Verifying PULPino RISCY Core for a Google Accelerator with STING

Shubhodeep Roy Choudhury[1], Shajid Thiruvathodi[2], Vaidyanathan Seetharaman[3], Matt Cockrell[4], Jon Michelson[5], Jason Redgrave[6]

Valtrix Technologies Private Limited, Bangalore, INDIA[1,2]
Google Inc., Mountain View, USA[3,4,5,6]

deepsrc@valtrix.in[1], sthiruva@valtrix.in[2], vaidy@google.com[3], mcockrell@google.com[4], jonmichelson@google.com[5], jredgrave@google.com[6]

***Abstract*— Google uses the PULPino RISC-V core RISCY as a job scheduling and dispatch mechanism for a hardware accelerator (similar to a GPU controller). This requires full compliance with the RISC-V RV32I base integer instruction set, standard extensions for integer multiplication and division ('M') and compressed instructions ('C'), and capability to handle interrupts from multiple sources. We used STING, a software-driven verification solution for RISC-V based CPU/SoC implementations, to verify the architectural compliance and functionality of the RISCY core. This verification effort helped uncover 30 RTL, documentation, and toolchain issues in the relatively mature RISCY implementation.**

***Keywords*— RISC-V, PULPino, RISCY Core, STING, System-on-Chip, Design-Under-Test, Functional Testing, Design Verification**

1. Introduction

A job scheduler configures an accelerator device, puts itself to sleep, and then awakens to service interrupt requests. Since it is only executes first party code, any instruction set that has a robust toolchain can be used. RISC-V is an attractive option, but available open source implementations have not been used in as many designs as their commercial counterparts and have not, in general, been subjected to the same level of verification. This paper describes one attempt to bridge this verification gap.

User-specific accelerator jobs are passed from the host processor through shared data structures. An ISA with integer computation is required to properly process shared job descriptors. Also, the ISA should have a compressed instruction format to minimize the memory footprint for executable storage. Full ISA compliance is necessary to utilize the latest upstreamed tool chain, in this case RV32I [4]. The core will have access to full system memory that will contain executable firmware in addition to accelerator configuration data. Full interrupt capability at the machine privilege level is needed to pass notification from the accelerator to the core of timely status changes. This paper describes the comprehensive verification flow setup using STING [7] in order to test this functionality.

The rest of paper is organized as follows. Section 2 and 3 present an overview of the PULPino platform and STING design verification tool respectively. Section 4 describes the steps we took to bring up the tool and verify the core. Sections 5 and 6 discuss the results in terms of coverage and the functional issues found in the design, and Section 7 summarizes the paper and concludes.

2. Overview Of PULPino Platform And RISCY Core

PULPino [2] is an open-source, single-core microcontroller system, based on PULP platform [1] developed by teams from ETH Zurich, Switzerland and University of Bologna, Italy. It implements several ideas developed through research on ultra-low power parallel processing, and has a rich set of peripherals and full debug support. PULPino can be configured to use different 32-bit RISC-V core configurations depending on the application use-case.

The core configuration Google uses, RISCY [3], is a highly power efficient, 4-stage, in-order, single-issue, 32-bit RISC-V core designed for DSP applications. The core has complete support for the RV32I base integer instruction set and extensions for compressed and integer multiplication/division instructions. It also implements several other custom extensions such as hardware loops, post-incrementing load and store instructions, bit manipulation instructions and MAC operations.

