SIEMENS

Dolphin

RISC-V Verification Plan

Revision 1.1
December 2022

Contents

1. Introduction

This document is meant to demonstrate the work done to formally verify Dolphin's RISC-V core using Onespin tool flow.

Seven sections are included in this document.

Introduction, this section.

RISC-V Core, Main Features and Architecture of the design.

OneSpin Overview, Tool Introduction and Features.

RISC-V Formal Verification, Workflow and Configuration Descriptions.

Run Results, Properties Verified and Bugs found.

Formal Coverage Results, OneSpin Quantify results.

Appendices A, B, C, and D, List of documents and references used in verification.

2. RISC-V Core

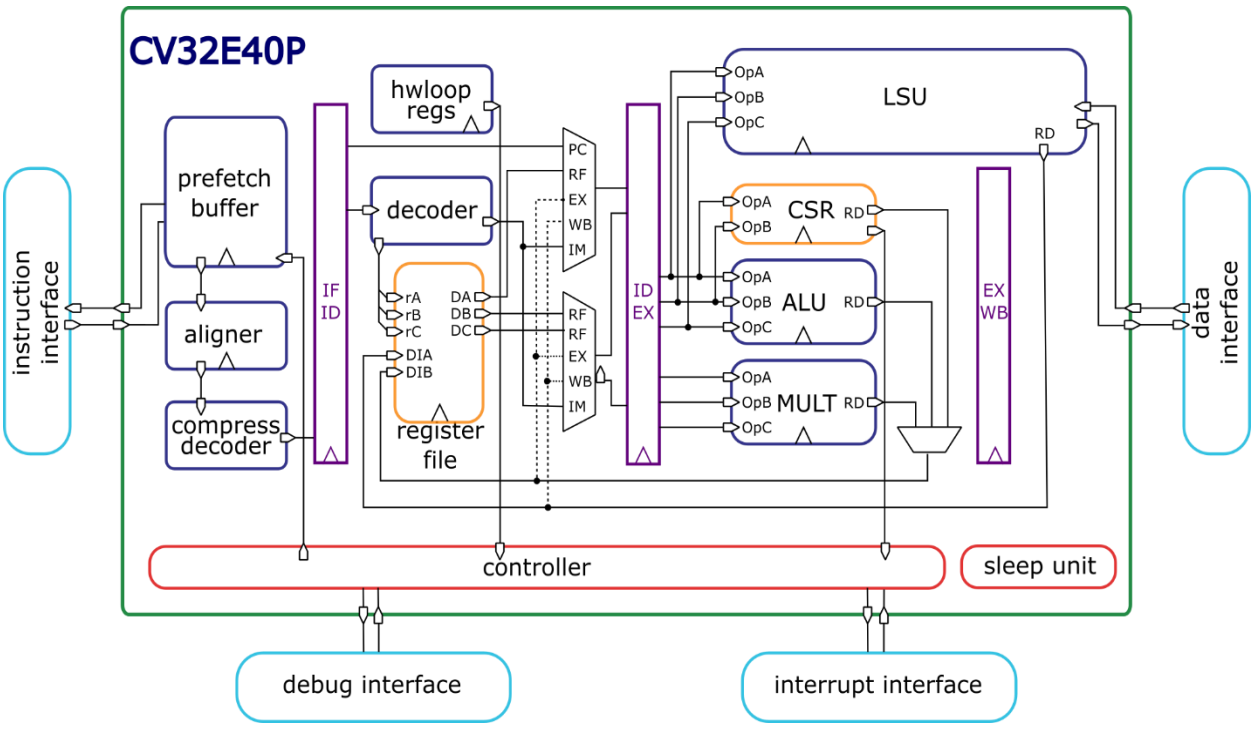
2.1 RISC-V Processor Intro

Pronounced "risk-five", RISC-V is an Instruction Set Architecture based on reduced instruction set computer (RISC) principles. Unlike most other ISA designs, it is provided under open-source license that does not require fees to use. The RISC-V project started in 2010 at the University of California, Berkeley along with volunteer contributors not affiliated with the university.

2.2 CV32E40P Specifications

CV32E40P is a 4-stage 32-bit RISC-V processor core from OpenHW group organization. The ISA of CV32E40P has been extended to support multiple additional instructions including hardware loops, post-increment load and store instructions, additional ALU instructions, and 8 and 16-b SIMD instructions that are not part of the standard RISC-V ISA.

The following figure shows the block diagram of the CV32E40P core.



2.3 Core Parameters

Many features in the RISC-V specification are optional. CV32E40P can be parameterized to enable or disable these features. The core supports *RV32IMCZifencei* Instruction set by default which corresponds to CV32E40Pv1 tag in OpenHW Group Github repository. Other Parameters can be enabled as follows.

2.3.1 FPU

It enables FPU unit which is used to calculate Single Precision Floating point using F registers. The F Instructions can be found in RISC-V Specification and is conform to IEEE 754-2008 standards. Supported operations are compare, min-max, conversions, addition, subtraction, multiplication, fused multiply add, square root and division.

2.3.2 ZFINX

FPU must be enabled first. Setting this parameter allows FPU to use X registers instead of F ones.

2.3.3 XPULP

Custom PULP Extensions and Instructions added that are not part of the original RISC-V ISA. These custom extensions include post-Incrementing load and stores, Hardware Loop extension, ALU extensions, Multiply-Accumulate extensions, Single Instruction Multiple Data (aka SIMD) extensions.

