A Hybrid Verification Solution to RISC-V Vector Extension

Chenghuan Li
Mediatek.inc, Beijing, China
Chenghuan.Li@mediatek.com

Yanhua Feng
Mediatek.inc, Beijing, China
Yanhua.Feng@mediatek.com

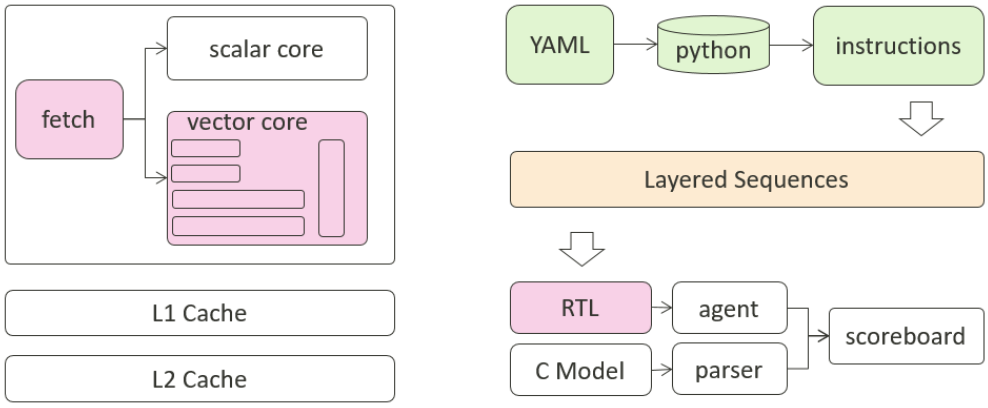
Liam Li
Mediatek.inc, Beijing, China
Liam.Li@mediatek.com

***Abstract*- Since the inception in 2010, RISC-V has gained more and more popularity. RISC-V vector extension (RVV) as a standard extension was introduced to enhance its capability in AI applications. RVV requires more complex exception and hazard handling, which raises a big challenge for verification. In this paper, we will demonstrate a practical hybrid solution to RVV verification. A flexible and automated instruction modeling flow is proposed to catch up with the continuous evolution of RISC-V instructions. For exception verification, a UVM-based solution is adopted to satisfy the requirement of RVV instruction's contextual relevance. For hazard handling, a simulation & formal hybrid solution is adopted to achieve better design quality with less simulation resources and signoff schedule shift left.**

I. Introduction

Due to its compact, modular and extensible characteristics, RISC-V has been gaining more and more popularity. As a standard extension of RISC-V, RVV v0.7.1 was released in June/2019. After that, its maturity has been highly increased along with v0.8, v0.9 and v1.0 releases. Over 300 vector instructions are introduced in RVV, covering load/store, integer, fixed-point and float-point operations.

RVV along with some other customized ISAs are implemented in our design with 12 stages pipeline, 4-issue superscalar architecture and configurable L1/L2 cache, as shown in figure 1(a). RVV has a higher requirement for the instruction's contextual relevance, which makes exception verification more challenging. To gain better PPA (performance, power and area), our design implemented multi-lane parallel execution with different retire stages, resulting in complex data hazard handling. To validate such a design, we created a UVM-based testbench, as shown in figure 1(b). In the paper, the demonstration will be focused on the YAML based instruction model auto-generation flow, exception verification and a simulation & formal hybrid solution to hazard handling verification.



(a) Design block diagram (b) Testbench diagram

Figure 1. Overall design and testbench diagram

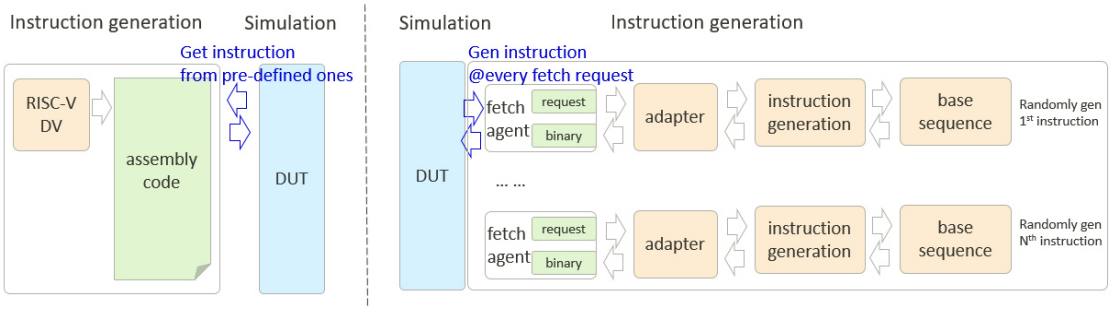
II. YAML Based Instruction Model Auto-Generation Flow

