RISC-V Design Verification Strategy

by Mike Bartley – Tessolve, Lavanya Jagan, G S Madhusudan & Neel Gala - InCore Semiconductors Pvt. Ltd.

As the RISC-V architecture becomes increasingly popular, it is being adopted across a diverse range of products. From the development of in-house cores with specialized instructions, to functionally safe SoCs and security processors for a variety of verticals – RISC-V adoption brings several verification challenges that are discussed in this article, along with potential approaches and solutions.

This article first considers verification of the core. The core is made up of several blocks including: a fetch unit, execution units, instruction cache and data cache, TLB and complex logic for controlling functions like branch prediction and out-of-order execution. We discuss the pros and cons of performing block level verification. Either way, a core level verification strategy is required (this includes both constrained random instruction generation targeted at micro-architectural features and corner cases captured in functional coverage), and architectural compliance. The article outlines the tools and compliance suites required and available.

Moving out from the core the article considers both fabric integration verification and SoC level verification where safety, security, and low power features come into consideration. In addition, the article also considers the verification of RISC-V specific features including multiple architectural options, configurability, and the ability to add new instructions.

INTRODUCTION

With the recent explosion in domain specific applications (DSAs), the industry has seen an unprecedented need for customized hardware solutions. Intelligence, safety, and security, which were previously a mere afterthought, are now considered first class citizens in designing processors and other hardware solutions. While exploration of DSAs is still underway, it is not uncommon to think that there are significant overlaps in terms of compute and resource requirements across these domains, and a configurable architecture can address multiple domains.

With this in mind, RISC-V, an open source instruction set architecture (ISA) was introduced in 2010 as an extensively configurable and customizable solution for general purpose processors. The customization capabilities of the ISA enable one to implement a processor for RISC-V ranging from resource constrained deeply embedded in-order cores all the way to high performance highly pipelined multi-core micro-architectures. Over the past decade more than 300 industries and academic institutes have shown interest in RISC-V and contributed to its growth significantly. For the current standard extensions (i.e. I, M, A, F, D, C and privileged) the entire software stack including compiler, assembler, debugger, operating systems, kernels, etc., are all available in the open source domain. Commercial entities such as InCore Semiconductors, Western-Digital, and Bluespec Inc. have also contributed their core implementations (Chromite, SweRV and Piccolo/Flute respectively) to the open-source community enabling users to take these to silicon at no design cost.

One should also note that, with a decade of progress and success of the RISC-V ISA, it continues to evolve and define new extensions which cater to new paradigms of computing and is thus truly an ISA of the future.

Architectural compliance of a RISC-V Design

One of the advantages of the RISC-V ISA is that it provides no mandates with respect to the micro-architecture. This allows designers to realize the same specification in application specific configurations catering to various targets of power, area, and performance. This limits the scope of compliance checking to only an important subset of the specification making compliance an achievable goal.

Compliance means that the implementation of the ISA spec meets all the requirements set forth by the spec under all conditions defined by the spec. Essentially, this would mean that a compliant design

is functionally bug free. Considering the numerous configurations that RISC-V provides, it is impossible to create a test-suite which can declare any and every RISC-V device as bug-free. What is possible, is to test for compliance in limited areas. Some such tests could be comprehensive – testing for full compliance in a specific area.

The compliance test-suite thus consists of directed tests. Tools like RISCOF, RISCV-ISAC and RISCV-CTG from InCore Semiconductors, have started addressing the above challenges of compliance and have provided a framework and test-suite for the base instruction extensions. Section 3 provides more details on these tools and infrastructure which can be used and extended to check compliance of a device.

Verification of a RISC-V Design

The compliance suite is thus narrowed down to a very small subset of the verification and the more rigorous testing is required to declare a device "almost" bug-free (but is not required to declare a device compliant).

To cater to the configurability of the ISA, entities like InCore Semiconductors have developed core-generators which can provide almost the same number of hooks as described in the ISA spec. This allows the same RTL source to be used across different instances and configurations of the core. To verify such core-generators, it makes sense to first validate individual design blocks at the unit level and then re-use the validated blocks to build new cores. This reduces the stress of verification at the core level and enables the verification suite to grow in tandem with the design and the spec. In section 3.1 we describe RiVer-Block, a python based framework, which has been developed to verify and validate unit b.

As designs become more complex, each unit/block will also grow to be more complex and customized, thereby requiring more effort to perform unit level testing. Also, in the initial stages of the design, it is required to flush out common bugs as soon as possible. In view of this, RISC-V has seen development of quite a few stress-test / pseudo-random-test generators like: AAPG, MicroTesk and RISCV-DV. These tools, varying in capabilities and limitations, generate random assembly programs for different supported configurations of the spec. Each test will generate a signature of the user-visible context of the core which needs to be verified against an instruction set simulator/model like Spike, riscv-ovpsim or SAIL.

The challenge now is to tune these pseudo-random-generators with configurations and setup files specific to the DUT. To manage this complexity, RiVer-Core (see section 3.2) provides an umbrella-like framework to choose, configure and run the test-generators on the DUT and compare the results with a reference simulator. One can use RiVer-Core to run thousands of parallel simulations in an automated daily routine enabling quick coverage-climb and catching bugs as early and fast as possible.