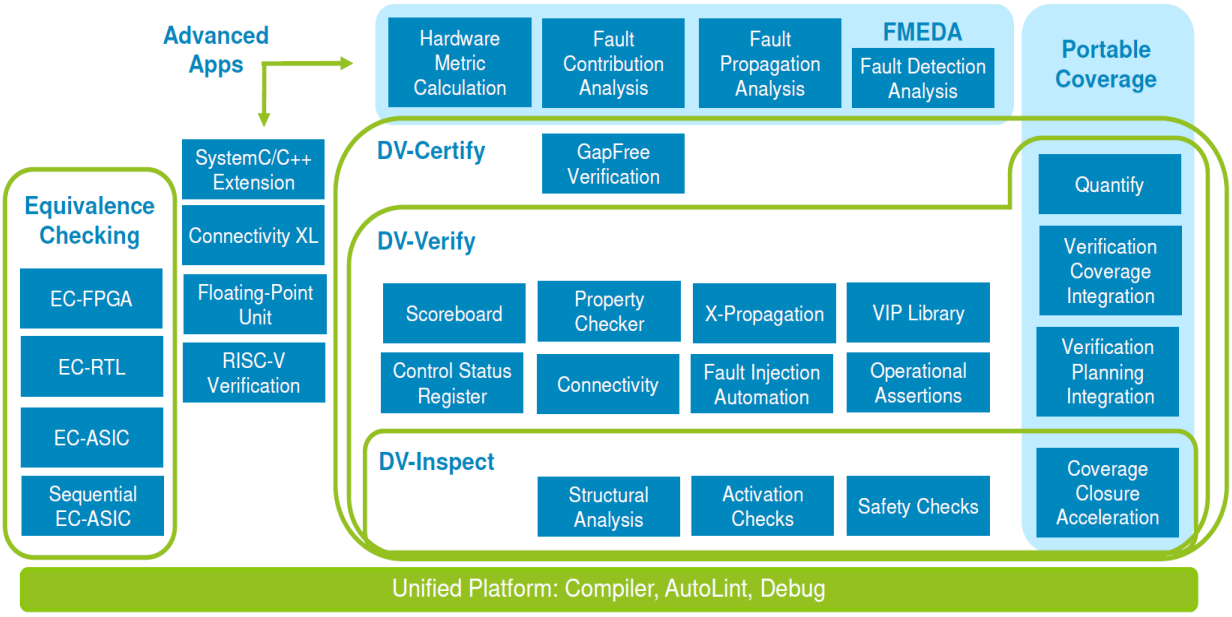
2.3.4 XCluster

Other Custom PULP Instructions. The documentation for both XPULP and XCluster can be found in the Appendix A.

3. OneSpin Overview

3.1 OneSpin Intro

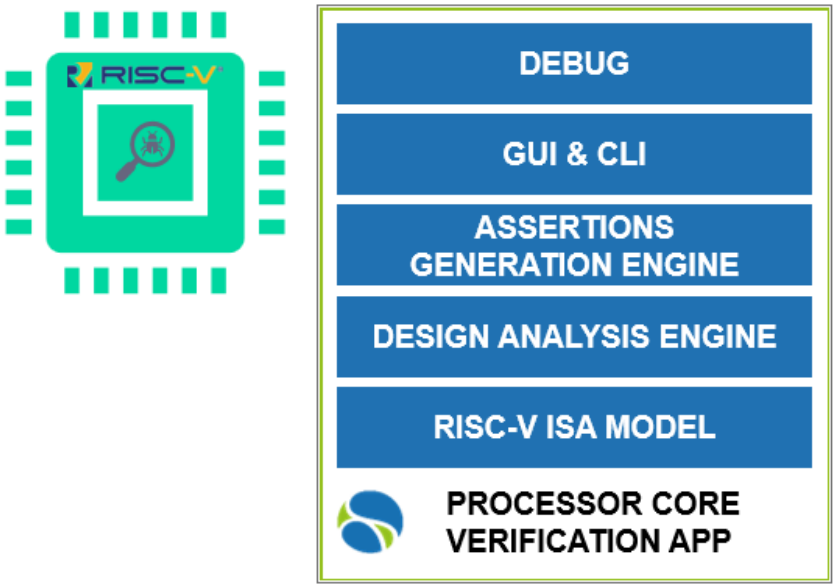
OneSpin is considered the most advanced formal platform available today. It proved its capabilities through top capacity, performance, convergence, and solvers. It has the most advanced formal coverage: Quantify, Operational Assertions, and GapFreeVerification. OneSpin Products span many formal directions from Formal Property checking and Equivalence Checking to Automated Formal Apps such as RISC-V Processor Verification App.



3.2 RISC-V App

The OneSpin Processor Verification App ensures that an IP core implementation does everything it is supposed to do and does not do anything it is not supposed to do. System-on-chip (SoC) designers can license a RISC-V core confident that it complies with the Instruction Set Architecture (ISA) specification, while IP vendors can support their own ecosystems and ensure that partners also comply. Further, SoC designers can add custom features to the RISC-V ISA to support their specific applications.

The OneSpin solution ensures that nothing is broken as features are added and is flexible enough to verify custom instructions and registers. The solution allows faster runtimes than simulation freeing simulation licenses to be used for other tasks. Complete coverage can be achieved with minimal to no use of a testbench. Automated Checker and Automated Custom Extension capability is used to make verification easier, and the time taken to verify the core is much less.