Google has created a SV/UVM based instruction generator (called RISC-V DV [1]) for RISC-V processor verification, which is available on GitHub. The biggest difference between RISC-V DV and our generator is the way to generate instruction. Google's RISC-V DV adopts the 'offline' way to generate instruction, while our generator use the 'online' way as shown in Figure 2.

- 'Offline' means that all instructions are generated before simulation.
- 'Online' means that instructions are dynamically generated during simulation.

The advantage of ‘online’ instruction generation is that it increases the controllability of stimulus. The instructions can be generated using the feedback information from design to hit more corner cases.

Another difference is that the instruction generated in our generator contains the implementation-aware information. The information includes but not limited to the instruction’s retire stage, source operands reading stage, the issue rules and the exception rules etc. This information provides feedback to stimulus generation and work as the ‘golden rule’ for checkers. For example, our design implementation adopts 4-issue superscalar micro architecture (i.e. at most 4 instructions can be issued together at one cycle). However there are some multi-issue requirements when issuing multiple instructions together due to hardware resource limitation. That is to say, some instructions can be issued together but others cannot. Each instruction’s issue rule is described in its instruction class. When generating stimulus, the generator can reference the issue rule to select the following instructions. In this way, we can improve the efficiency of full verification of the issue rules. What’s more, the issue rule works as the ‘golden rule’ to check the correctness of RTL’s issue result in multi-issue localized checker.



(a) Offline instruction generator (b) Online instruction generator

Figure 2. Two ways to generate instructions

As is shown in Figure 3, each instruction’s hardware resources, name, issue rule, exception rule, operands and binary decode are described in the YAML file. The group information which describes each instruction’s classification is described in the template file. The YAML and template files then work as the input file to a Python script. There are two output files: one contains all instructions class and the other one contains the classification result. The instruction class provides various query functions, for example binary, assembly, operands etc. Besides, coverages including operands value and issue rules are also provided in instruction class. The classification result works as constraints in stimulus generation. In RVV exception verification, the classification result would be used in exception handling sequence, which will be introduced in Section III.

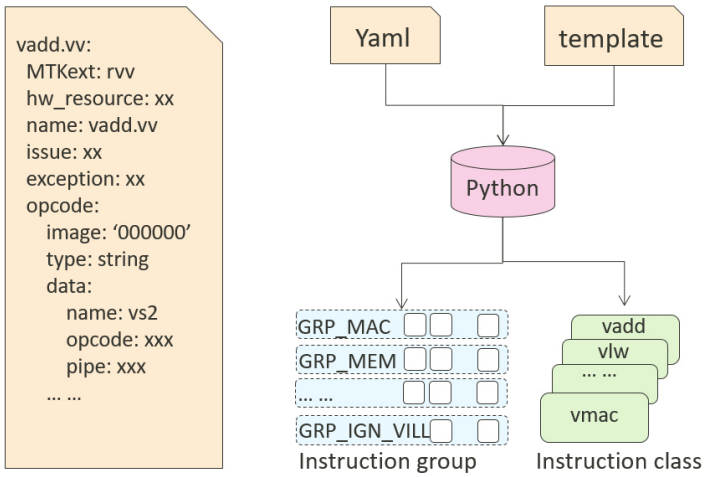


Figure 3. YAML auto-gen instructions

The instruction class takes the responsibility of instruction modeling including both the fixed and random information. The fixed information includes binary decode, issue and exception rule, and the random

information is the operands index. To achieve this goal, instruction class is organized as shown in Figure 4. In general, the instruction class can be divided into two parts: manually maintained part and YAML auto generated part. This class is consisted of one configuration class and several other classes each of which corresponds to one operand. The configuration class appends more constraints for this instruction's generation, for example, hazard mode, hazard number and reserved registers etc. The operands (both source and destination) class provides the controllability of index selection and its corresponding coverages. APIs provided in base instruction includes:

- autogen (): calls each operand's randomization.
- out (): completes the instruction's creation, filling binary and sampling coverage.
- fill_transaction (): is responsible for the instruction decode.
- bin2ins (): transfers the binary code to assembly code.