Verifying Extensions of a RISC-V Design

The most attractive feature of RISC-V is its ability to add custom instruction extensions. With standard open source SW toolchain components like LLVM, GCC and Spike supporting RISC-V, building a minimal working toolchain and reference simulators for custom extensions of RISC-V is now straightforward.

On the hardware end, there exist multiple ways in which these extensions can be integrated to an existing RISC-V core. Based on the integration mechanism different verification strategies need to be adopted which are highlighted below:

1. **Pipeline-coupled:** the implementation is tightly coupled with the existing pipeline and re-uses much of the resources already available in the core. Verification here requires a re-engineering of the unit-level testing for existing blocks which were modified and for newly introduced blocks. Verification at the core level will require RiVer core and its components to be updated to generate the new set of instructions.
2. **Accelerator:** the implementation is a standalone unit and interacts with the core through a dedicated accelerator interface like RoCC. This requires block level verification of the new standalone unit introduced and updating RiVer-Core to generate new instructions.

More details on verifying extension is summarized in Section 2.3 Modern instruction extensions for safety and security will further require novel data-flow validations and checks which are highlighted in Section.

Verifying RISC-V based SoCs

With the rapid adoption of RISC-V many vendors are migrating their existing designs and SoCs to use a RISC-V core. The challenge here is to now build the SoC level verification infrastructure which can integrate any RISC-V core/configuration for a customized set of peripherals. The verification framework should further re-use the standalone IP tests, and port existing SoC level tests to RISC-V seamlessly.

In response to the growing momentum of RISC-V, groups like ChipsAlliance and OpenHW are developing and standardizing on uncore components like omni-extend, TileLink , PLIC, and CLIC, which will further ease the verification of custom SoC built using these components. With a vision to leverage this opportunity, we present a unique automated framework in Section 3 which can not only generate custom SoCs but also provide the necessary SoC verification infrastructure as well. The goal of this tool is to provide a stable and robust starting point to create customized SoCs.

RIVER FOR RISC-V CORE AND SOC VERIFICATION

Figure 1 shows a typical RISC-V core block diagram. It illustrates a common RISC-V implementation along with SoC components. The ISA extensions of the implementation and microarchitecture features are listed on the opposite page. These are highly configurable in the sense that they can be enabled or disabled as well as having configurable design parameters.

