2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE 10 YEAR ANNIVERSARY

Planning for RISC-V Success

Verification Planning and Functional Coverage lead to quality RISC-V processor IP

Pascal Gouedo, Xavier Aubert, Yoann Pruvost, Dolphin Design, (pascal.gouedo@dolphin.fr, xavier.aubert@dolphin.fr, yoann.pruvost@dolphin.fr)

Aimee Sutton, Duncan Graham, Simon Davidmann, Imperas Software, (aimees@imperas.com, graham@imperas.com, simond@imperas.com)

I. Introduction

The CV32E40P [1], shown in Figure 1 below, is a 32 bit in-order RISC-V core that is being developed by the OpenHW Group [2], a not-for-profit, global organization driven by its members and individual contributors who collaborate in the development of open-source cores, related IP, tools and software. OpenHW provides an infrastructure for hosting high quality open-source hardware development projects in line with industry best practices with the goal of developing industrial-quality open-source IP.

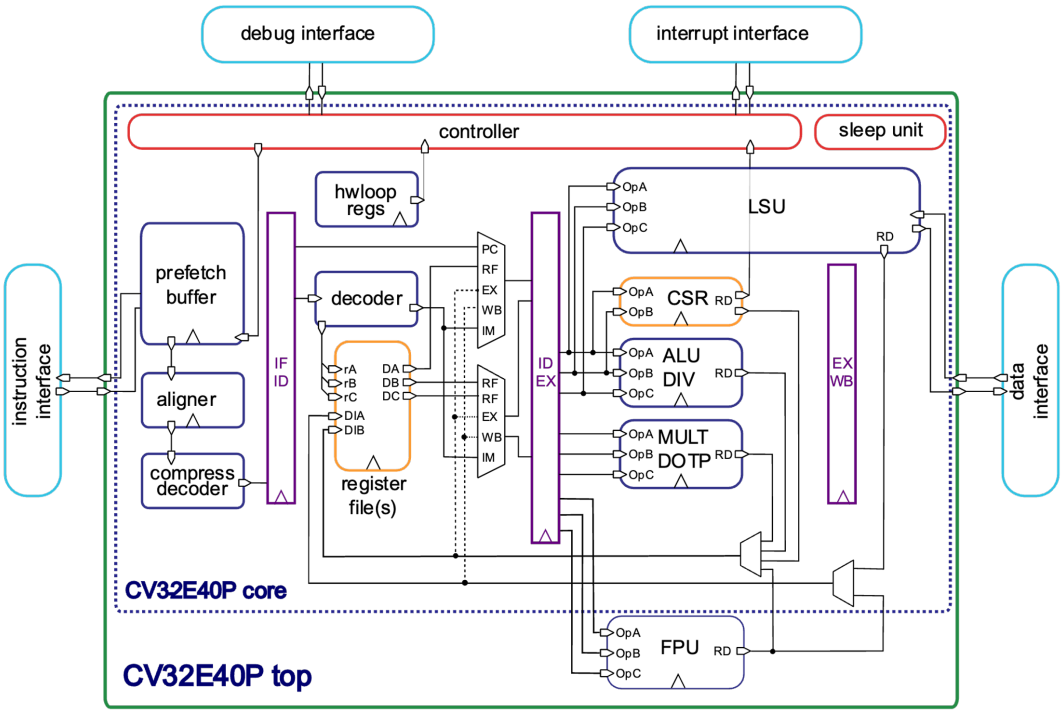

Figure 1: CV32E40P RISC-V Core

Dolphin Design became a member of the OpenHW Group in order to drive the continued development of the CV32E40P processor which will be used in its Panther DSP subsystem. The CV32E40P is a derivative of the RIS5CY core, which was originally developed at ETH Zurich and donated to the OpenHW Group in 2020. The CV32E40P was the first RISC-V core to achieve verification closure, a milestone defined by OpenHW as

Technology Readiness Level 5 (TRL-5) [3]. This milestone indicates, among other things, that the RTL has been subject to a comprehensive verification effort and is ready to go to silicon.