3.3 FPU App

Floating-point is essential for advanced artificial intelligence (AI) applications such as deep learning. FPU Verification faces multiple Challenges:

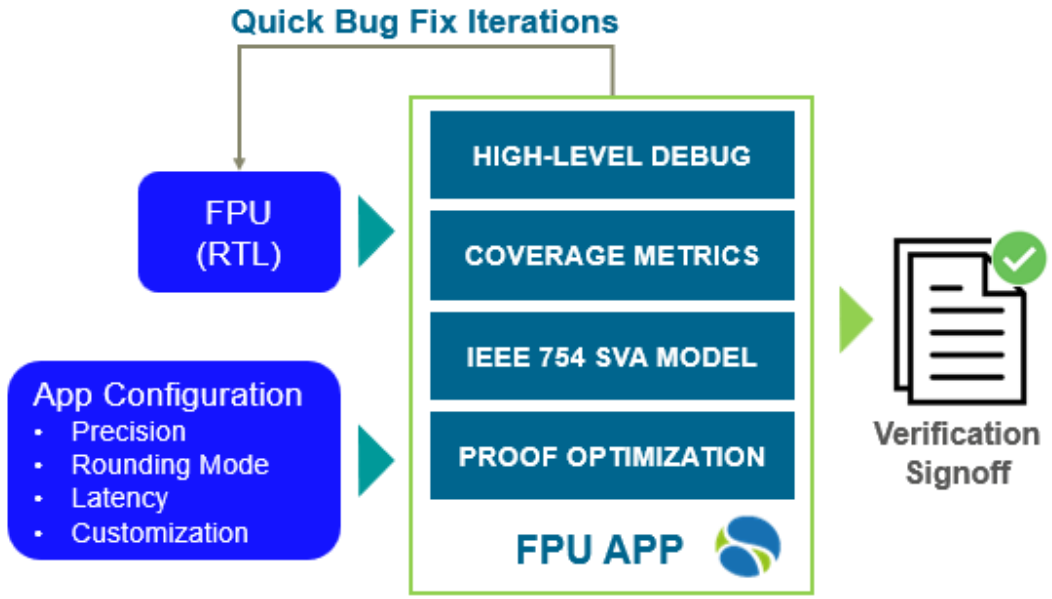
- Complex IEEE 754 floating-point specification
  - Arithmetic and comparison operations
  - Bfloat16, half, single, and double precision
  - Five rounding modes
  - Five exception flags
- Simulation cannot guarantee standard has been met
  - Only a formal App can prove compliance

OneSpin FPU App solution:

- Compliance rules captured using standard SystemVerilog Assertions (SVA)
- Supports all operands, rounding modes, and exception flags
- Highly automated formal proof strategies
- Parallel proof engines with network and cloud distribution
- Floating point value view of operands for debugging
- Integrates with RISC-V F/D extensions

OneSpin FPU App provides an easy to setup interface which supports half/single/double bfloat16 and custom precisions. All five rounding modes and five exceptions flags and arithmetic operations including add, sub, mul, fma, abs, neg, conversion and comparison are supported. There is absolutely no need for C++ model of the FPU with an easier way to model intended deviations from the IEEE-754 standard.

The FPU App has been adjusted to account for different latencies in the FPU design. This can be controlled through setting these two parameters: FPU_OTHERS_LAT, FPU_ADDMUL_LAT. For the scope of this verification plan these two parameters are set to Zero.



3.4 OneSpin Files

3.4.1 Checker file

Provided by OneSpin team, CV32E checker file is a property template file based on SystemVerilog Assertions and TiDAL properties which are used by the assertions generated from JSON files. TiDAL is an abbreviation for Timing Diagram Assertion Library, it is a Library of Constructs and Functions developed by OneSpin team to easily capture timing diagrams using SVA properties.

3.4.2 Setup files

This contains the OneSpin commands used to

- read in the design of a specific configuration
- set and run the processor core verification app
- read the verification files
- specify target properties to check
- choose the appropriate provers settings

3.4.3 JSON files

The JSON files contain

- core specific information
- app parameters
- description and decoding of all custom instructions
- custom instructions execution based on Sail description Language and the specifications of custom CSRs like memory map definition.

Details on custom instructions Sail description can be found in Appendix B.

3.4.4 Log/Output Files

The below files are generated automatically from OneSpin, they include either bindings or information related to the design and the checker.

- bind.sv
- custom_extensions.sv
- RISCV_ISA.sv
- RISCV_disass.tcl

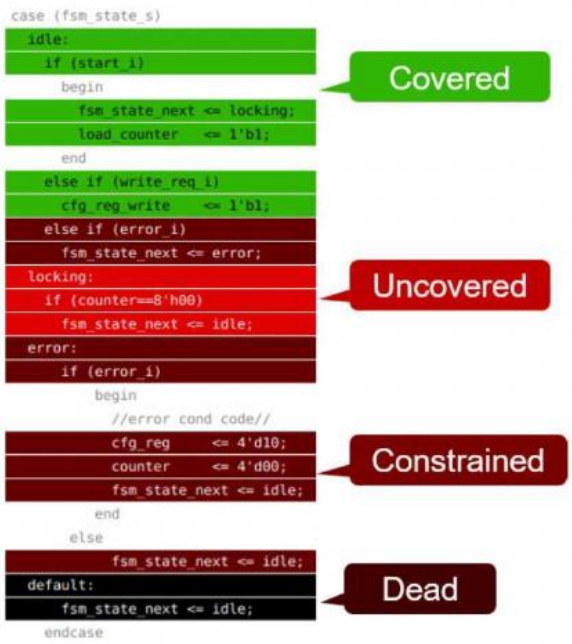
Other generated files.

- msg.log: log file
- default.onespin: OneSpin saved database
- vplan.csv: Verification Plan

3.5 Quantify App

The OneSpin 360 DV Quantify™ App measures the quality of a formal verification testbench. It provides precise, actionable information on what parts of the design-under-test (DUT) are verified, and it highlights RTL code that could still hide bugs.

Additionally, Quantify App reveals potential issues in the testbench that might corrupt metrics and give a false sense of confidence.