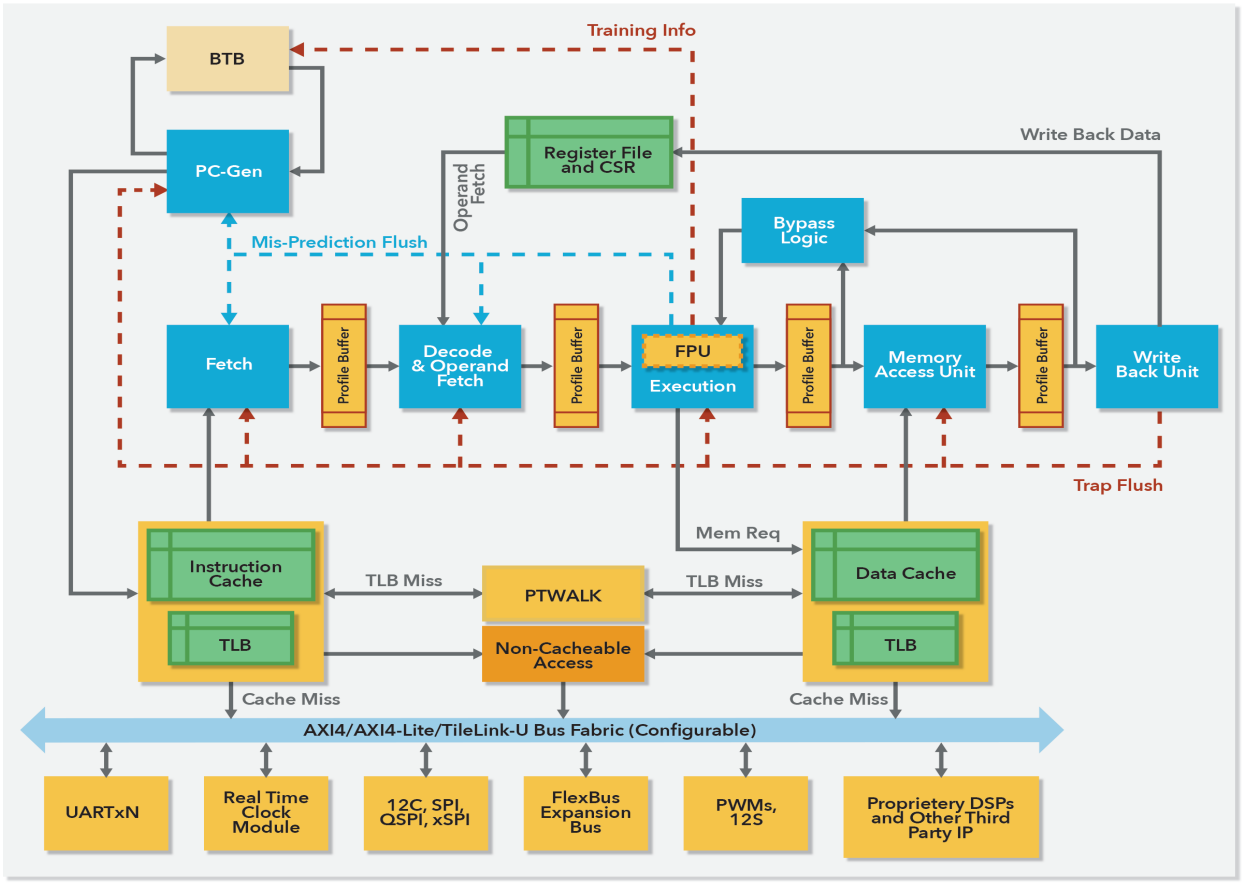


Figure 1: Typical Core Microarchitecture

- 32- or 64-bit variations with RISC-V ISA configurability of RV[32/64]I[MAFDCSUN] where
  - o I - Integer compute unit
  - o M - Integrated multiply-divide unit
  - o A - Atomic functional unit
  - o F/D - Single and Double precision floating point compute unit following IEEE-754 standards
  - o C - Compressed instruction support unit for lower memory footprint
  - o SUN - Supervisor, user, user trap mode of operation support
  - o PMP support: configurable number of regions for physical memory protection
  - o configurable number of performance counters
  - o Configurable number of triggers
  - o Debug support
- Optional branch predictor with fully associative Branch Target Buffer (BTB), variable size Branch History Buffer, configurable Return Address Stack (RAS)
- Separate instruction and data caches with configurable associativity, size, and replacement policy support
- Configurable Memory Management Unit with fully associative, separate instruction and data TLBs and variable page size support

For the extremely configurable processor design shown here, the verification environment is also required to be configurable in an automated fashion. This aids in rapid verification during the design phase improving verification quality. The verification itself can be performed at various levels as described in the following sections. The article predominantly concentrates on simulation-based verification using open source simulators or commercial simulators like Mentor's Questa® Simulation.

Block for block-level verification

Block or unit level verification deals with development of UVM (Universal Verification Methodology) based verification around the small unit, constraining random input sequences and comparing the design output with a reference model. Verification of a block in a processor design could either be carried out in the above fashion or by targeting RISC-V tests towards verifying the block at the processor level. The decision to choose a block for rigorous UVM based verification depends on the following criteria.

- The complexity of the block: What are the risks that we cannot discover all the bugs in core-level verification?
- The simplicity of block level verification: Does the block have standard interfaces? Is it possible to make a self-checking test bench (e.g. is there a model? Or is it easy to build a scoreboard?)

Cache controller is a good example of where a block level testbench is often used as it fulfills both criteria

- There are many verification challenges including coherency across multiple hierarchies of memory as well as liveness for requests from multiple cores. The latter might be best verified using formal property verification rather than simulation.
- On one side there is a standard CPU connection and a standard memory connection on the other. It is also easy to build a flat model of memory to act as a model or scoreboard.

RiVer block is a Python based verification framework that aids in block level verification. It uses the CoCoTb (Coroutine based Cosimulation Testbench) libraries to develop the UVM components as shown in the diagram below.

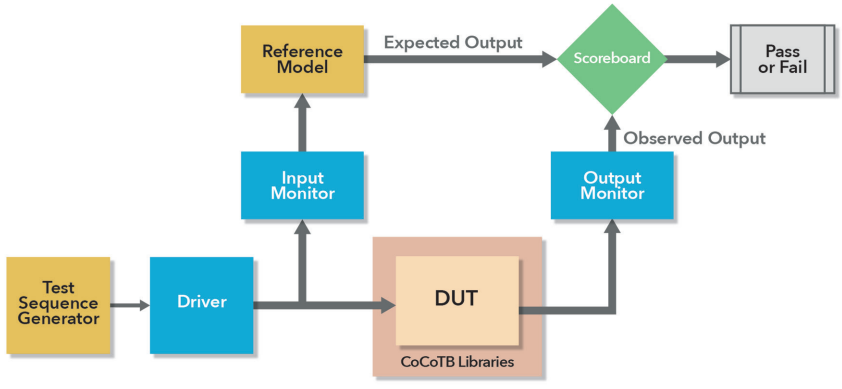


Figure 2: CoCoTB libraries and UVM components

The test transactions are provided to the Driver component which translates it to signals to be driven to the design-under-test (DUT). The *Input Monitor* observes the signal changes happening at the DUT and are taken as inputs to the *reference model* to obtain the expected output for the input transaction. In a similar fashion, the DUT's output is observed by the *Output Monitor* which is compared with the expected output in the *Scoreboard*. The framework supports DUT simulation using commercial simulators like Mentor's Questa® Simulation along with open source simulators.

When a block is verified using the UVM environment, the verification components (other than the driver) can also be re-used at the core level to localize the bug that is occurring when running processor level tests. The monitors and scoreboard attached to a block exposes more details of any mismatch seen at the core level.