In the process of achieving TRL-5, some features of the CV32E40P were removed in order to meet the time objectives and technical requirements of the project's contributors. One of these features were the PULP instructions: a set of custom instructions designed to achieve higher code density, performance, and energy efficiency. These instructions were of interest to Dolphin, and so the effort to develop version 2 (v2) of the CV32E40P processor began.

II. CV32E40P v2 VERIFICATION CHALLENGES

Although the CV32E40P v2 is a derivative of a previously verified core, the addition of 300 new instructions presents a significant verification challenge. The magnitude of change to the RTL means that the implementation of the previously-supported 120 RISC-V standard instructions also needs to be verified. Adding to this, Dolphin needs to validate seven different configurations of the single core, and two additional configurations in their multi-core platform. Constrained by limited resources, both human and in the form of EDA tool licences, the team adopted a pragmatic approach to verification.

III. FORMAL VERSUS SIMULATION

Considering the challenges described above, Dolphin decided to take a two-faceted approach to verification. Formal verification is being used wherever possible, primarily focusing on RISC-V architectural compliance. In a complementary fashion, they are using simulation-based verification to target complex features such as hardware loops, and scenarios involving external events such as interrupts and debug mode requests. With a limited number of EDA tool licenses, it would be difficult to complete the large number of simulation regression runs required to reach verification closure in a reasonable amount of time. Shifting some verification work to formal will reduce the number of simulation runs and tests required. At the time of this writing, the verification effort is still ongoing in both simulation and formal. As there has been more progress in the simulation effort it will be the focus of the remainder of this paper. Analyzing the overall effectiveness of the two-faceted approach is an interesting area for future work once verification closure has been achieved.

IV. VERIFCATION PLANNING

Determining when the verification effort is complete is a challenge in any custom silicon project, but that challenge is exacerbated for RISC-V. The main source of test stimulus for a microprocessor is the program executing on the core. Considering the number of instructions, sequences of instructions, and permutations of operands the set of potential stimuli is enormous. As previously mentioned, Dolphin needs to verify nine different configurations of the CV32E40P RISC-V core. This configurability is considered to be a strength of the RISC-V ISA, but poses a considerable challenge for design verification teams.

To begin to address this challenge the Dolphin team followed the OpenHW Group's practice of authoring a detailed verification plan that establishes criteria for verification completion, or closure. The presence of a verification plan, and the evidence that it provides about the quality of the verification effort, are a requirement for an OpenHW Group core to achieve the TRL-5 milestone. The CV32E40P v2 verification plans are publicly available on GitHub (4). Later sections of this paper will demonstrate how specific criteria from the verification plans are satisfied using SystemVerilog functional coverage.

V. SIMULATION VERIFICATION METHODOLGY

The CV32E40P v2 verification effort began with a strong foundation. The previous version of this processor had reached verification closure, and achieved TRL-5, through the use of a simulation-only verification strategy. However, that milestone was achieved using an ad-hoc verification methodology involving significant manual effort. The limitations of that ad-hoc approach, and the evolution to a more robust and reusable methodology for RISC-V processor cores is detailed in [5]. As a result, the Dolphin team needed a new testbench for the v2, using

the latest version of the OpenHW CORE-V-VERIF [6] UVM environment, the open standard RISC-V Verification Interface (RVVI) [7], and the ImperasDV RISC-V verification IP [8].

VI. SIMULATION VERIFICATION ENVIRONMENT

Although the CV32E40P v2 required a new testbench, there is one component which is common to both the v1 and v2 verification environments: the Imperas RISC-V processor reference model. Imperas has architected the reference model to support the flexibility and configurability of the RISC-V ISA. The model, shown in Figure 2 below, consists of four parts:

1. The RISC-V base model, which implements the ISA specification in full.
2. 250+ configuration parameters to match the configurability of the ISA.
3. Custom instructions and configuration parameters from RISC-V processor IP vendors.
4. Custom instructions and CSRs implemented by end users.

