Take property splitting method for example, original SVA (*ori_add*) can be guaranteed by proving the two split properties (*fwding_add* and *rd_wb_test*). Formula (1) is the basic AGR form.

$$\frac{\mathbf{M} \parallel \mathbf{A} \vdash \mathbf{P} \qquad \mathbf{N} \vdash \mathbf{A}}{\mathbf{M} \parallel \mathbf{N} \vdash \mathbf{P}} \tag{1}$$

The upper part is called premise and the lower part is the reasoning conclusion. This formula claims as follows: if a composition of **M** and **A**, denoted as **M** ‖ **A,** while **M** is the system and **A** is an assumption, can satisfy property **P**, and system **N** can satisfy assumption **A**, then we can reason that the composition of **M** and **N**, denoted as **M** ‖ **N**, can satisfy **P**. Now we set: **M** is a sub-system implemented based on the property *rd_wb_test*, **A** is the assumption describing that Target 1 is assumed valid, **N** is a sub-system implemented based on the property *fwding_add*, and **P** is the original property *ori_add*. The conclusion that **M** ‖ **N** satisfies **P** can be reasoned as valid because that the assumption **A** is existing and guaranteed valid by the fact of that *fwding_add* property is proven true.

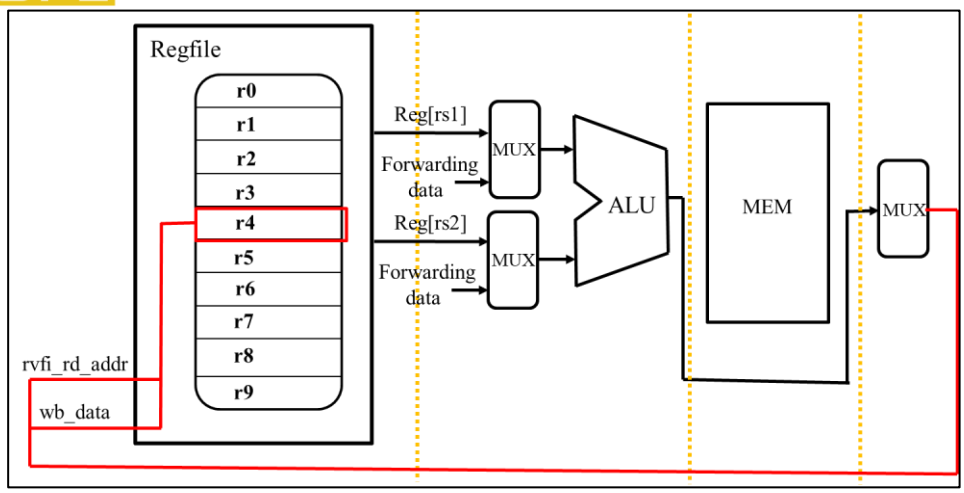

Figure 7 : Datapath of actual result written back to register file

```
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
```

Figure 8 : Split SVA properties for checking *add* instruction

III. PRELIMINARY RESULTS

All the experiments are verified by Cadence JasperGold 2018.03, on a server running CentOS 6.10., which has 48 cores with 2.20 GHz CPU and 256 GB memory embedded. Our result shows that we detect three errors in *sra*, *srai* and *jalr* of Vscale and one error in *csrrwi* in RV12.

Table I : Execution time and coverage information of original and split *add* properties of Vscale

	Original	**Abstract**	
Property name	*ori_add*	*fwd_add*	*rd_wb_test*
Time (second)	75140.4	1187.3	6.7
COI coverage of top module	90.86%	91.03%	
COI coverage of pipeline module	93.13%	93.13%	
Proof core coverage of top module	47.00%	59.32%	
Proof core coverage of pipeline module	48.98%	60.60%	

In Vscale case, the proof core coverage of top module is 76.96%. After further analysis, we find that the coverage holes are mostly out of our target of this work, including multiplication and division. After waiving these, we can obtain a higher proof core coverage, said 89.1%. Table II shows the overall ISA formal verification results of Vscale. After debugging, we find that the implementation of *jalr* in Vscale directly sets the first bit of the immediate value to be 0 and then adding it to *rs1*, which is different from RISC-V specification requirements. In the cases of sra and srai instructions, we find that the arithmetic right shift operator should be ">>>", while they are implemented as ">>" which is the logical right shift operator.