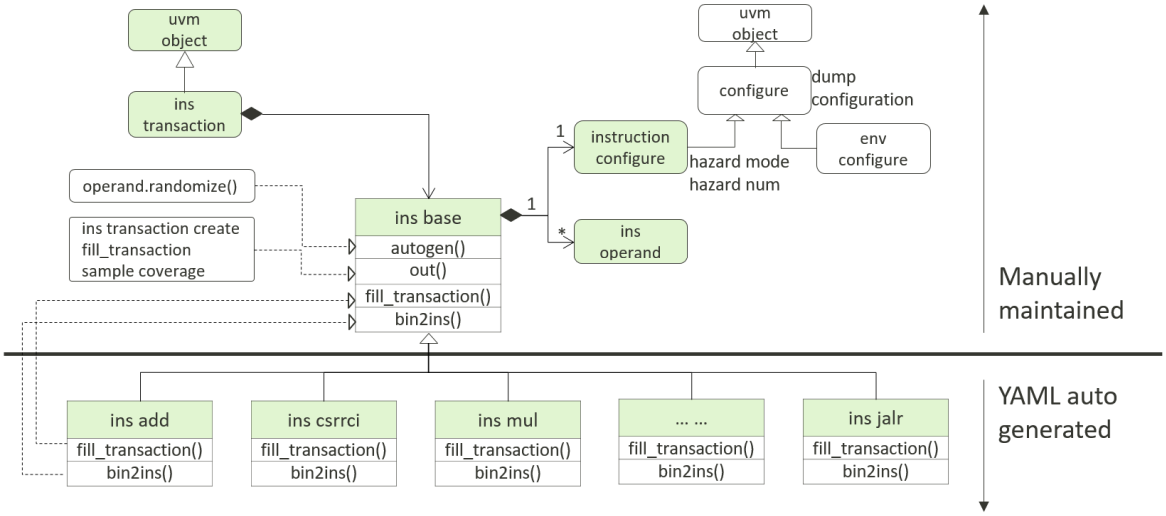


Figure 4. Instruction class

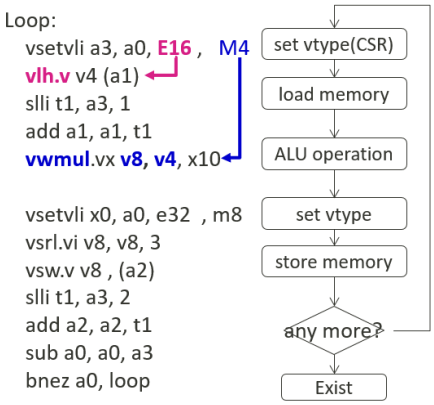
III. EXCEPTION VERIFICATION

Compared with other RISC-V standard extensions, RVV raises higher requirements for the instruction's contextual relevance. That is to say, one instruction would encounter exception if it is executed in an improper context which includes but not limited to unbefitting VTYPE setting, operand index selection and unaligned memory access etc. Figure 5(a) shows the possible exception category of RVV's instructions.

Figure 5(b) shows a snippet of RVV's typical usage. In this example, instruction 'vsetvli' set the VTYPE register as E16M4. The SEW=E16 requires that following instruction memory access size must not exceed 16 bytes. That is to say, 'vlb' or 'vlh' can execute successfully, but 'vlw' would encounter exception. The LMUL=M4, requires that the operand index of the following instruction must be a multiple of 4. If there is a violation, corresponding instruction would take exception.