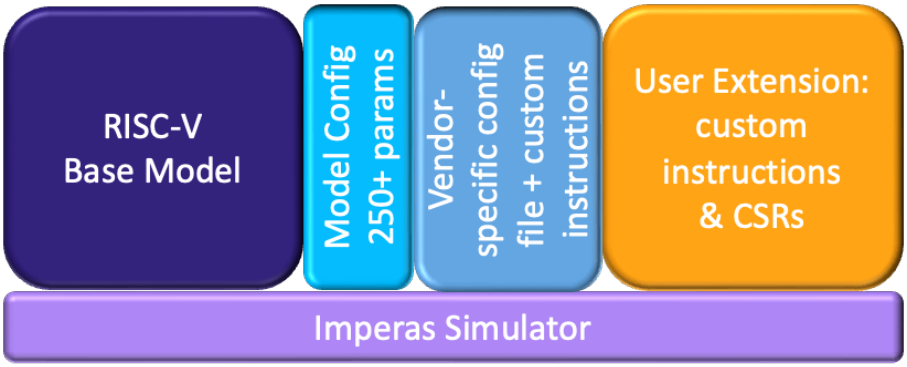

Figure 2: Imperas RISC-V processor reference model

The user extension component of the model is implemented in a separate library from the base model. This is intentional so that user extensions will not perturb the base model or require it to be revalidated. For the CV32E40P v2, the PULP instructions are implemented as a user extension, and thus the original CV32E40P model was reused and extended.

The RISC-V processor reference model is a critical component of the ImperasDV verification IP which is used in the CV32E40P v2 testbench. This testbench, illustrated in Figure 3 below, consists of the following main components:

- The device under test (DUT): the CV32E40P v2 RTL
- The tracer: a SystemVerilog module that sits next to the DUT, whose function is to monitor and extract the internal state of the core and convey this to the testbench using the RISC-V Verification Interface (RVVI)
- RVVI-TRACE: part of the RVVI standard, this is a SystemVerilog interface connecting the tracer to the ImperasDV verification IP
- ImperasDV: this verification IP for RISC-V processors enables the creation of a comprehensive testbench with ease. As each instruction is retired or a significant event (such as a trap) occurs, ImperasDV performs a comparison between the internal state of the reference model and the DUT and reports any mismatch immediately. It uses a proprietary pipeline synchronization technology to handle asynchronous events such as interrupts and debug mode requests allowing those events to be driven randomly during simulations. With an internal scoreboard maintaining metrics about successful and unsuccessful comparisons it enables the users testbench to easily make a pass/fail determination.

With this verification environment in place the Dolphin team began to create directed and constrained-random tests to fulfill the requirements of the verification plan.