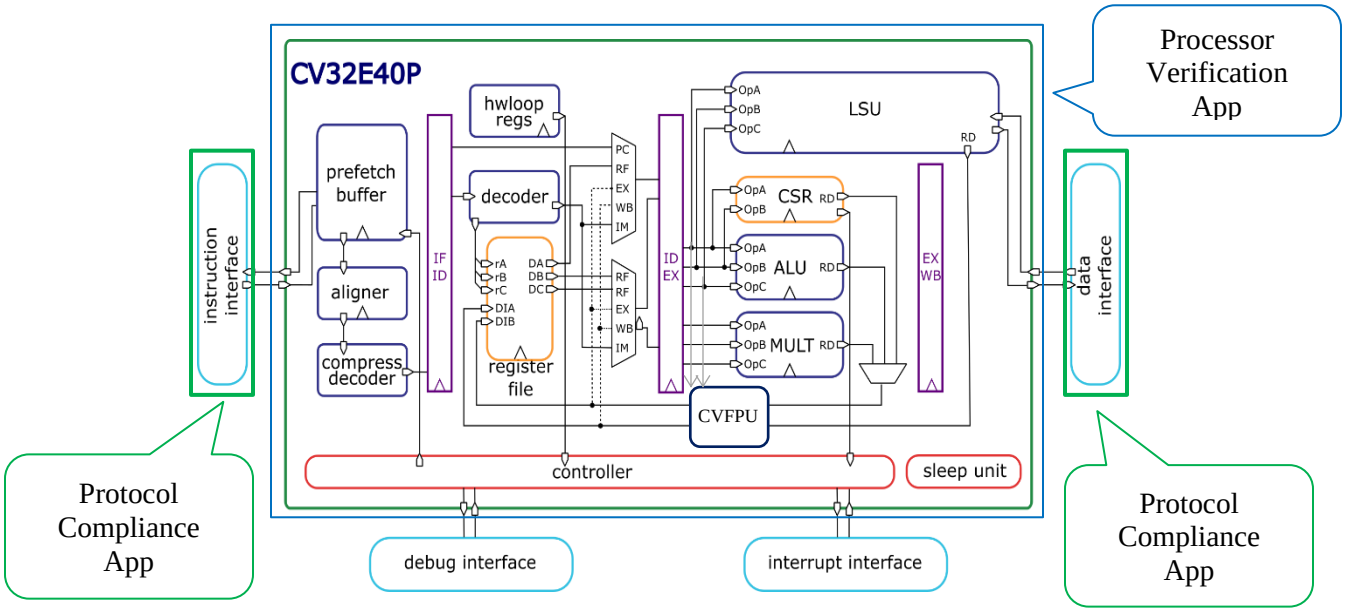


Quantify App produces detailed coverage metrics and a color-coded view of the DUT. The key metrics generated by Quantify App are:

- Covered: DUT code that is verified by at least one assertion
- Uncovered: DUT code that is not verified by any assertion. Uncovered code points to missing or inadequate assertions
- Open: DUT code that has not yet been quantified
- Constrained: DUT code that is dead because of constraints. Over-constraining is a major risk for both random simulation and formal testbenches. Excluding legal DUT input scenarios from verification can lead to silicon bugs. In most projects, constrained code is not acceptable and must be addressed before signoff
- Dead: DUT code that cannot be simulated. Formal engines have proven that no input stimulus exists that exercises this code. This could be due to bugs in the RTL, or to the specific values of configuration parameters. Dead code must be reviewed and waived before signoff

4. RISC-V Formal Verification

4.1 OneSpin Verification Scope



On the OBI memory interfaces level, Protocol Compliance App verifies that the fetch and data memory interfaces behave in correspondence with the bus protocol. On the core level, Processor Verification App verifies all modules that reflect pipeline's behavior after applying the reset sequence, once there is nothing being executed, i.e., cases of stalls or having no valid instruction in ID stage (Bubble state), and once an instruction is being executed or an exception is being taken, including if the hart is running in debug mode or enters debug mode due to ebreak forced entry or instruction address debug breakpoint exception.

Global constraints are added to control memory interfaces behavior regarding slave responses, assuring that slave modules behave according to the OBI protocol and respond to master requests in a bounded time. Other global constraints are related to having stable values for primary input address signals. Furthermore, depending on the parameters set or the configuration chosen, it is assumed that clock gating enable signal is always set and the signal indicating that the core is running in HW loop body is always unset. Finally, there are additional local constraints added as workarounds for the bugs found, these constraints are safe to be removed once bugs are fixed.

Overall verification focuses on the architecture state, that is the combination of integer (X) and floating-point (F) register files, control and status registers (CSRs), and program counter (PC). In addition, the data memory interface and the conceptual state, a state that reflects pipeline readiness to execute an instruction, are part of the verification targets as well.

Upon reset, if fetching instructions is enabled, there are no debug requests, and reset is inactive, it is verified that the first instruction is fetched from the correct boot address, all X and F registers are valid, CSRs are set as the RISC-V specification states, no unexpected data memory requests are set, and the fetched instruction is ready to be executed.

Once the pipeline is in a bubble state and there are no interrupt, or debug requests that are active, it is then verified that the architecture state is not updated, i.e., register files, CSRs, and PC are not updated, and no data memory requests are set. Moreover, it is verified that once the core is no longer in a bubble state, the next instruction is ready to be executed.

Instruction verification assumes that an instruction is valid in ID stage (this instruction is registered whenever there is a valid instruction in IF stage, thus the instruction before being decompressed is captured, which allows verifying compressed instructions), while the ID stage is not stalled, the EX-stage is ready to execute it, and there will be no interrupt, exception or debug requests that are going to interrupt its execution. It is then verified that the architecture state is updated as expected, i.e., the respective destination registers of register files are updated with the expected values and the other registers keep their values unchanged, the respective CSR counters are incremented as expected, and the next instruction address is the correct one. In addition, it is verified that data memory signals are set as expected if the instruction is load or store one, and that no data memory requests are set by non-memory instructions. Furthermore, it is verified that once the instruction retires, the next instruction is also ready to be executed.