Exceptions / Instructions		VTYPE			Operand index			Unalign	Rsvd	R/W
		LMUL	SEW	VILL	vd	vs2	vs1			
vsetvli(i)				√					√	
CSR				√						√
ALU	wdn	√	√	√	√	√	√			
	narr	√	√	√	√	√	√			
	norm			√	√	√	√			
Memory			√	√	√	√	√	√		
Special					√	√	√			
Customized		√	√	√	√	√	√	√	√	√

(a) Instructions and exceptions category



(b) Snipper of RVV's typical usage

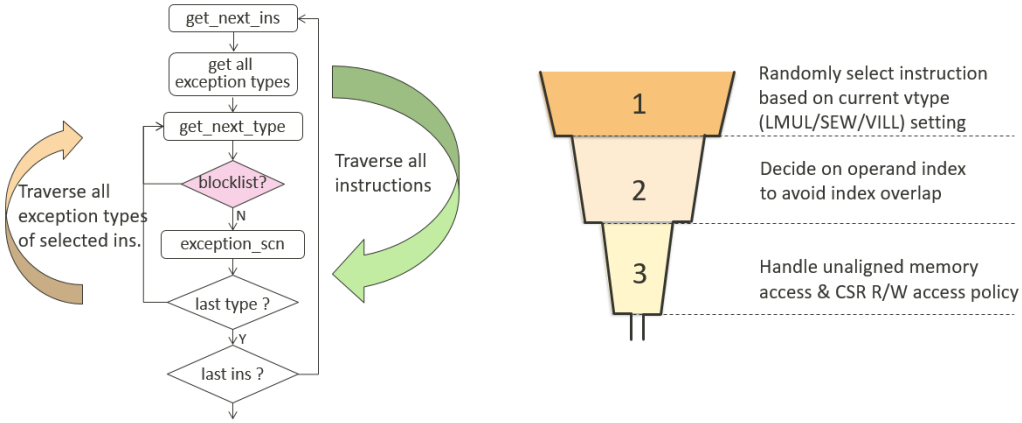
Figure 5. RVV's context relevance

This characteristic of RVV raises higher requirements for stimulus generation in two aspects:

a) For exception verification, testbench should be capable of traversing all possible exception cases.
b) For normal cases, testbench should generate legal instruction as much as possible, to avoid the simulation being interrupted frequently by exception handling.

A layered sequence is created to satisfy these two requirements:

a) For exception detection verification, a two-layer loop is used to traverse all possible exception cases for all instructions, as shown in figure 6(a). The outer loop traverses all instructions and the inner loop traverses the selected instruction's all possible exception cases. Each instruction's exception categories are automatically generated in the YAML auto-gen flow, and a block list is provided to handle the RVV's special rules (e.g. most RVV instructions would encounter exception if VILL is set, but vmv1r would not).
b) For legal instruction cases, a 3-step 'funnel' constraint solver is created to try the best to generate legal instruction, as shown in figure 6(b).
    1. First, randomly select instruction based on current VTYPE setting. For example, avoid selecting double widening instruction if LMUL=M8; avoid selecting memory instruction whose size exceeds current SEW's setting.
    2. Then decide on operand index based on current instruction type and LMUL setting. Operand index overlap issue is also solved in this layer.
    3. Finally, handle CSR instruction policy issue and memory access unaligned issue.



(a) Exception cases: traverse exceptions (b) Normal cases: generate legal instructions

Figure 6. Exception handling

IV. HAZARD VERIFICATION

Formal verification has been successfully used by a lot of companies to verify complex SOCs and safety critical designs. Different from simulation, formal can ensure the completeness of the verification. However, a problem commonly faced while verifying a design using formal is 'convergence'. If the state space for a given property is high, then it may end up being inconclusive (i.e. the proof is not able to complete even after a very long time). It is hard to complete the verification signoff using formal even though several techniques can be adopted to handle the convergence issue. So in our project execution, we adopt formal as a limited powerful methodology to solve some specific problems. We will demonstrate the formal application in the RISC-V vector extension verification. Generally speaking, we applied formal property verification in two ways: early stage design exploration and late stage bug hunting;

A. Early stage design exploration

Design verification starts with the design specification study. The problem most frequently encountered is the gap in the understanding of the design specification between designers and verification engineers as shown in Figure 7. It takes longer to eliminate this gap using traditional simulation. Another problem is the design quality of the initial version is usually filled with some obvious issues (e.g. typos). Simulation can't detect such issues until testbench is ready.

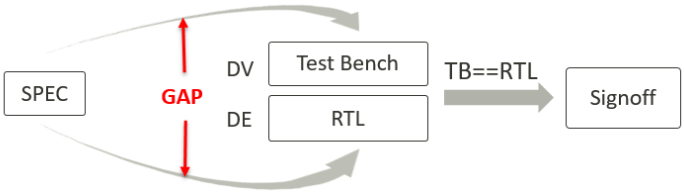


Figure 7. Design specification understanding gap

Formal property verification (FPV) is featured with easy environment buildup and fast debug iteration. The solution we adopt to solve these two problems is to apply FPV at the early verification stage to take advantage of these characteristics in formal. The purpose of FPV in this stage is to explore design behavior in relation to the specification and detect low-hanging fruits.