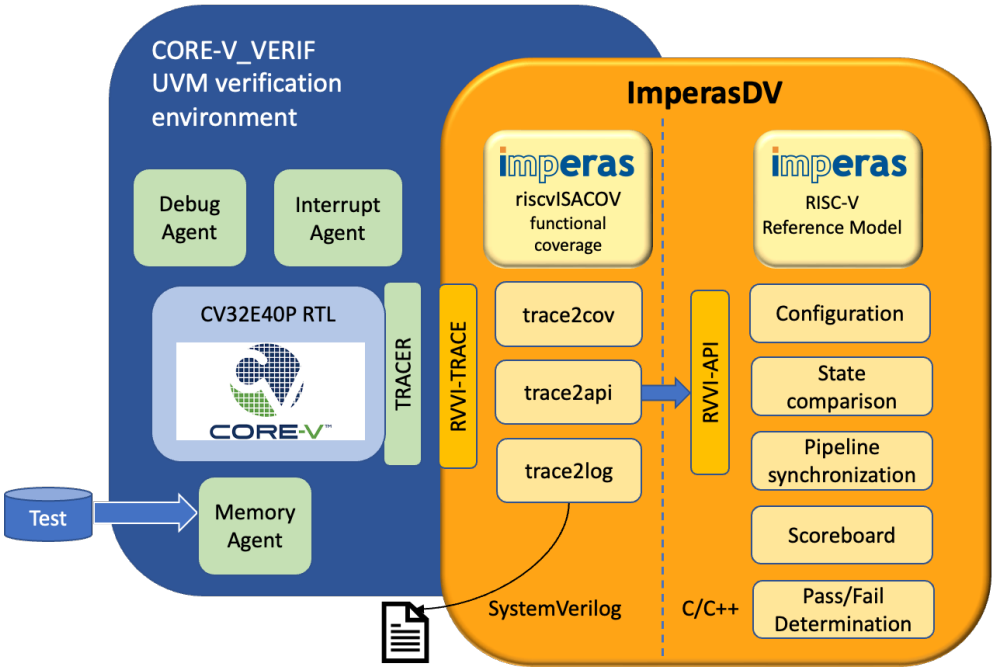

Figure 3: CV32E40P v2 simulation verification environment

VII. SystemVerilog Functional Coverage: Code Generation

Functional coverage for RISC-V processors can be divided into two broad categories. The first is architectural coverage. This is functional coverage that can be directly derived from the RISC-V ISA and covers such things as instructions, their operands, and the values of these. This type of coverage is reusable across different processor implementations because it is defined by a specification. In contrast, the second category is coverage that is particular to a processor implementation. This coverage involves areas such as custom features, instructions, microarchitecture (e.g. branch predictor, pipeline) and corner case scenarios. This second category has limited potential for reuse; however it is just as important as the first.

Consider the first category. The CV32E40P v2 implements the IM[F|Zfinx]C RISC-V extensions, for a total of 120 instructions. Each instruction requires between 10-30 lines of SystemVerilog code to define the coverage model, leading to a total of 3600 lines of code. Writing this code by hand would be tedious and error-prone. Since it can be derived from a specification it is an ideal candidate for automation. With this in mind, Imperas has developed a solution known as riscvISACOV: machine-generated SystemVerilog functional coverage code for the RISC-V ISA. It is created using a machine-readable definition of the ISA as the input to a code generator, the output of which is SystemVerilog functional coverage code. The generated coverage code is organized modularly into separate files, one for each RISC-V extension. This allows the end user to build a custom coverage model by only including the code that matches the combination of extensions implemented by their RISC-V core. The concept of the code generator is illustrated in Figure 4.

Using this same paradigm, Imperas was able to automatically generate the instruction-level coverage model for the 300 PULP instructions, resulting in approximately 9000 lines of code that did not have to be written by hand. This represents a time savings for the Dolphin team as well as the reassurance that the code was error-free and of high quality.