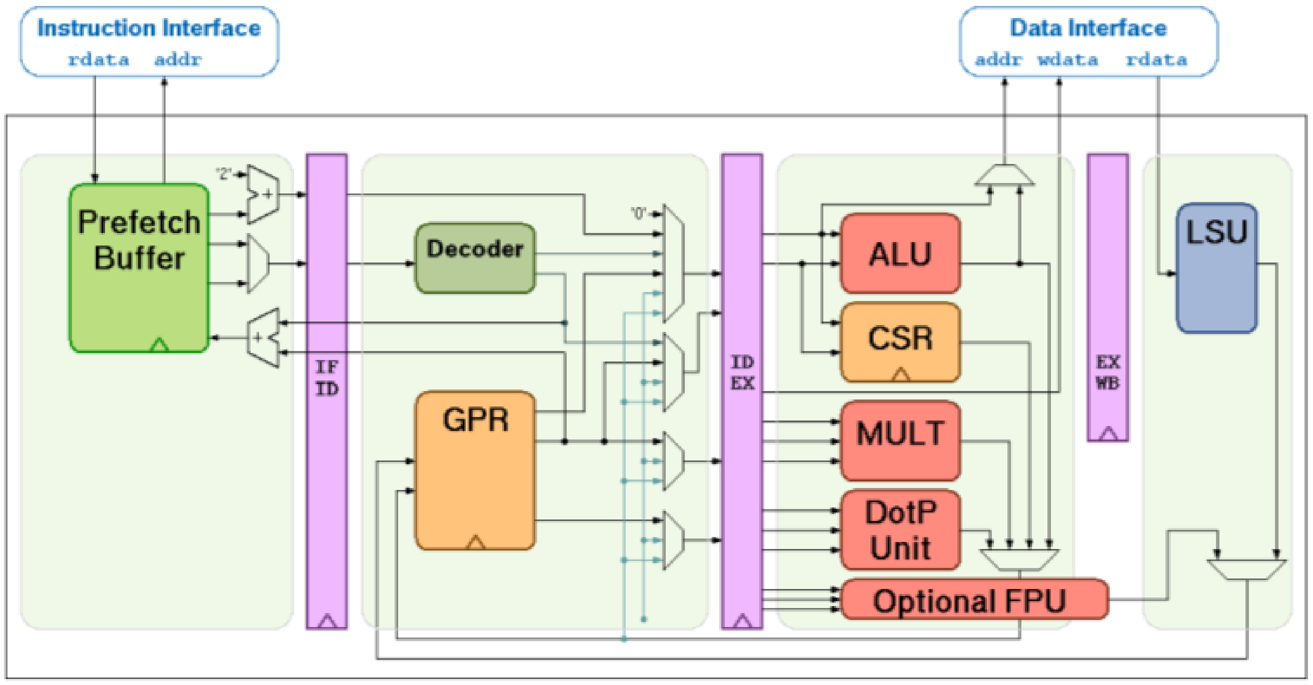


Figure 1: Block diagram of RISCY core

RISCY includes a small subset of the privileged specification and CSRs (control and status registers), which keeps the footprint of the core small while allowing embedded operating systems such as FreeRTOS to run. The design is available for RTL simulation and FPGA mapping and has been taped out in an ASIC before.

The PULPino SoC also supports a number of peripherals, including Timer, Event Unit and UART, which were used in STING to test the scenarios requiring external interrupts.

3. Introduction To STING Design Verification Tool

STING is a proprietary design verification tool for RISC-V based implementations developed by Valtrix Technologies Private Limited, India. STING is a light-weight software program (similar to an embedded operating system) which can be used to generate and execute different workloads (directed, algorithmic or random) on the device-under-test (DUT) for verifying architectural compliance and functional testing [7]. It embodies a software driven test generation methodology which allows portability of test stimulus throughout the life cycle of a SoC design.

The hardware resource utilization and instruction and memory footprint of the test can also be easily controlled using configurable parameters, making it easy to target any SoC configuration from IoT microcontrollers to multi-core servers.