In order to achieve this goal, we created a series of properties during design specification study. The properties created at this stage consisted of plenty of 'cover' properties and a few of 'assert' properties. The 'cover' properties mainly cover:

- Possible state transition sequence in finite state machine (FSM)
- Typical RVV instruction's retirement
- Scenarios described in design specification
- Special design implementation for some features

The 'assert' properties mainly check interface protocol, including:

- Instruction control signal must be inside legal values
- Source operand read and destination operand write signals
- Design output signal's mutual-check (i.e. using one design output signal to check another one)

It can be drawn from our experience that about 80% early design issues can be detected this way as shown in Figure 8. The elimination of design specification understanding gap can dramatically reduce the iteration times with designers. What's more, an additional benefit is that 'cover' properties created at this stage can work as function coverage in simulation signoff.

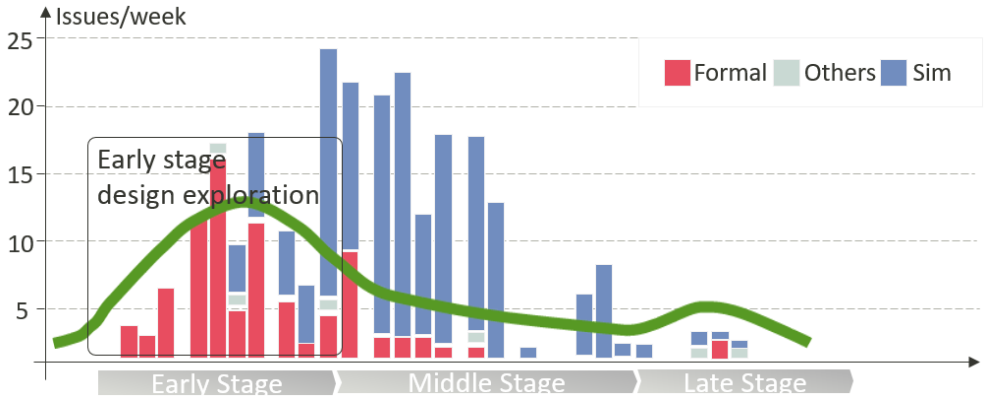
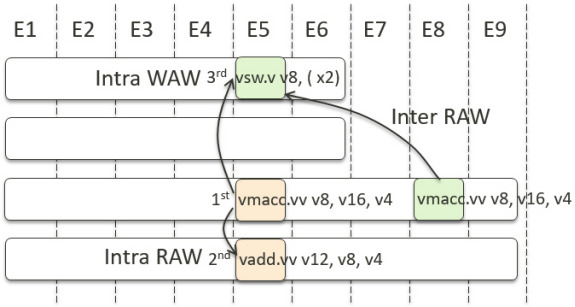


Figure 8. Formal's contribution at different verification stages

B. Late stage bug hunting

Data hazard handling is an import part of processor verification. This problem becomes more pronounced when it comes to RVV. In general, only inter RAW (read after write) hazard need to be taken into consideration. In order to get better performance, multi-lane execution is always adopted in RVV implementation, which requires extra consideration for intra data hazard. What's more, in our design, to reduce implementation resources, instruction retire stage is different among different lanes. So it is necessary to take WAW (write after write) and WAR (write after read) into consideration besides RAW (read after write) data hazard. A demonstration of inter RAW, intra WAW/RAW hazard is shown in Figure 9(a).



Expect\Actual	Stall	Forward	Not handled
Stall	√	??	Func. bug
Forward	Perf. bug	√	Func. bug

formal
simulation

(a) Data hazard examples (b) Verification strategies for data hazarding

Figure 9. Data hazard demonstration and verification

Stall and data forwarding are two common design techniques to handle data hazard. From the perspective of verification, the relationship between the design intent and its implementation on the hazard handling is shown in Figure 9(b). As is shown, simulation can detect cases that would cause functional bugs by cross-checking instruction correctness with golden C model. But it failed to detect the cases that would cause performance issue. What's worse, simulation may fail to detect the functional bugs despite having huge regression suite accumulation.