Table II : 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	---	---

Table III : Execution time and peak memory of Vscale

	Execution time (second)	Peak memory usage (MB)
R-type without abstraction	23788.9	572.47
R-type with abstraction	16167.3 (reduce 32.0%)	574.98
All-type without abstraction	34704.4	583.02
All-type with abstraction	25728.0 (reduce 25.9%)	577.60

Table III briefly compares the resource consumption when we verify Vscale with/without verification space reduction and abstraction methods. We can clearly see that reduction and abstraction techniques do improve the execution time. Table IV is for RV12. Totally 94 ISA properties and 60 properties for environment calls/breakpoints, CSR and exceptions are automatically generated for RV12. Most properties of L and U-types are full proven even without abstraction method applied. However, there 47.2% inconclusive properties of R, I, and J -types become full-proven after applying verification space reduction and abstraction methods. In Table V, even the verification tasks finally stop at 26 bounds for B-type and 22 bounds for S-type for RV12 on the average, compared with no abstraction method applied, they still obtain some bound improvements (2 to 10 more bounds are explored).

Expecting the verification engineers to manually create so many properties correctly and efficiently is not practical. Automatic property generation is a must for formal RISC-V processors verification. Moreover, with the abstraction methods, we can either save lots of the full proof time or have improvements on the verification bounds.

Table IV : 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%

Table V : Verification average bounds of B and S-type of RV12

	Average bounds (without abstraction)	Average bounds (with abstraction)
B-type	16	26
S-type	20	22

In RV12, there is a *csrrwi* error detected. Based on the specification, when *rs1* is *x0*, *csrrs*, *csrrc*, *csrrsi* and *csrrci* will not write CSR. That is, there will be no illegal instruction exception to the accesses to read-only CSRs. However, there is no such rule for *csrrw* and *csrrwi*. The bug is shown in the code line 728 in Figure 9, *csrrwi* considers if *rs1* is *x0* to decide if any exception, which is different from the specification descriptions. Actually, it should be the same as the code line 725.

```
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;
```

Figure 9 : CSR instruction implementations in RV12

IV. CONCLUSIONS

In this work, we propose a verification flow to automatically generate the formal properties for all the RISC-V RV32I instructions. Moreover, we integrate an abstraction technique into property generation flow to mitigate the state space explosion problem. We detect the defects of *sra*, *srai* and *jalr* of Vscale and *csrrwi* error implementation of RV12. For the Vscale case, our work can even reduce 98.4% verification execution time and achieve full proof with 89.1% proof core coverage.

V. REFERENCES

[1] A. Reid, R. Chen, A. Deligiannis, D. Cilday, D. Hoyes, W. Kenn, A. Pathirane, O. Shepherd, P. Vrabel and A. Zaidi, “End-to-End Verification of ARM Processors with ISA-Formal,” 28th International Conference on Computer Aided Verification (CAV) (Toronto Canada), LNCS Vol.9780, pp. 42-58, Springer, July 2016.

[2] C. Wolf. RISC-V Formal Verification Framework. https://github.com/cliffordwolf/riscv-formal

[3] RISC-V ISA. https://riscv.org/risc-v-isa/

[4] Roa Logic. RV12 RISC-V Processor. https://roalogic.com/portfolio/riscv-processor/

[5] Programming Languages and Verification Group at MIT CSAIL. Riscv-semantic. https://github.com/mit-plv/riscv-semantics

[6] U. Kühne, S. Beyer, J. Bormann, and J. Barstow, "Automated formal verification of processors based on architectural models," IEEE Proceedings of the 10th Formal Methods in Computer Aided Design (FMCAD), pp. 129-136, October 2010.

[7] R. Baranowski and M. Trunzer, “Complete formal verification of a family of automotive DSPs,” Design Verification Conference Europe (DVCON-Europe), pp. 456–485, May 2016.

[8] Cadence Design Systems, Inc. JasperGold Formal Verification Platform (Apps)

[9] S. Thompson, Haskell: the craft of functional programming. Addison-Wesley, 2011.

[10] RISC-V Foundation. RISC-V in Verilog. https://riscv.org/2015/09/risc-v-in-verilog/

[11] D. Giannakopoulou, C. S. Pasareanu and J. M. Cobleigh, “Assume-Guarantee Verification of Source Code with Design-Level Assumptions,” IEEE Proceedings of the 26th International Conference on Software Engineering (Scotland UK), v 26, pp. 211-220, May 2004.