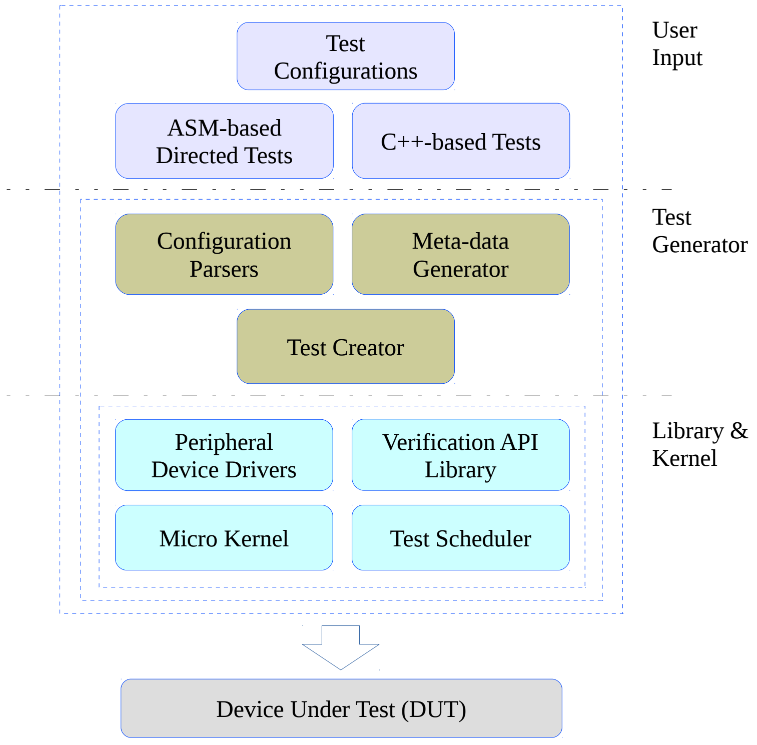


Figure 2: Different components of the software stack of STING

Figure 2 shows the different components of the software stack of STING. It consists of test generators, checkers, device drivers, verification libraries/APIs and a microkernel. These are built into a bare metal ELF image which can be booted seamlessly on the DUT in any verification environment, such as in simulations, in-circuit emulation, FPGA prototype and silicon. The user can control the generation of intelligent, self-checking, and architecturally correct test sequences in the portable test program using intuitive test configurations. After it is booted, the program can run tests targeting a specific SoC/CPU feature and report any anomalies detected during execution.

In addition to the random tests, STING also supports programming frameworks and mechanisms to cover different test stimulus generation methodologies.

Directed tests (for scenarios like *Mutual Exclusion*, *Cross Modifying Code*, *Memory Ordering*, etc.) can be developed using a programming framework which allows users to write stimulus in an assembly language like syntax. This framework forms the foundation of the architectural compliance tests in the RISC-V verification suite in STING. For tests which require complex programming constructs (such as *CRC calculation*, *Fast Fourier Transform*, etc.), or which are algorithmic in nature (such as *Level 3 Cache Collision*), STING provides a C++ based programming framework to develop stimulus in high-level code. Abstractions on top of this framework have been created to enable development of drivers for peripheral devices in an SoC. Snippets of code from the OS, applications, and benchmarks can also be ported to STING easily using the C++ based framework.

New tests and feature enhancements were added to STING in order to cover the scenarios identified in the test plan for the PULPino RISCY core. The next section describes the different tasks undertaken during the project execution in detail.

4. Tasks For Verification Of PULPino RISCY Core

The verification of the RISCY core was verified by running STING tests on the PULPino platform testbench. The PULP software tool chain was used for the build environment of STING to compile the source code into an ELF/executable, which was then loaded into the testbench and run on the DUT.

The RISCY core verification project was divided into multiple phases which have been described in detail in the sections given below.

A. Preliminary Enabling – Tool Bringup, Test-bench Changes

The default memory map of PULPino has 32kB of non-contiguous instruction and data RAM [2]. Since a small-sized memory limits the number of test instructions and size of memory buffers in STING, we changed the configuration parameters in PULPino top level SystemVerilog file to create a single unified 2MB RAM, with the boot ROM relocated to a high memory address. Every STING image for PULPino is run on SPIKE, a RISC-V functional simulator, to get the reference test results for checking. In order to match the program counter and memory buffer used for the reference run on SPIKE, the boot address was changed to a different value in the testbench.

These changes enabled simple STING tests to boot and execute on the PULPino test bench.

B. Test Plan

We leveraged an existing RISC-V CPU test plan [6] for the RV32I base integer instruction set and extensions for the compressed and integer multiplication/division instructions. We added an addendum for PULPino specific features such as the custom instruction extensions and the privilege level architecture.

C. Tool and Test Stimulus Enhancements

Since the RISCY core implements only a small part of the RISC-V privilege specification [5], we modified the kernel in STING to enable execution of tests on the PULPino test bench. We also made changes to account for the difference in the bit-fields of control and status registers and interrupt vector table (IVT).

One key functionality to be verified was asynchronous interrupts to the core. We added device drivers for the event unit, timers, and UART to enable asynchronous interrupt delivery. On the test stimulus side, we added new configurations to the test database to ensure that all the hardware features identified in the test plan were covered. We also developed cross product tests that exercise the interaction between the instruction execution and events/interrupts.