It is worth noting that, FDIV, FSQRT, and HW loop custom instructions are not verified at all. Also, no immediate preceding multicycle F instructions are allowed while verifying F instructions. Furthermore, fflags CSR update is not checked in case of a preceding multicycle F instruction and updates of the custom CSRs, lpstart0, lpend0, lpcount0, lpstart1, lpend1, lpcount1, are not checked anywhere. In addition, result verification of multiplication and division instructions is limited, i.e., only certain bits of the operands are allowed to toggle, the other bits can be all 0s or all 1s. (Except the FDIV, FSQRT, and bounded multiplication and division result constraints, it is expected to remove the other constraints with future verification deliveries).

Exception verification assumes that an instruction is valid in ID stage, while the ID stage is not stalled, the EX-stage is ready to execute it, and there will be no interrupt, or debug requests that are going to interrupt its execution. However, there will be an exception raised by the instruction itself. It is then verified that the architecture state is updated as expected, i.e., register files are not updated, the respective CSRs are updated as expected, and the next instruction address is the correct one. In addition, it is verified that no data memory requests are set by non-memory instructions. Furthermore, it is verified that once the exception is taken, the next instruction is also ready to be executed.

For detailed description of all verification targets and constraints applied, refer to the actual verification plan excel sheets.

Verification Plan Sheet is referenced in Appendix D.

4.2 Main Configurations

The following is the list of main core configurations to validate. Each configuration is followed by an integer number that is used to run this configuration using Makefile option “CONFS”. The configurations from 1 to 12 have FPU Latency parameters specified as (FPU_OTHERS_LAT = 0, FPU_ADDMUL_LAT = 0).

The following configurations are the ones chosen to run as they are most conclusive to all properties and parameters.

4.2.1 IMC (1)

These are the properties that were verified in CV32E40Pv1. It is the Base Instruction set in RISC-V Specification including Zicsr, Zifencei, and M Instructions.

4.2.2 XPULP+XCLUSTER (4)

Properties that are enabled with PULP_XPULP=1 and PULP_CLUSTER=1.

4.2.3 FPU+ZFINX+XPULP+XCLUSTER (6)

Properties that are enabled with FPU=1, PULP_XPULP=1, PULP_CLUSTER=1 and PULP_ZFINX=1.

4.2.4 XPULP (7)

Properties that are enabled with PULP_XPULP=1.

4.2.5 FPU+XPULP (8)

Properties that are enabled with FPU=1, PULP_XPULP=1.

4.2.6 FPU+ZFINX+XPULP (9)

Properties that are enabled with PULP_XPULP=1, FPU=1 and PULP_ZFINX=1.

4.2.7 FPU+ZFINX+XPULP+XCLUSTER with Dynamic FPU Latencies (13) – Optional Configuration

Properties that are enabled with FPU=1, PULP_XPULP=1, PULP_CLUSTER=1 and PULP_ZFINX=1 but with FPU_OTHERS_LAT = 5, FPU_ADDMUL_LAT = 5.

4.3 Custom Extensions Decoding

4.3.1 OpenHW Specification

As mentioned earlier, the custom extensions and instructions are available with their encoding, description, and behavior on GitHub CV32E40P repository. They include post-incrementing load and stores, Hardware Loop extension, ALU extensions, Multiply-Accumulate extensions, Single Instruction Multiple Data (aka SIMD) extensions.

4.3.2 OneSpin JSON

Two main inputs are needed for OneSpin Processor App to work successfully, the checker and the JSON file that has the Instructions and their behavior. It acts as the golden model for these Instructions. The JSON file contain Custom CSR definitions as well and their mapping to the RTL.

4.3.3 XPULP Instructions Sail Description

The Instructions behavior and encoding are mapped from the GitHub repo to the JSON file. The figure here shows an example of how the instructions are written.

```
{
 "name": "CV.MAC",
 "disassembly": "cv.mac {rd},{rs1},{rs2}",
 "decoding": "1001000 rs2 rs1 011 rd/rs3  0101011",
 "restrictions": "X(rs2) != 32",
 "execution": "X(rd) = X(rs3) + (X(rs1) * X(rs2))[31..0]"
 }
```

The new Instructions were around 350 Instructions, and all had to be written like the above. Note: rd/rs3 means that they are the same data but the distinction just for to differentiate between writing and reading the register.

There are some Sail description functions that are already pre-defined inside OneSpin. User can refer to them in Appendix B. The exact Sail Description file for the custom Instructions included in this project, will be in Appendix D. Other information regarding Sail Description can be found in the OneSpin User Manual.

4.4 XPULP Instructions Validation

The golden model (JSON file) is an input to OneSpin. Running the tool can uncover mismatches between the RTL and the JSON file. These mismatches are debugged to determine if they are: Specification mismatches, RTL Implementation bugs or simply the JSON file needs more tuning in either the decoding or the Sail description.

The debugging in OneSpin Processor Apps is either by checking the waveform or from the shell directly where user can see the result of RTL implementation and the Sail description and compare which is correct. Taking for example a simple Instruction CV.EXTHZ, if modified its sail description to a counter example like this which will produce wrong result .

```
{
"name": "CV.EXTHZ",
"disassembly": "cv.exthz {rd},{rs1}",
"decoding": "0110001 00000 rs1 011 rd  0101011",
"restrictions": "",
"execution": "X(rd) = EXTS(X(rs1)[15..0])"
},
```

OneSpin Output:
-R- Property 'core_i.RV_chk.RV32X.CV_EXTHZ_a' fails within 5 cycles from reset

User can follow one of two approaches to debug.