RISC-V Core Verification

In this section, verification of the RISC-V core design (or micro-architecture) will be discussed. Core design verification is typically based on running RISC-V assembly level programs (test programs) on the core to ensure correct program execution. The major test program suites are either manually developed or generated from RISC-V assembly program generators. The GNU toolchain compiles these test programs and are loaded into the design memory and the instruction set simulators (ISS). To check the execution of the program, the processor state, namely, the registers (general purpose, floating point, control, and status) and memory are compared. This state is obtained from the core design through the UVM based testbench environment and from ISS for comparison.
The following comparison strategy is generally followed for core verification:

- The register state is compared after every instruction retirement. This aids in quicker debug time from bug to the cause of failure with minimal test bench performance hit. This excludes micro-architectural registers and is done at the ISA level
- The whole memory state comparison is done at the end of test. This is basically termed as the signature of the test. The memory is initialized with test data and the test executes modifies the state of memory which is compared with the reference model at the end of the test.

Other than the normal execution flow, timing of external events (such as interrupts) that disrupt the flow are handled by syncing the design with the ISS. Some verification environments therefore use a cycle accurate model of the CPU to overcome this but at the cost of higher maintenance, potential bugs in the model and slower simulation performance.

Finally, a key aspect of a core verification strategy is the coverage analysis. Structural coverage of the design is typically used (and some engineers also add structural coverage of the ISS model) but the key aspect is the functional coverage model which can be at two levels.

- Architectural coverage including instructions, registers, etc. This is typically covered by the architectural compliance suite (see below) so the model might want to go further: for example, a write followed by a read to the same (cacheable or uncacheable) memory location. Although this is targeting a potential micro-architectural feature, it can be measured statically at the program (i.e. architectural) level.
- Micro-architectural level: for example, coverage of features controlling out-of-order execution or branch mis-prediction.

The above verification flow of generation of tests, compilation, simulation, and comparison of processor state are seamlessly integrated as a one-stop solution in the RiVer Framework. This framework for RISC-V Core Verification also exploits the extreme configurability of the ISA specification. The framework consolidates various ingredients necessary to perform instruction level verification of any core which includes the following: custom assembly test suites, random assembly program generators; and various instruction set simulators (ISS) as golden reference along with coverage definitions. RiVer is thus a robust verification methodology of instruction level checking for any RISC-V based core design.

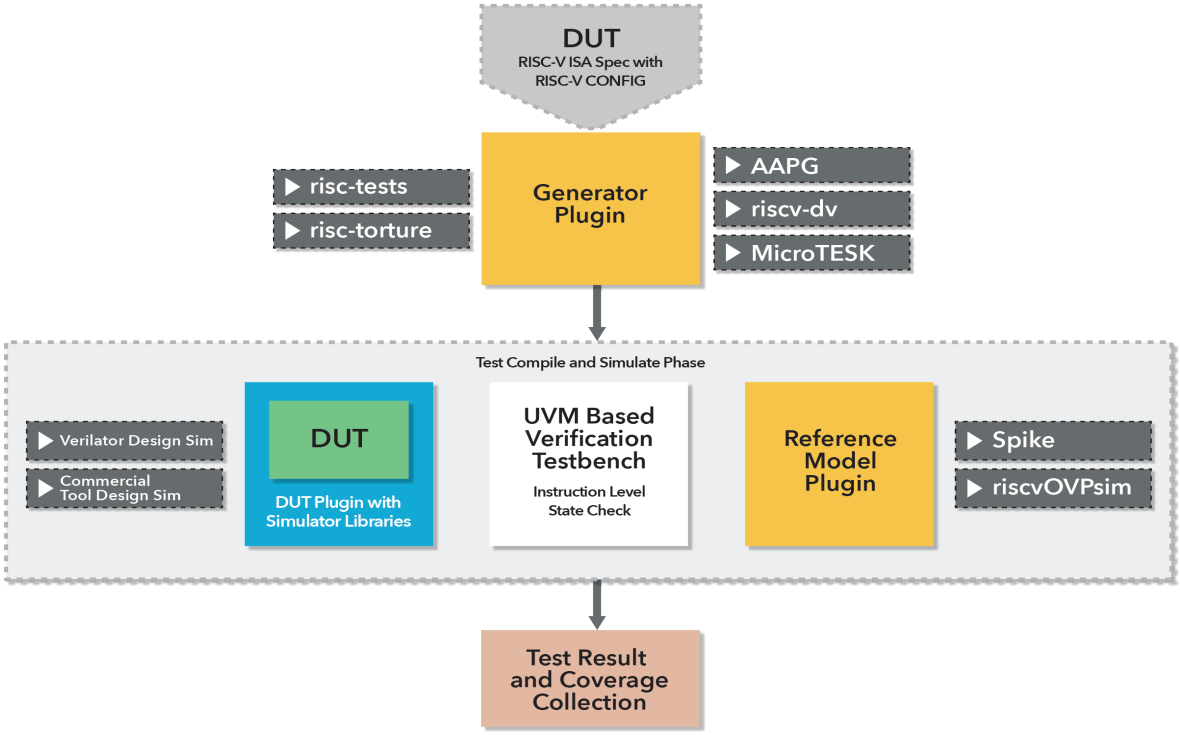


Figure 3: Generator plugin to test compile + simulate phase test result and coverage collection

The RiVer framework captures various choices and configurations of the DUT via an intuitive human readable YAML format: RISCV-CONFIG. RISCV-CONFIG employs a strong validator which ensures that the user has not made conflicting choices available in the ISA. Thus, riscv-config acts as an early filter of bugs.