D. Execution Phase - Readiness, Volume Regression, Debug

Once the tool and test stimulus development milestones were reached, we began volume regressions. Each test had a priority assigned in the hardware regression plan which determined the frequency at which it was selected in the regressions.

The failures from regression went through the first level of debug at Valtrix. If the failure trended towards being a logic bug, it was escalated to the design team at Google for further debug. The Google design team then worked with the PULP team at ETH to devise a fix and upstream it to the open source repository. In case a RTL fix was made, it was verified with extensive regressions of the test configuration which resulted in the original failure.

E. Coverage tasks

Measuring coverage is a crucial task in achieving verification sign-off. STING coverage manager is an utility which extracts information about different architectural events from the test files and execution logs and populates a database. The database can then be queried to determine if a particular scenario from the test plan is getting covered or not. For PULPino, we enhanced the coverage manager to process the core trace file generated by the test bench and developed queries to assess the overall test plan coverage. We also generated code coverage to find if there were any missing holes in the stimulus.

5. Coverage Results

We developed a large number of queries for the database created by STING coverage manager to check if the scenarios defined in the test plan are getting covered or not. We analyzed the results to adjust the test stimulus and fix gaps in the test plan coverage. For example, the initial rounds of analysis revealed a poor coverage of scenarios involving *ecall* instruction in the test, which was then fixed for the future regressions. We have been continuously monitoring the coverage results and tweaking the test stimulus and regression plan to achieve test plan completion.

The code coverage analysis has also yielded some interesting results. We have found that the STING tests have covered most of the core functionality. The code coverage numbers for the RISCY core from the regressions are given in the table below -

Block	Expression	Toggle	State	Transition	Assertion
94.14%	89.55%	98.41%	83.87%	70%	100%

Table 1: Code coverage results from STING regressions

The uncovered areas were mostly in PULP specific extensions (including hardware loops, bit manipulation, vector operations, MAC) and features of sleep control, floating point unit, debug unit, performance counters and illegal instructions, which were not in the scope of the verification as the use cases for these features did not exist. The remaining holes are being addressed by adding more STING based tests to achieve coverage. The new tests added to close coverage goals have also helped in discovering new bugs.

6. Design Issues Found In PULPino RISCY Core

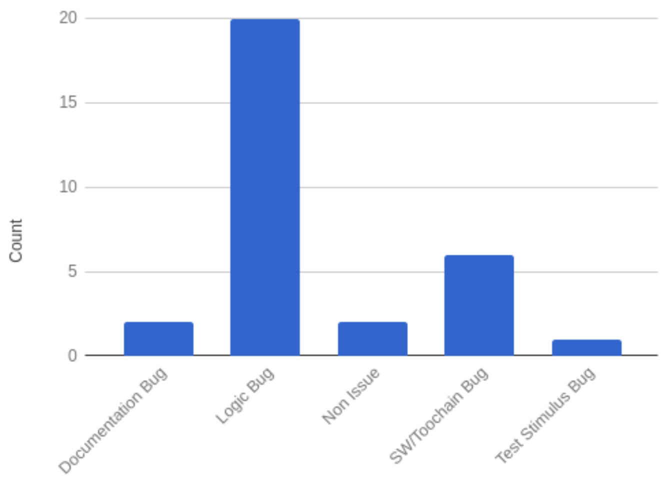
A number of design issues were found in the RISCY core by running STING tests on the PULPino test-bench. The high level details of the bugs have been summarized in the table given below.