1. Shell Analysis
   From the GUI, user can right-click on the property and click “Analyze Processor Verification counterexample”. User can also do this from the shell with the following command.

   processor_integrity::analyze_trace {core_i.RV_chk.RV32X.CV_EXTHZ_a}

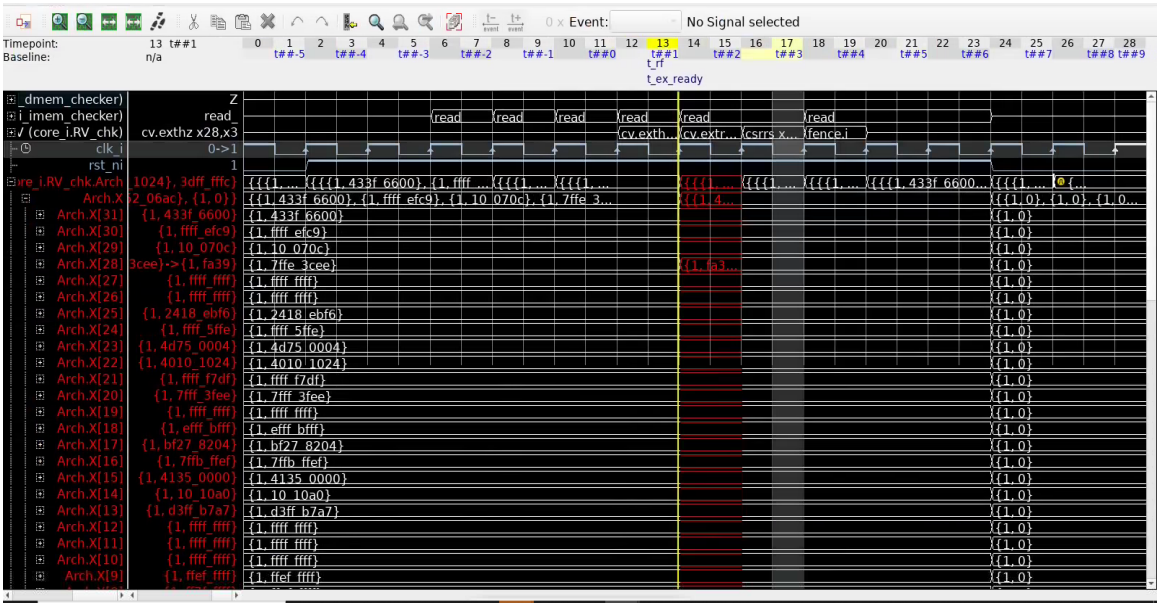
   OneSpin Output:

```
-I- Counterexample for property 'core_i.RV_chk.RV32X.CV_EXTHZ_a' generated.
-I- TraceAnalysis - Current architecture frozen at 't_rf'
-I- TraceAnalysis - Analysis of counterexample 'core_i.RV_chk.RV32X.CV_EXTHZ_a'
updating register file at time 13 (t##1, t_rf):
current instruction:
       cv.exthz x28,x3 (PC 0x3dfffff8) at time 13 (t##1, t_rf) (0x6201be2b,
0b01100010000000011011111000101011)
decoding fields (freeze variable exe.dec):
        instr: CV_EXTHZ
  ISA decoding: 0110001 00000 rs1 011 rd  0101011
        RS[1]: 3
        RD[1]: 28
        width: 0x4
register file source operands (freeze variable exe.op):
     1: X[ 3]: 0x6600fa39 (1711340089)
register file results (freeze variable exe.result):
    ISA X[28]: 0xffffa39 (-1479)
    DUT X[28]: 0x0000fa39 (64057) (unexpected)
ISA execution for cv.exthz x28,x3 with current register values:
       X(28) = EXTS(0xfa39)
DUT register file updates (freeze variable cur_Arch and signal core_i.RV_chk.Arch at time 15
(t##2)):
        X[28]: 0x7ffe3cee -> 0x0000fa39
-I- TraceAnalysis - pipe_result fails in property debugger!
-I- TraceAnalysis - pipe_rf_check fails in property debugger!
```

```
-I- TraceAnalysis - rfs_equal fails in property debugger!
-I- TraceAnalysis - ==_.X[28].data (expected value 0xfffffa39) fails in property debugger!
```

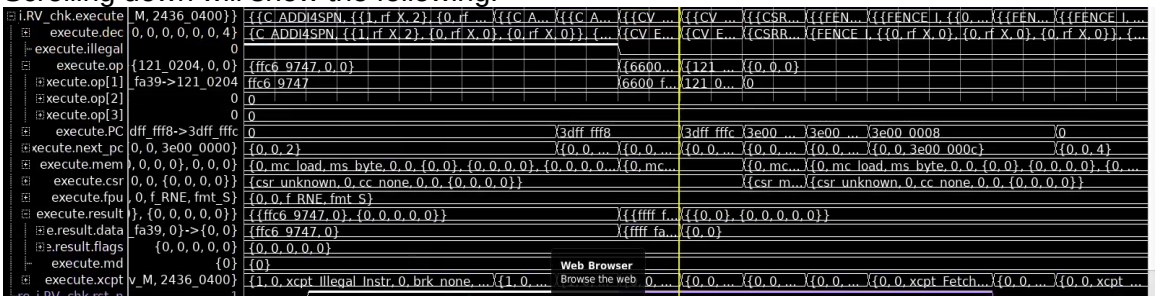
The values highlighted in red are the ISA and DUT values which are the Sail and RTL calculated values. Here user can check either the RTL or Sail results are wrong.