To solve the problems mentioned above, we introduced FPV to enhance the data hazard handling verification at the verification late stage. There are two purposes using FPV at this stage:

1. Detect the functional bugs which are hard to be detected in simulation;
2. Detect the performance bugs.

The approach we adopted is as follows:

- First, using liveness property to assert typical instruction's retirement.
- Second, using cover property to explore the longest stall cycles.
- Third, change liveness property to safety one, re-prove it.

For this complex data hazard handling design, what we worried the most was 'deadlock'. So we created liveness property to assert typical instruction's eventual retirement. What we expected formal engine to find was whether there exist possible scenarios where the selected instruction cannot retire forever. The property we used was as follows:

```
property ast_vmv_nfr_eventually_retire;
   logic [4:0] colored_idx;
   @ (posedge clk)) disable iff (! rstn)
   (insn_vld_e1 & insn_is_vmv_nfr_e1, colored_idx=insn_idx_e1) |->
   s_eventually (insn_wr_vrf & vrf_idx==colorder_idx);
endproperty
rvv_ast_vmv_nfr_eventually_retire: assert property (ast_vmv_nfr_eventually_retire);
```

This property really found one counter example that would be demonstrated in following part. After RTL bugs was fixed, formal engine ended up being inconclusive with limited resources. However, by debugging the counter example, we found that one stall at E5 stage was never dropped once raised. Inspired by this trace, we created another series of properties shown here:

```
property ast_stall_not_raised_forever (stall);
   @ (posedge clk) disable iff (! rstn)
   $rose (stall) |-> ## [1:$] (stall==1'b0);
endproperty
rvv_ast_stall_not_raised_forever: assert property (ast_stall_not_raised_forever (x_stall));
```

Generally, it was difficult for formal engine to prove these liveness properties. So we changed our mindset from asking formal engine to prove the liveness property to finding out the longest stall cycle. With assuming out a few known long cycle cases, we redirected from 'assert' liveness property to 'cover' property.

- No background math instructions whose execution can consume hundreds of cycles
- Reduce/mask compare/mask logic instructions executed under the condition of LMUL=M1

- NFR of vmv.nfr instructions cannot be 8.

```
property cov_stall_raised_20t (stall);
   @ (posedge clk) disable iff (! rstn)
   $rose (stall) |-> stall [*20];
endproperty
rvv_cov_stall_raised_20t: cover property (cov_stall_raised_20_t (x_stall));
```

Formal engine easily found traces that satisfy these requirements. After analysis, these traces were divided into two categories:

1. Reasonable cases (i.e. 20 was not the longest stall cycles)
2. Performance bugs (i.e. It wasn't necessary for RTL to raise stall, but it really did)

After the performance bugs were fixed in RTL and the stall cycle was finally increased to be 30, formal engine failed to find such a trace satisfying the requirements. So we drew conclusion that the longest stall cycle wouldn't exceed 30.

With this conclusion, we changed the liveness properties to safety ones. After 80 hours' running, this property was proven by formal engine.

```
property ast_vmv_nfr_eventually_retire;
   logic [4:0] colored_idx;
   @ (posedge clk)) disable iff (! rstn)
   (insn_vld_e1 & insn_is_vmv_nfr_e1, colored_idx=insn_idx_e1) |->
   ## [8:30] (insn_wr_vrf & vrf_idx==colorder_idx);
endproperty
rvv_ast_vmv_nfr_eventually_retire: assert property (ast_vmv_nfr_eventually_retire);
```

C. Bug examples

We will demonstrate two examples found using the above methods in the following part. One is a functional bug and the other one is a performance bug.

First example was a functional bug that would have been very hard to be detected in simulation, as shown in Figure 10. The program sequence was shown here. In this scenario, the second instruction 'vnclip.qv' was stalled at 'E5' stage forever (i.e. 'vnclip.qv' would never retire).

1st vlw.v v1, (x2)
2nd vnclip.qv, v8, v0 (MF4)
3rd vlw.v v2, (x2)
4th vlw.v v1, (x2)
5th vlw.v v2, (x2)
6th vlw.v v1, (x2)
…

The root cause of this deadlock was the unexpected RAW stall from 1st instruction 'vlw.v' to 2nd instruction 'vnclip.qv'. Following is the detailed description.