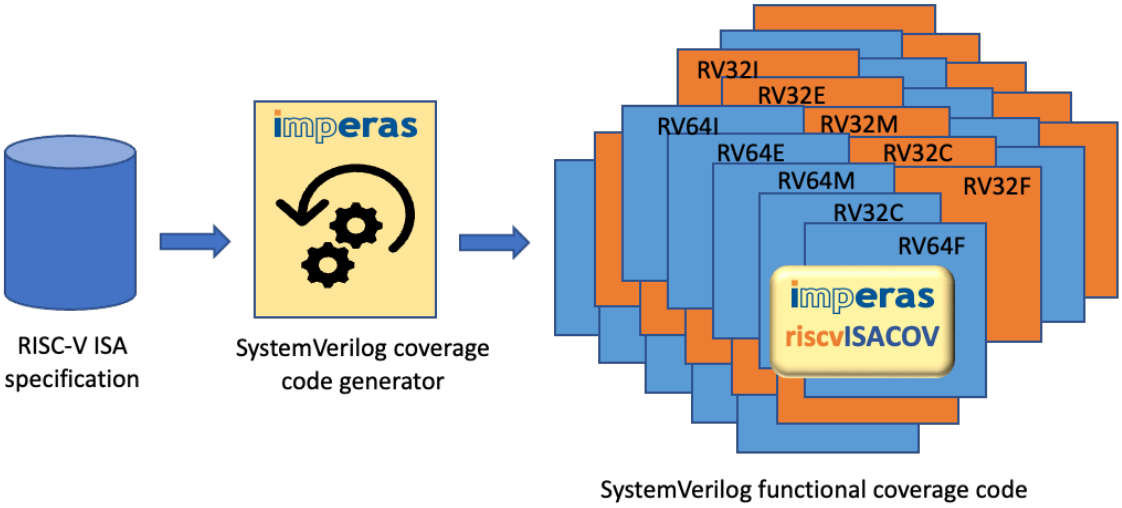

Figure 4: Imperas riscvISACOV generation

VIII. HARDWARE LOOPS

The hardware loops feature was added as part of the PULP / v2 version of the CV32E40P. The concept of hardware loops is well described in the CV32E40P v2 user manual [9]:

> *"Hardware loops make executing a piece of code multiple times possible, without the overhead of branches penalty or updating a counter. Hardware loops involve zero stall cycles for jumping to the first instruction of a loop.*
>
> *A hardware loop is defined by its start address (pointing to the first instruction in the loop), its end address (pointing to the instruction just after the last one executed by the loop) and a counter that is decremented every time the last instruction of the loop body is executed."*

The hardware loops feature is not a candidate for formal verification, thus it is used below as an example of fulfilling verification plan requirements using functional coverage.

IX. EXAMPLE: HARDWARE LOOP INSTRUCTIONS

Eight different instructions can be used in the implementation of a hardware loop. For example, the CV.START instruction is used to set the start address of the loop relative to the current value of the program counter (PC). CV.START uses a register to hold the offset from the current PC value. This is used to load the loop start address register.

The coverage goal identified in the CV32E40P v2 Hardware loop verification plan for the CV.START instruction is to measure that all possible general purpose registers (GPRs) have been used as the RS1 operand. In addition, all bits of each register must have toggled. An excerpt from the verification plan is shown in Figure 5 below.

Requirement Location	Feature	Sub Feature	Feature Description	Verification Goal	Pass/Fail Criteria	Test Type	Coverage Method
CV32E40P User Manual - Chapter 18.3	Hardware Loops Instructions	CV.START	cv.start L, rs1 lpstart[L] = PC + rs1 Loads the lpstart[L] CSR register with current PC value plus an unsigned integer	Register operands All possible rs1 registers are used.	Check against RM	Constrained-Random	Functional Coverage
				coverage: All bits of rs1 are toggled	Check against RM	Constrained-Random	Functional Coverage

Figure 5: CV32E40P v2 Hardware loop verification plan excerpt

Imperas riscvISACOV includes documentation in csv (comma separated value) and markdown format, enabling users to easily understand what coverage is provided without having to read the source code. It also allows the documentation to be imported or copied into the end user's verification plan. Figure 6 below is an example of the riscvISACOV documentation for the CV.START instruction.

Extension	Subset	Instruction	Description	Covergroup	Coverpoint	Coverpoint Description	Coverage Level
XPULPV2	RVXPULPV2	cv.start		cv_start_cg			
					cp_asm_count	Number of times instruction is executed	Compliance Basic
					cp_L	HW Loop L	Compliance Basic
					cp_rs1	RS1 (GPR) register assignment	Compliance Basic
					cp_rs1_sign	RS1 (GPR) sign of value	Compliance Basic
					cp_rs1_toggle	RS1 Toggle bits	Compliance Extended
					cp_rs1_maxvals	RS1 Max values	Compliance Extended

Figure 6: riscvISACOV generated documentation for CV.START instruction

As illustrated in Figure 6, there are more coverpoints present in the riscvISACOV model than are specified in the CV32E40P verification plan. The coverpoints are divided into two categories, or coverage levels: compliance basic and compliance extended. The reason behind this is to provide configurability and flexibility in the coverage model. When an RTL design is unstable and still undergoing frequent change, it does not make sense to review extended coverage results. However basic coverage can still be utilized to assess the quality of the random test stimulus and / or regression test suite.

The riscvISACOV generated SystemVerilog source code for the CV.START instruction is included in the Appendix.