2. Waveform checking
   This would be helpful to check the previous values or if more comprehensive debugging is needed. User can right-click on the property and click “Debug”, detach the window from “Detach window” icon.



Here user can check the instruction and which registers are written, also all the values in the registers in the design.

Scrolling down will show the following.



The RV_chk.execute represents the supposed values in the registers, so user can check the operands and also the result of the sail calculation.

4.5 Run Flow

A Makefile has been developed to automate the OneSpin Setup and Property Checking. It includes all customization options. Makefile options can be found in Appendix C.

4.5.1 Automatic Property Checking

RISC-V CV32E40P has various core configurations. For Each Configuration a set of properties are enabled. File lists that contain list of each configuration properties are provided and then they are combined if the user would like to check different configurations.

Each configuration runs with four variances

1. No PVE_FPU_SUPPORT or Multiplication
2. PVE_FPU_SUPPORT flag
3. Multiplication but without PVE_M_SUPPORT flag
4. Multiplication with PVE_M_SUPPORT flag

The following table explains the intention of each run variance.

1	cuts FPU result and flags signals, as well the multiplier and divider result signals, which makes it easy to run the non-F or multiplication assertions. Fails are expected in case F or multiplication assertions are run due to cuts and non-complete app setup for F, as the PVE_FPU_SUPPORT isn't defined.
2	doesn't cut the FPU result and flags signals. It also adds the PVE_FPU_SUPPORT to the set of defines, and thus individual F assertions are generated and can be run
3	doesn't cut the multiplier or divider result signals and thus running M, or X multiplication assertions doesn't produce unexpected fail due to the cuts. It combines many M instructions in one assertion, like RV32M.MULH_a verifying (MULH, MULHSU, MULHU) instructions. What is verified with these assertions, is everything except the result itself.
4	doesn't cut the multiplier or divider result signals and thus running M, or X multiplication assertions doesn't produce unexpected fail due to the cuts. It also adds the PVE_M_SUPPORT to the set of defines, and thus individual assertions for each of M instructions are generated, verifying the actual results

These flags are also needed in the checker to specify which properties are enabled and hence User must specify RUNS and CONFS in the make command (Check Appendix C). Also, custom file is provided if Only Specified Properties are needed to run.

4.5.2 Property Status Script

This is a python script to display the status of the run based on the log file produced by OneSpin To run this, go to logs folder and then run command: *python3 result.py.*

4.5.3 Prover Configurations

OneSpin has different engines of provers and disprovers based on different algorithms. For this project, 2 sets of provers and disprovers are used based on the need and the computational

power and time resources and limitations. Each configuration is enabled by a OneSpin command:

1. Provers Configuration: *set_check_option -prover_exec_order { { prover2:0 prover2:3 prover2:8 prover2:11 } }*
2. Disprover Configuration: *set_check_option -prover_exec_order { { approver1 approver2 prover2 approver4 disprover1 } } -disprover1_steps 40 -disprover3_steps 40*

User must specify PROVE option in make command to choose either configuration.

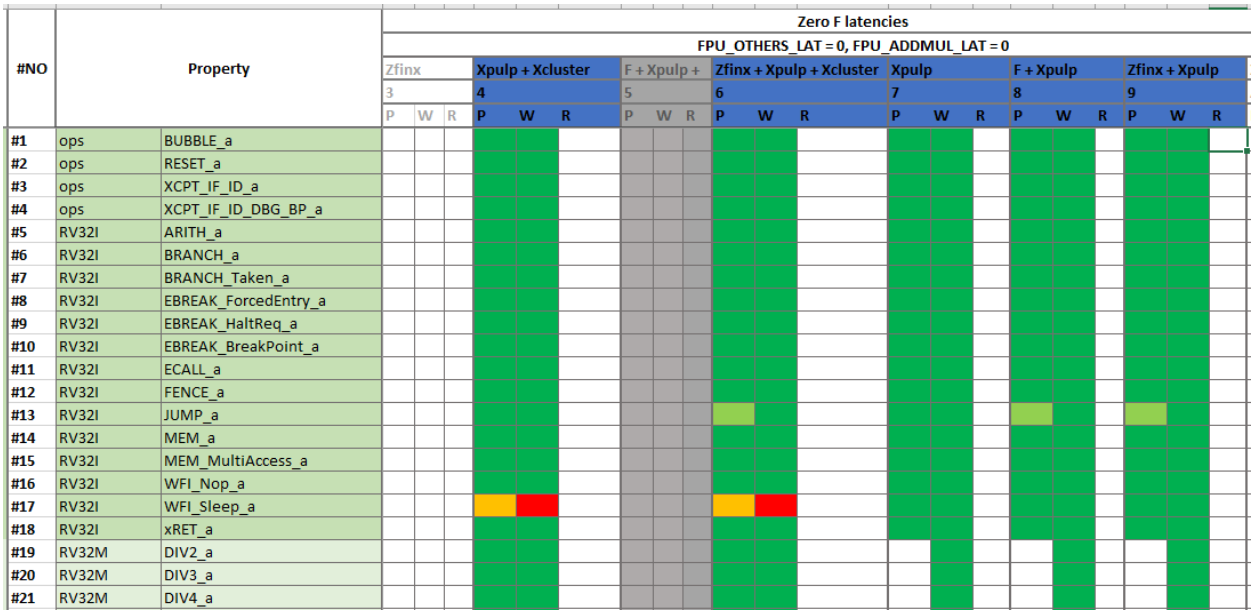
4.5.4 Setup Network

Although this option was not used by Dolphin, this can be beneficial in case Multiple machines are used. User can modify setup_network function inside setup script to add different servers and machines.