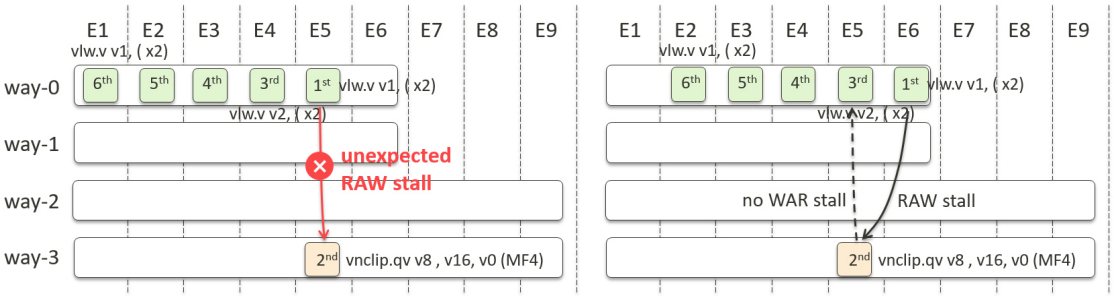
The 'vnclip.qv' was a double narrowing instruction who executed at MF4. At cycle T shown in Figure 10(a), both 'vlw.v' and 'vnclip.qv' entered 'E5' stage and then design started to handle data hazards. In reality, there was no RAW data dependncy between them, because:

- Only considering double narrowing, 'vnclip.qv' needed v0, v1, v2, v3, totally 4 registers.
- With additional consideration of LMUL=MF4, 'vnclip.qv' only needed v0.

However, design didn't take fractional LMUL (MF4) into consideration when handling RAW data hazard, resulting in this redundant stall. So far, the unexpected RAW stall only caused performance issue.

At cycle T+1 shown in Figure 10(b), the 1st 'vlw.v' entered 'E6' and the 3rd 'vlw.v' entered E5. At this time, 1st 'vlw.v' raised inter RAW stall to 2nd 'vnclip.qv', but 2nd 'vnclip.qv' did not raise WAR stall to 3rd 'vlw.v', because design considered both narrowing and framctional LMUL when handling WAR data hazard. So it went back and forth resuling in 2nd 'vnclip.qv' can never retire.

In a summary, RTL considered fractionl LMUL (MF4 in this example) when handling WAR hazard stall, but ommited it when handling RAW hazard stall, resuling in the 2nd vnclip.qv instruction being stall forever.



(a) Cycle T (b) Cycle T+1

Figure 10. Example of functional bug

Second example was a performance bug that was found using 'cover' property, as shown in Figure 11. The first instruction was 'vadd', the second one was 'vlw' and the third one was 'vmac'.

1st vadd v4, v2, v0
2nd vlw.v v8, (x2)
3rd vmac v8, v16, v24

According to the issue rules, 'vlw' was sent to way-0, 'vadd' was sent to way-2 and 'vmac' was sent to way-3. As mentioned above, data hazard stall was handled in pipeline stage 'E5' in our design. In this case,

- The **excepted** register retire order: v8 (vlw) -> v4 (vadd) -> v8 (vmac), totally needed 10 cycles.
  - 1st instruction 'vadd' retired at E9 without any data hazard stall;
  - 2nd instruction 'vlw' retired at E6 without any data hazard stall;
  - 3rd instruction 'vmac' encountered one cycle's inter WAW stall and then retired at E9.
- The **actual** register retire order: v4 (vadd) -> v8 (vlw) -> v8 (vmac), totally needed 14 cycles.
  - 1st instruction 'vadd' retired at E9 without any data hazard stall;
  - 2nd instruction 'vlw' encountered unexpected WAW stall and could not retire until 'vadd' retired;
  - 3rd instruction 'vmac' encountered one cycles' inter WAW stall and then retired at E9.

The actual registers retire order did not cause functional bug, but the unexpected WAW stall caused about 50% performance drop in such program sequence.