S.No.	Title	Root Cause	Fix Classification
1	Binding between timer interrupts and event unit lines is different from what is documented	Documentation Bug	Documentation/Errata Fix
2	Documentation issue with lp.setupi instruction	Documentation Bug	Documentation/Errata Fix
3	Illegal instruction exception with fence and fence.i instructions	Logic Bug	Documentation/Errata Fix
4	Invalid memory access exception not generated for memory accesses to non-existent memory	Logic Bug	Documentation/Errata Fix
5	Core execution hangs after a directed test sequence which access invalid memory addresses	Logic Bug	Documentation/Errata Fix
6	Destination register value miscompare with mulhsu instruction	Logic Bug	RTL Fix
7	Hardware loop counter does not decrement if loop end boundary spans across an uncompressed instruction	Logic Bug	Documentation/Errata Fix
8	Unexpected mepc csr read return value from csrrwi instruction	Logic Bug	RTL Fix
9	Execution hang with hardware loops and timer interrupts	Logic Bug	RTL Fix
10	Instructions in the branch shadow of bge getting executed	Logic Bug	RTL Fix
11	Writes to RO CSR mhartid using cssrc does not generate illegal instruction exception	Logic Bug	Documentation/Errata Fix
12	Low bit of CSR mepc is not set to 0	Logic Bug	Documentation/Errata Fix
13	Data from load bypassing a CSR read and incorrectly forwarded to store	Logic Bug	RTL Fix
14	Unexpected loop exits if interrupts arrive at the last instruction of the hardware loop	Logic Bug	Documentation/Errata Fix
15	Access to non-existent CSR does not generate illegal instruction exception	Logic Bug	Documentation/Errata Fix
16	Possible tracer issue when lw tries to read its own instruction memory	Logic Bug	Documentation/Errata Fix
17	Unpredictable behavior on code execution from invalid memory	Logic Bug	Documentation/Errata Fix
18	RV64 instruction lwu is getting detected as p.elw in PULPino instead of generating illegal exception	Logic Bug	Documentation/Errata Fix
19	Unexpected illegal instruction exception with HINT forms of compressed instructions	Logic Bug	Documentation/Errata Fix
20	Reserved instruction form C.JR (rs1=0) does not generate illegal instruction exception	Logic Bug	Documentation/Errata Fix

21	Performance bubbles introduced in the pipeline when IRQ happens when MUL* instruction is in progress	Logic Bug	Documentation/Errata Fix
22	Performance bubbles introduced in the pipeline when taken branch followed by DIV/REM instructions	Logic Bug	Documentation/Errata Fix
23	CPU hardware threads are not getting stalled with a WFI instruction	Non Issue	Documentation/Errata Fix
24	Issue with dynamic loop end update using lp.endi inside hardware loop	Non Issue	Documentation/Errata Fix
25	Assembler incorrectly detects few compressed instructions with zero immediate as illegal operations	SW/Toochain Bug	Toolchain Fix
26	riscv32-unknown-elf/bin/ld: can't relax section: No more archived files	SW/Toochain Bug	Toolchain Fix
27	Call to virtual functions resulting in jump to 0x0	SW/Toochain Bug	Toolchain Fix
28	Issues with compilation of inline assembly at higher levels of optimization	SW/Toochain Bug	Toolchain Fix
29	Assertion failure in md_apply_fix at gas/config/tc-riscv.c line 2205	SW/Toochain Bug	Toolchain Fix
30	SPIKE does not generate illegal instruction exception for c.sxxi instructions encoded with zero shift amount	SW/Toochain Bug	Toolchain Fix
31	Execution hang with RAW dependency between load operations	Test Stimulus Bug	Test Stimulus Fix

Table 2: List of issues found in verification of PULPino RISCY core

Logic bugs in the design which were critical for the use case were fixed in the RTL, while the others were reported in internally published documentation/errata. A number of issues were found in the software tool-chain and PULPino documentation as well. The distribution of the bugs according to the failure root cause and the fix classification are given in Figure 3 and Figure 4 below.



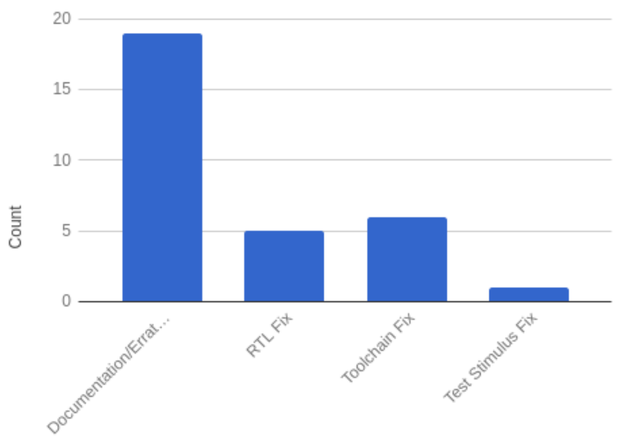


Figure 3: Distribution of bugs according to failure root cause

Figure 4: Distribution of bugs according to fix classification

During early verification we found some critical issues in the implementation of hardware loops (check sub-section C below for details). We removed hardware loops from scope of verification because there was risk of closing the bugs in time and we had limited performance gains from this feature in our usage model. We restricted the toolchain not to use them in the current implementation.