X. SUMMARY AND CONCLUSION

The addition of 300 new custom instructions to an existing RISC-V processor (CV32E40P) presents a significant challenge for Dolphin Design. This challenge is compounded by limited human and EDA tool resources. To address these challenges the team has followed best practices and efficiencies endorsed by the OpenHW Group. These include:

- Authoring a comprehensive verification plan to ensure quality.
- Using verification IP that includes a configurable and extensible processor reference model.
- Using machine-generated functional coverage code to fulfill the requirements of the verification plan.

At the time of this writing, the CV32E40P v2 verification plan has not yet been updated to include riscvISACOV generated documentation, and no coverage results are available for publication. The presentation which accompanies this paper will aim to provide a status update on the verification effort as well as functional coverage results for the hardware loop instructions.

REFERENCES

[1] https://github.com/openhwgroup/cv32e40p

[2] https://www.openhwgroup.org

[3] https://ised-isde.canada.ca/site/innovation-canada/en/technology-readiness-levels

[4] https://github.com/openhwgroup/core-v-verif/tree/cv32e40p/dev/cv32e40p/docs/VerifPlans

[5] L. Moore, A. Sutton, M. Thompson, "The Evolution of RISC-V Processor Verification", DVCon North America, March 2023

[6] https://github.com/openhwgroup/core-v-verif

[7] https://github.com/riscv-verification/RVVI

[8] https://www.imperas.com/imperasdv

[9] https://docs.openhwgroup.org/projects/cv32e40p-user-manual/en/latest/

APPENDIX

SystemVerilog functional coverage code for the CV.START instruction

```
covergroup cv_start_cg with function sample(ins_xpulpv2_t ins);
  option.per_instance = 1;

  cp_asm_count : coverpoint ins.ins_str == "cv.start"  iff (ins.trap == 0 )  {
    option.comment = "Number of times instruction is executed";
    bins count[]  = {1};
  }
  cp_L : coverpoint int'(ins.current.imm3)  iff (ins.trap == 0 )  {
    option.comment = "HW Loop L";
    bins zero  = {0};
    bins one  = {1};
  }
  cp_rs1 : coverpoint ins.get_gpr_reg(ins.current.rs1)  iff (ins.trap == 0 )  {
    option.comment = "RS1 (GPR) register assignment";
  }
  cp_rs1_sign : coverpoint int'(ins.current.rs1_val)  iff (ins.trap == 0 )  {
    option.comment = "RS1 (GPR) sign of value";
    bins neg  = {[$:-1]};
    bins pos  = {[1:$]};
  }

`ifdef COVER_LEVEL_COMPL_EXT
  cp_rs1_toggle : coverpoint unsigned'(ins.current.rs1_val)  iff (ins.trap == 0 )  {
    option.comment = "RS1 Toggle bits";
    wildcard bins bit_0_0  = {32'b???????????????????????????????0};
    wildcard bins bit_1_0  = {32'b??????????????????????????????0?};
    wildcard bins bit_2_0  = {32'b?????????????????????????????0??};
    wildcard bins bit_3_0  = {32'b????????????????????????????0???};
    wildcard bins bit_4_0  = {32'b???????????????????????????0????};
    wildcard bins bit_5_0  = {32'b??????????????????????????0?????};
    wildcard bins bit_6_0  = {32'b?????????????????????????0??????};
    wildcard bins bit_7_0  = {32'b????????????????????????0???????};
    wildcard bins bit_8_0  = {32'b???????????????????????0????????};
    wildcard bins bit_9_0  = {32'b??????????????????????0?????????};
    wildcard bins bit_10_0  = {32'b?????????????????????0??????????};
    wildcard bins bit_11_0  = {32'b????????????????????0???????????};
    wildcard bins bit_12_0  = {32'b???????????????????0????????????};
    wildcard bins bit_13_0  = {32'b??????????????????0?????????????};
    wildcard bins bit_14_0  = {32'b?????????????????0??????????????};
    wildcard bins bit_15_0  = {32'b????????????????0???????????????};
