Automation and Reuse in RISC-V Verification Flow

by Marcela Zachariasova and Lubos Moravec, Codasip Ltd

The Open RISC-V Instruction Set Architecture (ISA) managed by the RISC-V foundation\(^1\) and backed by an ever increasing number of the who’s who in the semiconductor and systems world, provides an alternative to legacy proprietary ISA’s. It delivers a high level of flexibility to allow development of very effective application optimized processors, which are targeted to domains that require high performance, low area or low power.

The RISC-V ISA standard is layered and contains a small set of mandatory instructions as well as optional instruction set extensions, and finally, custom instructions defined by the intended application. As a result, when searching for the best functionality-performance combination, we can end up with at least 100 viable standard ISA configuration variants, and nearly unlimited combinations when custom extensions are taken into account. This flexibility is without a doubt a good thing from the system design perspective, but generates significant verification challenges that will be discussed throughout this article.

In traditional processor IP development, the supplier verifies a small number of core variants which customers must then use as they are. To exploit the flexibility offered by RISC-V we need to find a way how to verify a rich selection of application and customer-specific variants. However, in order to meet the time-to-market and quality demands of customers, we must consider alternative and enhanced verification approaches.

Therefore, in this article Codasip proposes a technique based on automated generation of a hardware representation for any/every RISC-V core variant (at the Register-Transfer Level, RTL), automated generation of tools for software development (Software Development Kit, SDK), automated generation of UVM verification environments and at last but not least, an effective reuse of assertions, coverage points and directed tests among verification environments. It is also shown how Mentor’s Questa® Simulator, Questa® VRM, Questa® VIP and Questa® Autocheck contribute to achieving quality of the resulting IP products.

CHALLENGE STATEMENT

RISC-V is a free to use, modern and open ISA under the governance of RISC-V Foundation already having rich member base to support industry implementations and education.

The only mandatory part that must be present when implementing a RISC-V compliant processor is the base integer instruction set (“I” for 32 general purpose registers or “E” for 16 general purpose registers). Further optional ISA extensions that can be implemented are: multiplication and division extension (“M”), compressed instructions extension (“C”), atomic operations extension (“A”), floating-point extension (“F”), floating-point with double-precision extension (“D”), floating-point with quad-precision extension (“Q”), and other experimental and custom extensions. Hardware extensions are not defined in the standard but are usually present for the purposes of hardware debugging (JTAG), improving performance (parallel multiplier, divider) or jump optimizations (jump predictors). There is also a provision in the RISC-V specification allowing addition of custom instructions that are specific and advantageous to a particular domain (such as security, image and audio processing, etc.).

Codasip is a Czech-based IP company and our commercial implementation of a RISC-V compliant core is the Codix Berkelium (Codix-Bk) series of processors\(^2\). Codix-Bk contains configurable standard ISA extensions and a highly automated way to add custom ISA extensions or micro-architectural optimizations. While this configurability rapidly accelerates the IP delivery times and allows customization, the question arises how to effectively verify all the RISC-V IP variants.

46 mentor.com
THE VALUE OF AUTOMATION
Codasip enables delivery of customized processor IPs in the order of days, due to the high level of automation available within our processor development environment called Codasip Studio\(^2\). Codasip Studio can be used both for optimizing standard processor cores like Codix-Bk as well as for creating new unique processor implementations. It is an Eclipse-based environment in which a processor is described at the higher abstraction level in the Architecture Description Language called CodAL. In particular, two parts of the description are needed: the instruction-accurate (IA) model describing instructions (ISA) and their behavior without particular micro-architectural details, and the cycle-accurate (CA) model describing micro-architecture like pipeline, decoding, timing, etc. From these two high-level models, Codasip Studio will automatically generate all the deliverables needed by both the hardware and software development teams.

Automation eliminates the sources of errors introduced by manual design, and ensures that when a problem is identified and fixed, it is fixed for all future generated deliverables. This ensures that the generated design starts off at a high level of quality, but does not eliminate the need to perform comprehensive verification on every unique implementation.

VERIFICATION AUTOMATION
From a verification perspective, the generated RTL for a selected RISC-V variant serves as the Design Under Test (DUT). Furthermore, the Codasip automation flow generates a UVM verification environment (from both IA and CA models) together with the reference model (generated from the IA model only, see Figure 2) and assembler programs for the processor (generated from the IA model only, sometimes called tests as in Figure 2).

Generated deliverables and resources (captured in Figure 1) include assemblers and compilers (both generated from the IA model), simulators, debuggers and profilers (all three are available in the IA and the CA variant), and even RTL in VHDL, Verilog, SystemVerilog and SystemC (generated from the CA model only, see also Figure 2).
For every generated assembler program, DUT outputs are automatically compared to those of the reference model. It can be deduced that the generated verification environment is processor-specific and for the purposes of UVM generation, all the necessary information is extracted from the high-level processor description in CodAL.