Some of the interesting logic bugs found in the RISCY core have been described in the sections given below.

A. Destination register value miscompare with mulhsu instruction

This issue was reported from the value sweeping test for *mulhsu* instruction. The test picks random input values and compares the results from the execution of the instruction on DUT and a functionally accurate simulator (SPIKE in this case).

RISCY implements the multiplier as a multi-cycle operation which involves cross multiplication of 16-bit operands. A problem would arise in the third multiply cycle, where one of the operands was getting incorrectly interpreted as a signed negative value, if the MSB of both the operands were set. The implementation was fixed to address the corner case condition.

B. Data from load bypassing a CSR read and incorrectly forwarded to store

This issue was found by a pipeline test which exercises the register write-after-write (WAW) dependency between a load and CSR read instruction. In case of this failure, the data read by load was bypassing the CSR read instruction and getting forwarded to a later store.

In this case, the core was not being stalled on a load to an ALU register causing the incorrect value to be stored. The stall mechanism was updated to fix the issue.

C. Unexpected loop exits if interrupts arrive at the last instruction of the hardware loop

This issue was a result of interaction between instructions inside a hardware loop and asynchronous timer interrupts. It was observed that the execution was coming out of the loop in case the timer interrupt arrived at the last instruction of the hardware loop.

This bug comes when an interrupt arrives in the middle of hardware loop execution and the *mret* restores the last instruction of the loop. If that instruction is misaligned, only the first part of the instruction is correctly fetched before the hardware loop jumps to the first instruction of the loop. The RTL fix for this issue is being currently investigated.

D. Performance bubbles introduced in the pipeline when taken branch followed by DIV/REM instructions

This issue was found on reviewing the coverage results accumulated from the regression tests by the STING coverage manager. It was found in few cases that the compressed conditional branch instructions were taking a large number of cycles to execute.

In this case, we observed that the multiplier remains enabled while branch is taken causing a second execution of the DIV/REM. This is a performance issue and has not been addressed in the implementation yet.

7. Conclusion

STING has been successfully used for the functional verification of PULPino RISCY core. The coverage results and the list of design issues from running STING tests on the PULPino test bench has been presented in the previous sections. We hope that the verification flow established by this work will be reused for future RISC-V based implementations.

Acknowledgements

We would like to thank Schiavone Pasquale Davide from the PULP team at ETH Zurich for helping out with the RTL fixes for the RISCY core. We would also like to thank Don Stark from Google for reviewing the paper and provide his feedback.

References

[1] D. Rossi, F. Conti, A. Marongiu, A. Pullini, I. Loi, M. Gautschi, G. Tagliavini, P. Flatresse, L. Benini, "*PULP: A Parallel Ultra-Low-Power Platform for Next Generation IoT Applications*", August 2015, Hot Chips - A Symposium on High Performance Chips, California, USA

[2] "PULPino Datasheet", June 2017, Integrated Systems Lab, ETHZ Zurich, Switzerland

[3] *"The RI5CY User Manual"*, July 2017, Revision 1.7, Micrel Lab and Multitherman Lab, University of Bologna, Italy/Integrated Systems Lab, ETHZ Zurich, Switzerland

[4] *"The RISC-V Instruction Set Manual, Volume I : User Level ISA"*, Document Version 2.2, CS Division, EECS Department, University of California, Berkeley

[5] *"The RISC-V Instruction Set Manual, Volume II : Privileged Architecture"*, Privileged Architecture Version 1.10, Document Version 1.10, CS Division, EECS Department, University of California, Berkeley

[6] *"RISC-V CPU Test Plan"*, Revision 1.0, October 2017, Valtrix Technologies Pvt. Ltd.

[7] S. Roy Choudhury, S. Thiruvathodi, "STING – A software driven portable stimulus generator for SoC design verification", April 2017, RISC-V International Conference, Chennai, India