Using this framework, the user can choose from a wide variety of custom and random assembly suite generators. The custom suites are generated using the Compliance Test Generator developed by InCore. The random program generators currently supported by RiVer include Shakti's AAPG [5], Google's riscv-dv [7], MicroTESK [8] and Berkeley's riscv-torture [6]. These random assembly program generators are typically associated with configuration file or template file that controls the generation sequence, instruction distribution along with targeting any micro-architectural features. Since the framework is built on a plugin philosophy, adding new suite generators to RiVer is easy and can be seamlessly achieved.

Once the user provides the riscv-configuration YAML file and choice of test suites, RiVer automates the flow of generation of tests, test compilation and execution on DUT. To validate the correctness of the DUT, RiVer also enables integration of external open/proprietary RISC-V Instruction set simulators which act as golden reference models. RiVer provides a UVM based verification testbench environment which enables state comparison across the DUT and the reference model of choice, and thereby arriving at a decision if a particular test has passed on the DUT or not.

The simulation phase involves simulating the DUT on the chosen simulator (Mentor's Questa® Simulation, Verilator, etc.). Again, the plugin and abstraction philosophy contribute, allowing easy switch between simulators without a hassle. RiVer also exploits the coverage analysis features of these tools to provide a code and functional coverage of the tests being executed on the DUT. ISA level functional coverage of the tests can be obtained using tools like RISCV-ISAC as discussed in the next section.

RiVer also comes along with a CI (Continuous Integration) flow allowing developing teams to start ISA verification as early as possible an on every change. Since RiVer integrates largely open source tools for generation and compilation, many of the RiVer stages can be easily parallelized across threads, cores, and machines. The limiting factor is typically the commercial simulators which limit the number of parallel licenses.

RISC-V Compliance Testing

A RISC-V design is said to be compliant if the implementation of the ISA meets all the ISA feature requirements under all conditions. Compliance is not aimed at certifying an implementation to be bug free as it is not a replacement for verification. Rather, it handles a subset of the verification space. The compliance tests focus on checking an implementation in limited areas of the ISA and not on any micro-architectural details. These tests basically rely on signature-based testing where the RISC-V test is developed according to the test specification format as defined by the RISC-V Compliance group. The tests are compiled and executed on the specified implementation and a chosen reference model. Each test has a signature section which is basically a memory section to which the test stores intermediate results. The unique signature captures the essence of the test and is used to detect the compliance correctness. The same test is run on the reference model and the signatures are compared. This flow helps in compliance checking of an implementation with simulation, emulation, an FPGA prototype, or an ASIC device.

To provide a comprehensive flow to perform compliance testing, a Python YAML based RISCOF framework can be used to prove compliance of a RISC-V implementation against the RISC-V privileged and unprivileged ISA spec. The framework provides the following features to prove compliance.

- Allows an implementation to be compared against a chosen reference model
- Compliance test suite applicable to the implementation are filtered based on its configuration definition
- Allows usage of different SDKs or toolchains for compiling the compliance test suite
- Provides compliance coverage reporting the quality of the testing done

To provide the above flow, RISCOF takes the ISA and Platform specifications (in Python YAML format) as input which defines the DUT's implementation specification. After validating the specification using RISCV-CONFIG, a compliance suite applicable to the design implementation is selected by RISCOF. These tests generate signatures from the DUT and the chosen reference model following which the framework compares them to prove compliance.

To improve automation in compliance testing and coverage closure, RISCV-ISAC (RISC-V ISA Coverage) provides RISC-ISA compliance coverage definition. Using ISAC's definition RISCV-CTG tool generates assembly tests targeting the coverage for compliance. RISCOF uses these generated tests and provides ISA level completeness to compliance testing.

RISC-V ISA Custom Extensions Verification

In addition to the standard ISA extensions, RISC-V ISA provides opcode space to support custom extensions. These custom extensions would either become the differentiating factor of the implementation or target custom application domain. To verify such a combination of a standard RISC-V design along with the custom extension, the following support are to be provided to the verification ecosystem:

- Custom instruction support in the toolchain to compile the tests with the new instructions
- ISS is updated to have the custom instruction features so that they are used for comparison with the implemented design
- Self-checking manual tests must be developed to check the above updates
- Random generators must be updated to generate the custom instructions
- Templates to generate the random instructions must be developed for verification.
- Additionally, the functional coverage model should be updated based on both the architectural extensions and design updates needed for the additional instructions

With the above ecosystem updates, the RiVer Core methodology can be used for verifying the implementation with the custom instruction support. Compliance testing will be performed for the RISC-V standard ISA specification.