5. Run Results

Run results of target core configurations are summarized in a Property Status sheet that shows status of each property whether failing or passing. This sheet has been updated throughout the project timeline. For properties classified as bugs a bug report is provided with details on each bug.

Property Status & bug report sheets are referenced in Appendix D.



#NO	Property		Zero F latencies																			
			FPU_OTHERS_LAT = 0, FPU_ADDMUL_LAT = 0																			
			Zfinx			Xpulp + Xcluster			F + Xpulp +			Zfinx + Xpulp + Xcluster			Xpulp			F + Xpulp			Zfinx + Xpulp		
			3			4			5			6			7			8			9		
			P	W	R	P	W	R	P	W	R	P	W	R	P	W	R	P	W	R	P	W	R
#1	ops	BUBBLE_a																					
#2	ops	RESET_a																					
#3	ops	XCPT_IF_ID_a																					
#4	ops	XCPT_IF_ID_DBG_BP_a																					
#5	RV32I	ARITH_a																					
#6	RV32I	BRANCH_a																					
#7	RV32I	BRANCH_Taken_a																					
#8	RV32I	EBREAK_ForcedEntry_a																					
#9	RV32I	EBREAK_HaltReq_a																					
#10	RV32I	EBREAK_BreakPoint_a																					
#11	RV32I	ECALL_a																					
#12	RV32I	FENCE_a																					
#13	RV32I	JUMP_a																					
#14	RV32I	MEM_a																					
#15	RV32I	MEM_MultiAccess_a																					
#16	RV32I	WFI_Nop_a																					
#17	RV32I	WFI_Sleep_a																					
#18	RV32I	xRET_a																					
#19	RV32M	DIV2_a																					
#20	RV32M	DIV3_a																					
#21	RV32M	DIV4_a																					

The referenced run results are based on the RTL version provided on 12 October 2022 and Core Specification provided on 7 October in 2022.

6. Formal Coverage Results

6.1 Quantify App Flow

To be able to run the Quantify flow, The user needs to run this command: *make all QTF=1*
This enables OneSpin Command *quantify -assume_hold -html qtf_results.*

6.2 Demo Run Results

The figure shows here part of the output of Quantify Coverage results. The outputs are shared in cv32e40p-formal repo inside qtf-results folder. The outputs are in HTML form so it can be viewed using any web browser.

The following screen capture shows coverage report generated by Quantify App for IMC base configuration.

Structural Coverage Overview — Simplified View

Status		Statements		Branches	
D	detected	841	45.02%	545	53.64%
N	undetected	317	16.97%	260	25.59%
O	open	710	38.01%	211	20.77%
Sum	quantify targets	1868		1016	

Excluded Code Overview — Simplified View

Code Status		Statements		Branches	
X	excluded	194	9.41%	7	0.68%
D/N/O	quantify targets	1868	90.59%	1016	99.32%
Sum	total code	2062		1023	

Structural Coverage by File — Simplified View

File	Statements	Branches
cv32e40p_aligner.sv	62	21
cv32e40p_alu.sv	248	125
cv32e40p_alu_div.sv	51	11
cv32e40p_compressed_decoder.sv	51	81
cv32e40p_controller.sv	358	149
cv32e40p_core.sv	18	0
cv32e40p_cs_registers.sv	223	112
cv32e40p_decoder.sv	284	150
cv32e40p_ex_stage.sv	53	17
cv32e40p_ff_one.sv	7	0
cv32e40p_fifo.sv	32	18
cv32e40p_id_stage.sv	276	105
cv32e40p_if_stage.sv	45	32
cv32e40p_int_controller.sv	43	37

6.3 UCDB File Generation

After the quantify run is done, user can run this command *export_quantify_coverage* inside OneSpin shell to export a UCDB file that contains formal coverage database that can be viewed by other simulators and merged with other UCDB files.

8. Appendix B: Sail Constructs

Here is a table that summarizes main constructs of the Sail Language:

Literals	42, 0xdecafbad, 0b10111, true, false, bitone, bitzero
function call	f(a,b)
record access	a.b
array index	a[4]
array slice	a[4..2]
bitvector concatenation	a@b
bitvector arithmetic	a+b, a-b, a*b, a/b, a%b
logical operators	a&b, a|b, aˆb
logical negation	~(a) ~ is a function with mandatory parenthesis for arguments, not a unary operator
shift	a>>b, a<<b
arithmetic shift right	shift_right_arith32(word,distance), shift_right_arith64(word,distance)
equality	a==b, a!=b
signed comparison	a>=_s b, a=_u b, a<=_u b
assignment	a=b
intermediate variables	let addr : xlenbits = RS1+imm
sign extension	EXTS(a)
zero extension	EXTZ(a)
if	if cond then exp, if cond then exp else exp
register file access	X(rs1), F(rd) No special treatment for X(0) is needed either for reading or writing as X(0) already models the intended semantics of always being 0

More on Sail description can be found inside OneSpin User Manual.

9. Appendix C: Makefile Options

make all <option1>=<value>	
Options	**Description**
GUI	0: Batch mode (default) 1: Interactive mode
CUSTOM	0: Using generated properties based on the configuration and run (default) 1: Using Custom.flists file to check specific properties automatically
CONFS	Specify certain configuration, takes values from 1 to 15
RUNS	Specify certain runs, takes values from 1 to 4
DBG	0: Makes the property run to hold Any other value: The number of seconds each property will run.
VERBOSE	0: Don't Show progress information messages (Default) 1: Show them
PROVE	0: Use the disprovers 1: Use the provers
SERVER	(For Mentor) SC: Setup network for Secure Cell WV: Setup network for WV Machines VP: Setup network for VP Machines
QTF	0: default 1: Run Quantify
AUTO	0: default 1: Run auto checks