The generated verification environment has a standard, UVM-style structure (it is illustrated in Figure 3). What are not depicted in Figure 3 are the generated assertions and functional coverage points. Assertions are checking communication protocols on interfaces, correct decoding of instructions, detection of unknown instructions, propagations of undefined values, etc. Functional coverage points check for the occurrence of all instructions and their valid combinations, occurrence of all valid types of communication events on buses, uniform or non-uniform approaches (reads and writes) to registers, etc.

Cooperation of components depicted in Figure 3 is as follows. In the preparation phase of verification, programs (that will be running on the processor) are loaded to the program part of the memory by the Program Loader component in Memory Agent. During verification, data is driven to the inputs of the Codix-Bk processor by Driver of Processor Agent and after their processing, outputs are read from the Codix-Bk processor by Monitor in Processor Agent. Output data is automatically compared to those of Reference Model. For more precise verification, transactions on buses, values on output ports, the content of memories and registers can be compared to their counterparts within Reference Model. The comparison is possible every time program data is completely processed or even during processing of the program - for example, after every instruction. For reading the content of registers and the memory, Monitors in Registers Agent and in Memory Agent are used.

We utilize QuestaSim RTL simulation to execute this UVM environment. To further validate the quality of the generated RTL we also utilize Questa® Autocheck static formal analyzer. This ensures not only that the design is functionally correct, but also structurally correct.

**HIERARCHICAL INTEGRATION OF VERIFICATION COMPONENTS**

Verification engineers can connect additional verification components to the generated UVM. For example, we support dedicated hardware units in processors such as FPU, dividers and multipliers in various implementations, jump predictor, etc. A verification engineer may generate the UVM testbench for just these units in Codasip Studio to check functionality before moving to debugging of the whole processor. Afterwards, when the UVM environment for the full processor is generated, the UVM environment for the smaller internal unit with all the user manual enhancements will be automatically integrated.

In the case that a verification engineer wants to increase the verification detail within existing agents, we recommend the addition of new coverpoints and assertions to the generated UVM or the creation of simple patches.

The third scenario could be integration of the generated UVM environment into a more complex SoC verification environment. Thanks to UVM scalability, and the style of the generated verification environment, this can be easily achieved. A typical approach is to instantiate the UVM processor environment class into a hierarchy of UVM components. For example, when we know that the core will be connected by a specific bus to
surrounding peripherals or other cores, we can use Mentor Verification IP (VIP) components as protocol checkers.

**REUSE STRATEGY**

While Codasip aims at automating verification as much as possible, it is hard to automatically generate all the detailed checks, especially some of those that are specific to a unique processor implementation. Such details and corner cases are usually listed in the verification plan and when following the items of this plan, it may be necessary to add some hand-written parts to the generated verification environment.

Such additions can be represented by new directed tests (manually written programs or applications for the processor), assertions (for example, checking hazards in the pipeline, checking timing of operations such as waking from the sleep mode or duration of multiplication, exceptions handling, etc.), and functional coverpoints (for example, combinations of interrupts, co-appearance of interrupts and hazards, etc.).

Having RISC-V processors in mind, the RISC-V Foundation provides the ISA standard from which verification details can be easily extracted and converted into the verification plan and consequently to hand-written additions. It is even possible to categorize them for better handling, for instance, there can be assertions, coverage points, directed tests targeted to integer (I/E) instruction set, multiplication (M) instruction set or to a specific functionality of the core like interrupts or hazards.

Therefore, instead of modifying the UVM generator for specific scenarios in a unique core, a better strategy is to enable an effective reuse of manually written parts of the verification environments. Let’s demonstrate the idea on a series of examples dedicated to a unique Codix-Bk (RISC-V processor) variant.

It is beneficial to start with a more complex variant of the core, for example, we can select Codix-Bk5 with 5 pipeline stages and configure I+M+F ISA and 32 general purpose registers. After the configuration is chosen, tools and resources are automatically generated from CodAL using Codasip Studio (see the arrow GENERATE in Figure 4). From the verification perspective, we now have RTL, reference model, UVM and assembler (ISA) tests. If we find some mismatches between RTL and the reference model during verification (for example, different data in general purpose registers, in memory, on ports or in transactions on buses), we must correct the CodAL model and regenerate the resources (see the arrow MISMATCHES FOUND, MODEL MUST BE FIXED in Figure 4). If results of comparison are ok, but coverage is not 100% it means that some corner-cases are being missed by the randomly generated programs (see the arrow RESULTS OK BUT COVERAGE NOT 100% in Figure 4). In this case, we usually manually prepare some directed tests in order to increase coverage. We can also add more coverage points or assertions for the corner cases. Finally, when coverage is high enough (see the arrow RESULTS OK, COVERAGE 100% in Figure 4), the verification is finished.