RISC-V BASED SOC VERIFICATION

RISC-V based SoC design complexity has seen a shift in terms of increased adoption towards Domain Specific Architectures (DSA) like for security and AI/ML domains. This adoption brings in new design and novel interconnect components into the SoC design. Verification of these SoCs thus deals with verifying the added components along with the highly configurable RISC-V core implementation.

Consider an example of a RISC-V based SoC shown below. It consists of a RISC-V processor implementation with various blocks like multiplier/divider, floating point, debug units. Furthermore, it also includes micro-architectural components like branch prediction, cache controller blocks. This processor implementation is integrated through interconnect fabrics to other peripherals like SPI (Serial Peripheral Interface), QSPI (Quad SPI), UART (Universal Asynchronous Receiver Transmitter), PWM (Pulse Width Modulation) etc.

Thus the various types of components like generic peripherals (like SPI, UART, PWM), hardware accelerators (like crypto accelerators or systolic arrays), co-processor units (like bit manipulation, floating point) and fabric bridges form the basic building blocks of a generic RISC-V SoC. Their parameters can be varied, and several different RISC-V SoC configurations can be considered targeting different application domains. These components are verified at the block/unit/IP level using industry standard Verification IPs (VIPs) and then system level integration tests are performed along with use-cases or applications, performance testing, power and reset testing, multiple clock domain testing; and safety and security based verification. As seen, the design complexity increases from the notion of a highly configurable RISC-V processor implementation to an extremely configurable RISC-V SoC with that configurable processor along with other peripheral components.

To meet this challenge of maintaining different SoC design and verification systems for varied application domains, a unified approach of generating the SoC as well as its verification environment from the same design specification is proposed. This approach accelerates building the whole SoC design in an automated fashion along with generating its verification environment at the same time. As a first step towards this automation approach, the SoC specification must be defined in the form of Python YAML format. The SoC YAML Specification describes the various SoC components, its connectivity and design parameters in a YAML

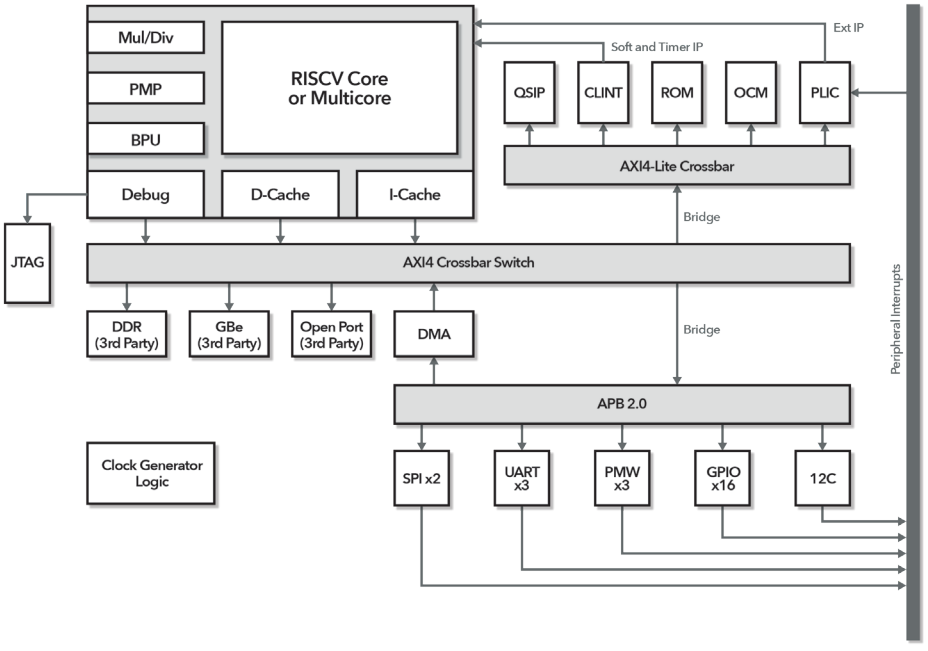


Figure 4: RISC-V based SoC

structure as follows thereby aiding in generating the SoC design and its verification environments:

- Modules specify various design instances like the core, peripheral units, hardware accelerators, co-processors, interconnect bridges
- Clusters specify the interconnects associated with the SoC details the master/slave roles of the modules associated with a particular cluster
- Module Parameters define in detail the design parameters of the modules. This is part of the module instance. They typically include the feature list and design parameters of an SoC component like ISA extensions, modes of operation supported for a RISC-V core, address widths supported for the clusters. The exhaustive specification aids in configuring the Verification IPs (VIPs) associated with the block as well as filtering the tests applicable for the SoC component
- Interface signals of a module or a cluster allows the verification framework to use their corresponding VIPs to be integrated in the SoC verification environment
- Connection Hierarchy enables automation in building the SoC design or its verification environment
- Memory Map associated with the SoC is also part of the specification

Once the SoC Specification is in place, as seen in figure 5, the SoC Generator builds the (BSV based) design in an automated fashion and at the same time the testbench VIP components as well as the test infrastructure applicable to the SoC configuration is generated. This eases the integration efforts for verifying the different SoCs. In addition, it automates several redundant test bench development, test filtering instead of re-writing the same test functionality for slight variations in the configurations. For this ease of automation to happen, the VIPs which include the UVM testbench components like drivers, monitors and scoreboards must be strictly parameterized based on the available design parameters. Along with the testbench, the VIP test cases also must be transformed to be aware of the parameters. This transformation of the VIP to become design parameter aware makes it a principle building block of generating an SoC test bench and its associated tests from the design parameters itself and will boost the actual verification efforts of the SoC.