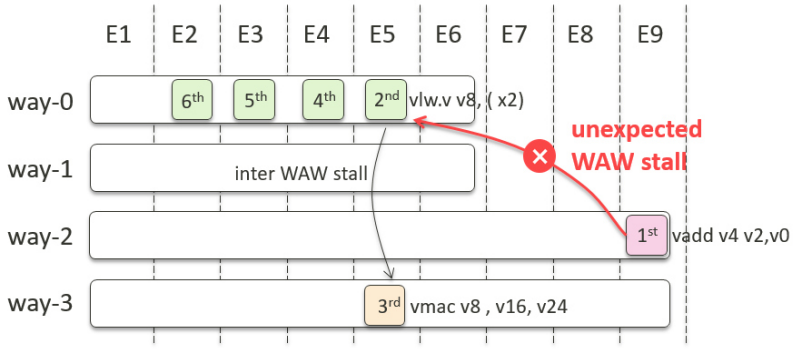


Figure 11. Example of performance bug

D. Results

A formal & simulation hybrid method was adopted for data hazard verification. The hazard handling logic was implemented in an individual module in our design. We conducted the formal property verification for this module by creating 80+ properties. Figure 12 shows the result of the hybrid method in our project execution.

Three benefits are gained by using this hybrid method:

1. Early stage bug hunting. Formal could report design issues 1 week after RTL release, while it took almost 1 month to make testbench ready before starting simulation regression.
2. Signoff schedule shift left. Design issues that were detected at the late stage in simulation could be shifted left using formal property verification.
3. Better design quality. Fully proven properties increase our confidence in design quality.

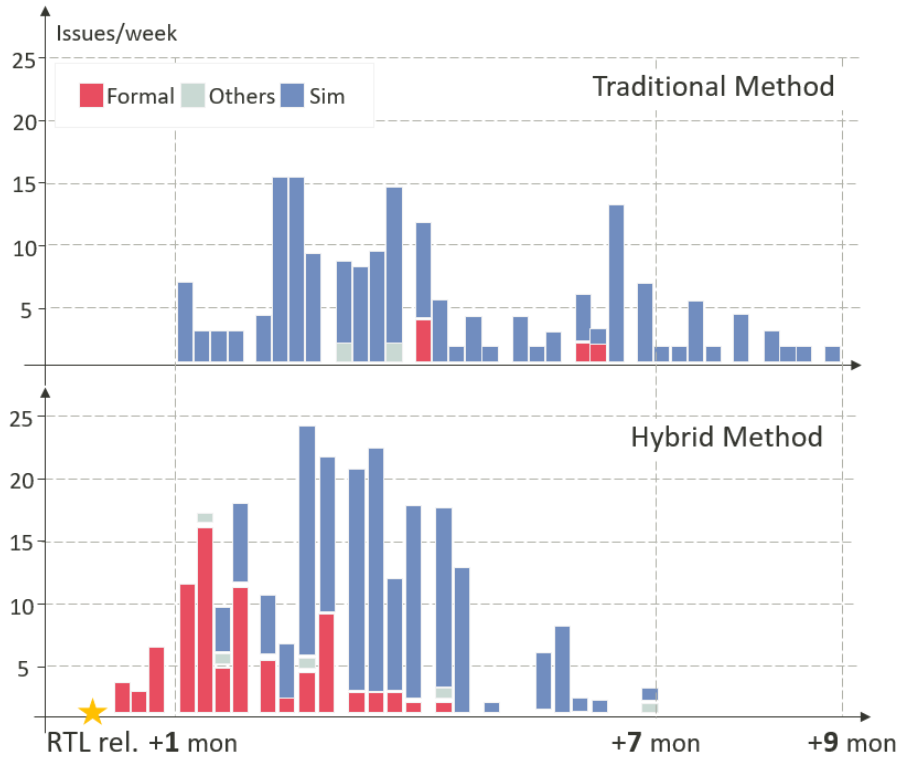


Figure 12. Comparison of traditional and hybrid method in our project

V. SUMMARY

In this paper, we presented our hybrid solution to verify RSIC-V vector extension. A YAML-based instruction model auto-generation flow is developed to respond to the changes from both RISC-V evolution and user customization. For exception verification, A UVM-based solution is adopted to satisfy the requirement of contextual relevance of RVV instructions. For hazard handling, a simulation & formal hybrid solution is adopted to shift-left signoff schedule with better design quality.

REFERENCES

[1] T. Liu and H. Richard, "UVM-based RISC-V Processor Verification Platform," RISC-V Summit, December 2018.