    wildcard bins bit_16_0  = {32'b???????????????0????????????????};
    wildcard bins bit_17_0  = {32'b??????????????0?????????????????};
    wildcard bins bit_18_0  = {32'b?????????????0??????????????????};
    wildcard bins bit_19_0  = {32'b????????????0???????????????????};
    wildcard bins bit_20_0  = {32'b???????????0????????????????????};
    wildcard bins bit_21_0  = {32'b??????????0?????????????????????};
    wildcard bins bit_22_0  = {32'b?????????0??????????????????????};
    wildcard bins bit_23_0  = {32'b????????0???????????????????????};
    wildcard bins bit_24_0  = {32'b???????0????????????????????????};
    wildcard bins bit_25_0  = {32'b??????0?????????????????????????};
    wildcard bins bit_26_0  = {32'b?????0??????????????????????????};
```

```
    wildcard bins bit_27_0 = {32'b????0????????????????????????????};
    wildcard bins bit_28_0 = {32'b???0????????????????????????????};
    wildcard bins bit_29_0 = {32'b??0????????????????????????????};
    wildcard bins bit_30_0 = {32'b?0????????????????????????????};
    wildcard bins bit_31_0 = {32'b0????????????????????????????};
    wildcard bins bit_0_1 = {32'b???????????????????????????????1};
    wildcard bins bit_1_1 = {32'b??????????????????????????????1?};
    wildcard bins bit_2_1 = {32'b?????????????????????????????1??};
    wildcard bins bit_3_1 = {32'b????????????????????????????1???};
    wildcard bins bit_4_1 = {32'b???????????????????????????1????};
    wildcard bins bit_5_1 = {32'b??????????????????????????1?????};
    wildcard bins bit_6_1 = {32'b?????????????????????????1??????};
    wildcard bins bit_7_1 = {32'b????????????????????????1???????};
    wildcard bins bit_8_1 = {32'b???????????????????????1????????};
    wildcard bins bit_9_1 = {32'b??????????????????????1?????????};
    wildcard bins bit_10_1 = {32'b?????????????????????1??????????};
    wildcard bins bit_11_1 = {32'b????????????????????1???????????};
    wildcard bins bit_12_1 = {32'b???????????????????1????????????};
    wildcard bins bit_13_1 = {32'b??????????????????1?????????????};
    wildcard bins bit_14_1 = {32'b?????????????????1??????????????};
    wildcard bins bit_15_1 = {32'b????????????????1???????????????};
    wildcard bins bit_16_1 = {32'b???????????????1????????????????};
    wildcard bins bit_17_1 = {32'b??????????????1?????????????????};
    wildcard bins bit_18_1 = {32'b?????????????1??????????????????};
    wildcard bins bit_19_1 = {32'b????????????1???????????????????};
    wildcard bins bit_20_1 = {32'b???????????1????????????????????};
    wildcard bins bit_21_1 = {32'b??????????1?????????????????????};
    wildcard bins bit_22_1 = {32'b?????????1??????????????????????};
    wildcard bins bit_23_1 = {32'b????????1???????????????????????};
    wildcard bins bit_24_1 = {32'b???????1????????????????????????};
    wildcard bins bit_25_1 = {32'b??????1?????????????????????????};
    wildcard bins bit_26_1 = {32'b?????1??????????????????????????};
    wildcard bins bit_27_1 = {32'b????1???????????????????????????};
    wildcard bins bit_28_1 = {32'b???1????????????????????????????};
    wildcard bins bit_29_1 = {32'b??1????????????????????????????};
    wildcard bins bit_30_1 = {32'b?1????????????????????????????};
    wildcard bins bit_31_1 = {32'b1????????????????????????????};
  }
  cp_rs1_maxvals : coverpoint unsigned'(ins.current.rs1_val) iff (ins.trap == 0 ) {
    option.comment = "RS1 Max values";
    bins zeros = {0};
    bins min = {32'b10000000000000000000000000000000};
    bins max = {32'b01111111111111111111111111111111};
    bins ones = {32'b11111111111111111111111111111111};
  }
`endif

endgroup
```