The following flow is maintained for the RISC-V SoC Verification Automation:

- The Python based SoC YAML spec is first validated to be correct and devoid of any human errors by the SoC TB generator
- In the next step, the validated spec is used to resolve the interface connections and hierarchical details to build the SoC testbench using the various standard VIPs
- With the testbench generated, a test selector filters though the suites to generate the tests applicable for the SoC configuration
- Once the whole environment is ready to be deployed for verification, test regressions in the form of continuous integration is triggered to perform the primary verification task for detecting design failures.

To meet the challenges of verifying the highly configurable RISC-V SoCs, configuration-aware verification automation is crucial for maintaining the verification quality.

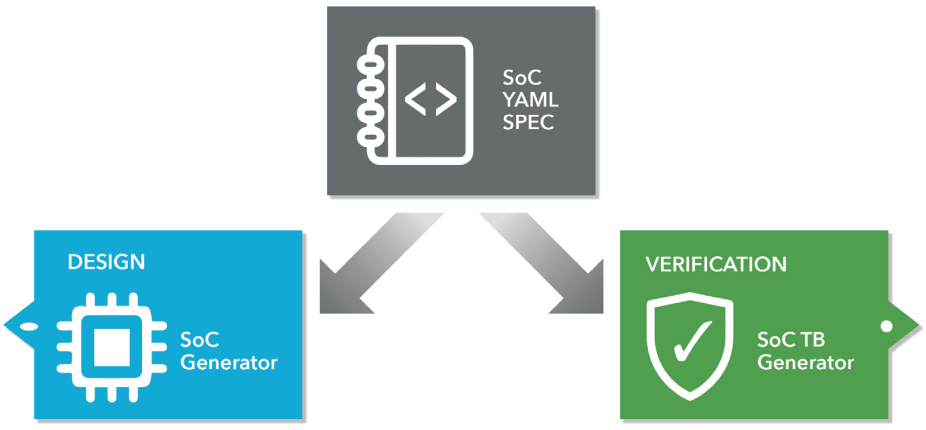


Figure 5: SoC specification

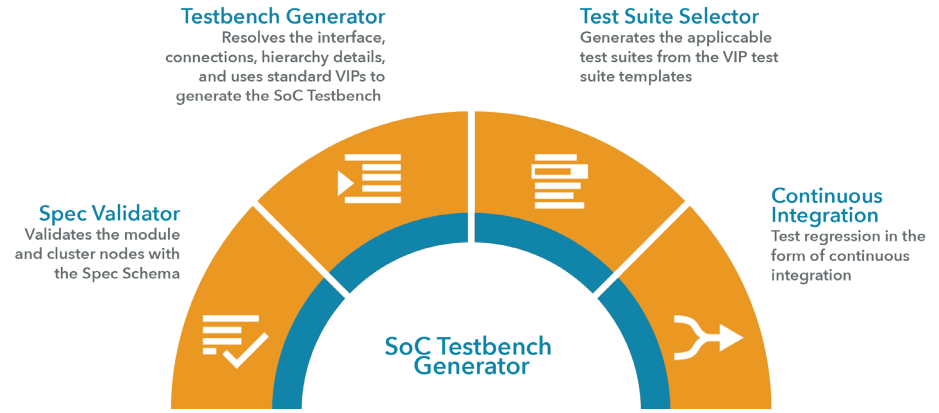


Figure 6: SoC TB generation components

VERIFYING SECURITY EXTENSIONS FOR RISC-V

Verification primarily aims at proving that a system does what it is supposed to do. Security, on the other hand, aims to ensure that the system does not do what it is not supposed to. Security has two aspects: the hardware, and the software. The hardware provides some features, and a "base of trust", which the software can exploit to provide security guarantees. Two (amongst many) of the main challenges with ensuring that a system is secure is the ambiguity in the definition of the system specification, and the closed source nature of the components used.

With the advent of RISC-V ISA, and availability of a plethora of open source processors, it opens new possibilities of verification where one can provide stronger guarantees from the hardware. Moreover, the simplicity of the RISC-V ISA makes it amenable to formal verification, and to that end, multiple formal versions of the specification are also available [1, 2]. Having a formal model eliminates the problem of ambiguity in the specification. However, verifying a complete RISC-V processor also involves defining a formal model of the microarchitecture (and proving equivalence between this and the formal model of the ISA specification), which is a herculean task. There do exist frameworks like Kami [3] that can aid in doing formal verification. This framework supports Bluespec [4] style hardware where the language itself has strong semantics which are amenable to formal verification.