When another variant of the Berkelium core is needed, let’s consider now a simpler variant of the more complex core (subset of Codix-Bk5 functionality from the previous example), we just configure only the wanted extensions and re-generate the resources (see the arrow GENERATE in Figure 5). The automation flow is the same as in the previous case, except that it is unlikely to find any further mismatches, because we already tuned them in the more complex core variant (see the
stroked arrow MISMATCHES FOUND, MODEL MUST BE FIXED in Figure 5). As for the manually written parts, this is the right moment when we should have the reuse strategy in mind in order to significantly reduce the complete verification cycle, and focus on only those parts that have been changed. So when targeting a high level of coverage we can reuse hand-written parts effectively and just reuse them from the more-complex variant.

Finally if a non-standard extension of a RISC-V core is required, for example, adding a new instruction to optimize performance, or adding a hardware block to improve debugging, it is possible to do such extension in CodAL and again, benefit from automation and reuse. Let’s see how it works. We extend the CodAL model of Codix-Bk and regenerate the resources (see the arrow GENERATE in Figure 6). We may need to fix some bugs, but it is likely they would be connected to the new extensions, as the standard parts have already been verified (see the arrow MISMATCHES FOUND, MODEL MUST BE FIXED in Figure 6). Then we target coverage closure - we can again reuse tests, assertions, coverage points from the more-complex version for the standard parts of the core and only add new ones connected to the new extension.

CONCLUSION

Automation in generating RTL, UVM verification environments, reference models and tests (programs) can rapidly improve productivity in the development of RISC-V cores. The advantage of this approach is that we do not spend valuable time on coding, but rather on debugging problems and exploring corner cases. We showed two use-cases where we reuse manually written parts of the verification environment: 1) when verifying a simpler variant of a more complex RISC-V core, 2) when adding a new user-extension.

A few time estimations were already provided in the article, but to present a bigger picture, Table 1 and Figure 7 summarize the overall time savings in comparison to the standard application-specific processor development flow.

Processor design: as you can see, if we are making a new processor by Codasip tools, after approximately 35 days spent on creating a high-level processor model in CodAL, we spent only a few minutes by generating SDK, RTL, UVM and assembler tests. In the standard flow, months are burned as all these tools and resources must be written by hand. The difference is even bigger in the RISC-V compliant processor development. If we want to integrate a new user extension to the already verified standard configuration of the RISC-V core, we spent only around 10 days on CodAL re-design (it always
depends on the difficulty of the extension). If we want a subset of already verified RISC-V core, we just change the configuration. Generation of SDK, RTL, UVM and random tests is always done in minutes.

Following the verification plan always takes time, because we usually implement additional directed tests, coverpoints or assertions to the generated UVM environment. Approximately, we spend 2 months running tests and debugging a new RISC-V design. However, when standard configurations of the RISC-V core are demanded, we can shorten the verification process by an effective reuse to a few days.

REFERENCES

<table>
<thead>
<tr>
<th></th>
<th>Standard flow</th>
<th>Codasip + Mentor</th>
<th>RISC-V with a new extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADL processor model</td>
<td>0 days</td>
<td>35 days</td>
<td>10 days</td>
</tr>
<tr>
<td>SDK</td>
<td>180 days</td>
<td>avg. 5 minutes</td>
<td>avg. 5 minutes</td>
</tr>
<tr>
<td>RTL</td>
<td>180 days</td>
<td>avg. 1 minute</td>
<td>avg. 1 minute</td>
</tr>
<tr>
<td>UVM Environment</td>
<td>90 days</td>
<td>avg. 2 minutes</td>
<td>avg. 2 minutes</td>
</tr>
<tr>
<td>ISA tests</td>
<td>30 days</td>
<td>avg. 2 minutes</td>
<td>avg. 2 minutes</td>
</tr>
<tr>
<td>Micro-architectural tests</td>
<td>30 days</td>
<td>30 days</td>
<td>5 days</td>
</tr>
<tr>
<td>Verification and debugging</td>
<td>60 days</td>
<td>60 days</td>
<td>15 days</td>
</tr>
</tbody>
</table>

Summary
570 days 125 days 30 days

Figure 7. Development time savings in comparison to standard flow
VERIFICATION ACADEMY
The Most Comprehensive Resource for Verification Training

30 Video Courses Available Covering
- UVM Debug
- Sequential Logic Equivalence Checking
- Portable Stimulus Basics
- SystemVerilog OOP
- Formal Verification
- Intelligent Testbench Automation
- Metrics in SoC Verification
- Verification Planning
- Introductory, Basic, and Advanced UVM
- Assertion-Based Verification
- FPGA Verification
- Testbench Acceleration
- PowerAware Verification
- Analog Mixed-Signal Verification

UVM and Coverage Online Methodology Cookbooks
Discussion Forum with more than 8100 topics
Verification Patterns Library

www.verificationacademy.com