Currently, RISC-V provides various security features at the hardware level, including Physical Memory Protection (PMP), modes (Machine, Supervisor and User) and virtual memory. There are also various task groups and individuals in the RISC-V community that are working on ISA extensions for cryptographic accelerators and true random number generators, hypervisor, secure boot, Trusted Execution Environment (TEE), and Control Flow Integrity (CFI). The verification of each of these requires special handling because each of these features are aimed at thwarting a particular set of attacks.

The verification plan for security in general can be quite complex because it is not enough to verify individual components, but it is also important to clearly understand the interactions between these components. Also, some of these solutions might depend on other features/extensions of RISC-V. In general, the verification can be carried out in 3 stages, and in each stage one would try to model various possible attacks, and check if the system is resistant against it. We will try to understand the same with an example of verifying a system that does AES encryption using a dedicated accelerator.

The first stage is a module level verification where one verifies individual modules. For an AES accelerator, a basic testing would involve providing streams of data for encryption/decryption and checking if the output is correct. Moreover, one could try to configure some of the configuration registers in an illegal manner and check if the behavior of the AES encryption module changes. It is also very important to note here that the verification of a particular module is only as good as the robustness of the specification. For example, if the specification does not mention anything about the timing properties of the accelerator, it might be vulnerable to side channel attacks. But if timing

properties are mentioned, one would have to provide test cases and verify if any of those properties fail.

The second stage is system level verification where you check if any vulnerabilities in other parts of the system might affect your module. In AES, it is important to keep the symmetric key secure. An implementation could have possibly assumed that PMPs are used to protect the various symmetric keys. A vulnerability in the PMPs could result in exposure of the symmetric key.

The third and the final stage is verifying the software driver. In most of the cases, a software vulnerability can render hardware protections futile. Hence, it is important to verify if the software has vulnerabilities in it. The possible vulnerabilities in the software again depends on the threat model.

Provable security for a complex system is an NP-hard problem, and hence one should always try to break it down to smaller problems and solve them first, and carefully stitch up the guarantees at various levels to provide security.

SUMMARY

InCore and Tessolve offer a comprehensive suite of IP and services to help customers build RISC-V solutions across a range of domains. This combination ensures the customer of the best possible combination of Cores, SoC fabrics, Verification IP, and Verification methodology to realize complex SoCs. Post verification services including physical design and post silicon testing is also part of the portfolio of services offered.

- IP Cores - Chromite CPU generator, AXI, APB, PLIC, CLIC, DMA, Cache blocks, DMA controller
- Peripheral IPs - SPI, QSPI, UART, PWM, PWM
- SoC dev pack
  - o Comprehensive SoC development environment
- Verification IP
- Verification Methodology - RiVer
- Silicon dev pack
  - o Convert RTL into a first tapeout success
- Product dev pack
  - o Build eval boards and embedded products

REFERENCES

[1] Sail RISC-V Model. https://github.com/rems-project/sail-riscv
[2] Forvis RISC-V ISA Spec. https://github.com/rsnikhil/Forvis_RISCV-ISA-Spec
[3] Kami framework. https://plv.csail.mit.edu/kami/
[4] Bluespec System Verilog. https://bluespec.com/technology/

Mentor®
A Siemens Business

www.mentor.com

Editor:
Tom Fitzpatrick

Program Manager:
John Carroll

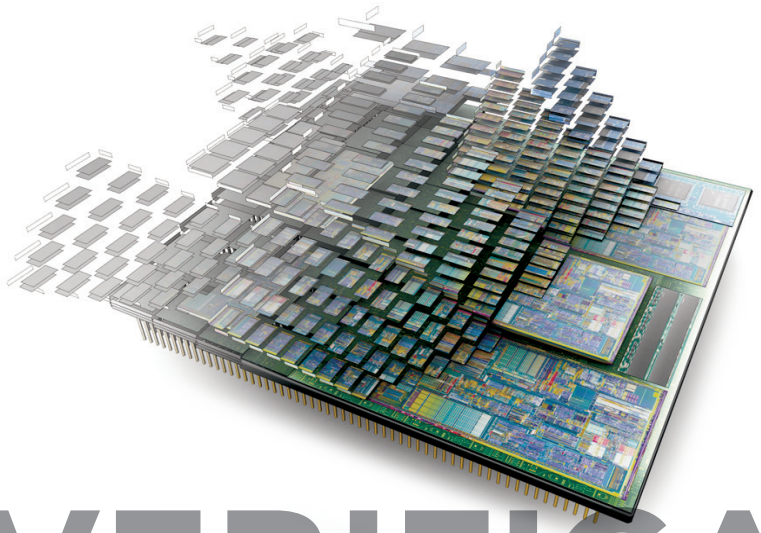
Mentor, A Siemens Business
Worldwide Headquarters
8005 SW Boeckman Rd.
Wilsonville, OR 97070-7777

Phone: 503-685-7000

To subscribe visit:
www.mentor.com/horizons

To view our blog visit:
VERIFICATIONHORIZONSBLOG.COM

Verification Horizons is a publication
of Mentor, A Siemens Business



VERIFICATION HORIZONS

VH