A compact functional verification flow for a RISC-V 32I based core

Roberto Molina-Robles*, Edgar Solera-Bolanos*, Ronny García-Ramírez*,
Alfonso Chacón-Rodríguez*, Alfredo Arnaud† and Renato Rimolo-Donadio*
*Escuela de Ingeniería Electrónica, Tecnológico de Costa Rica
†Depto. de Ingenieria Electrica, Universidad Catolica del Uruguay
{rmolina, esolera ,rgarcia, alchacon, rrimolo}@tec.ac.cr
aarnaud,@ucu.edu.uy

***Abstract*—The structure of a functional verification flow used for the design of a RISC-V core is presented. The paper offers a guide on the test-planning used and details of the flow architecture, showing how to integrate the Universal Verification Methodology with the required, reference models, while implementing key futures in standard verification environments, such as testing regressions and code and structural coverage. The designed flow is compact yet efficient, making it affordable for small design teams, without requiring extra investment other than the already necessary licenses for RTL synthesis and the eventual fabrication of the chip.**

***Index Terms*—Functional Verification, RISC-V 32I, UVM, SystemVerilog, EDA tools, architecture, test generation, processor, compiler, simulation, coverage, regression, reference model.**

I. Introduction

Functional verification is hardly a new topic. There is a solid standard [1], and plenty of teaching resources available (see for instance [2] for a comprehensive site with plenty of free resources). And there exists some recent literature with examples that can provide guidance to non-experienced verification teams. Some works, for instance, show how to implement verification environments with or without UVM, such as [3], where a custom environment was proposed to improve coverage, or [4] and [5], where UVM was applied to different RTL blocks. Yet, most of the documentation available points towards the use of massive, highly integrated frameworks attached to a particular methodology—typically derived from the Universal Verification Methodology (UVM)—, where the access to commercial tools and personnel is not a limitation, and the departure specifications for the expected results are readily defined (or at least, do not depend directly on the verification team itself). Yet, small design teams usually do not have the budget to tackle the verification problem using such an approach, particularly when the design and verification process must be handled by the same people. This means finding ways to optimize hardware and software resources, through the use of open source tools whenever possible, while providing with a flexible environment that can easily be migrated from a project to another. Now, some examples of small teams using functional verification for their chips approach can be found. Yet, to our knowledge, most of the goal specifications in those papers were already defined by the use of a standard given architecture (SiFive's Rocket-chip), written in Chisel. This was not the case here, being Siwa an architecture written from scratch, with several modifications from the RISC-V 32I standard mandated by the ultra low power specifications of its intended application. This paper's main purpose is to document the implementation of a functional verification environment used for the pre-silicon verification of a RISC-V 32I based processor, called Siwa, developed by a small team of only six people as the main controller core of a medical implantable tissue stimulator (see [6] for details on the processor and the medical device system itself). The present work gathers strategies as those used in the previous references, and incorporates them not only for the verification of architectural blocks, but also, adds up the use of reference models, the integration of custom architectures and the automation of regressions for random verification, topics often missing in the literature.

As such, the structure here presented allows for functional verification at different hierarchical levels, using oriented and random-constrained tests, based whether on custom reference models or requiring the incorporation of code simulators serving as golden vectors generators, while using regression systems for extending code coverage. All these in an environment compact and versatile enough for a verification team of only three people.

This paper is organized as follows: Section II describes the tools and resources selected for this work. Section III details the establishment of the verification plan. Section IV describes the proposed verification flow. Section V presents the results. Lastly, Section VI highlights the main conclusions of the work and discusses some future work.

II. Definition of Tools and Resources

Small IC design teams with modest financial resources typically require affordable yet competitive, low maintenance tools. Yet, one cannot always resort to the open source community for such tools, although there is a strong movement pushing in that direction of open hardware design, as for instance the Linux Foundation CHIPS Alliance. There is also a lack of technology kits for open design tools, as foundries base their production environments on the three major EDA

978-1-7281-3146-7/20/$31.00 ©2020 IEEE

providers: Mentor Graphics, Synopsys and Cadence. Particularly, our group has a Synopsys license available. In the case here presented, selecting SystemVerilog (SV) as specification language, and UVM as the verification methodology was straightforward. SystemC was discarded due to its lack of support from the Synopsys synthesis tools and the libraries flow provided by the foundry for the project, with the added extra of SV already having a UVM library incorporated, and being supported by most of compilers/simulators. This meant using Synopsys VCS for the compilation, simulation and coverage processes and Synopsys DVE as the Waveform Viewer and Coverage Analyzer of choice. This mainly because of the tools' easy interfacing with the Design Compiler and IC Compiler flows. Alternative simulators (such as Verilator) and wave analysis tool (GTKwave), nonetheless may be used.

Concerning the setup of the regression platform, Linux's Cron utility and basic Bash scripting were selected, even though there are other options such as Jenkins and Bamboo that are typically more favored by industry because of their wide array of features and support for large development teams. Cron, nonetheless, is easier to setup and less demanding in terms of computation resources.

III. First Steps: Verification from scratch

Before addressing the verification architecture, a road-map is first created. This means studying the design via specification documents and industry standards, and coordinating with the core's architects and designers. Having adequate knowledge about the functionality of the chip to be verified accelerates the verification process. Coding can be tackled once the verification plan is ready. Methodology, verification architecture, test plans, resources and tools, time-lined efforts, coverage points and others aspects are expected sections of the verification plan, and should be reviewed several times with architects, designers and other verification engineers.

Regarding this work's implementation, the environments were custom built for a processor based on RISC-V 32I standard [7], focused on medical applications [6], and most of them developed under the UVM standard.

IV. The Verification Architecture

Since the processor was small, verification efforts were into only two hierarchical levels. The lower level for block verification and the higher level for chip verification. A simulation-based verification flow must include several characteristics, which can be found here [8].

A. Block-Level Verification

The selected blocks submitted for verification were: an Arithmetic-Logic Unit (ALU), a Memory-Bus Controller (MBC), a System Bus and a Universal Asynchronous Receiver-Transmitter port (UART). The ALU and the UART blocks were designed using a standard register-based RTL approach, with minor custom modifications. The MBC and the bus used a latch-based micro-architecture, for area and power reduction. Either way, for block-level verification then, UVM was used to implement the verification architectures. The remaining block was an SPI module that was stimulated at chip-level, however, its complete verification will be done in further spins. A Black-Box philosophy was selected for the block-level verification efforts. In [9], there is a clear explanation of the advantages of this type of verification. Figure 1 shows the generalized block diagram for the block-level architecture, based on [1].

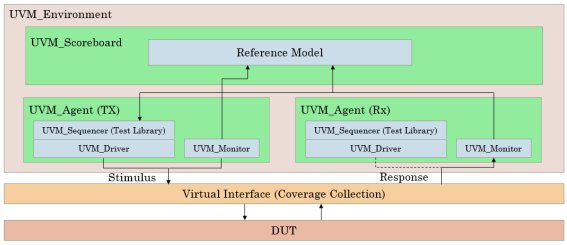

Figure 1. A testbench implemented using a UVM architecture for block-level verification. The quantity of agents varies depending on the modularity of the design block interfaces. The sequencer calls sequences built with sequence items to form a test. The desired test is called from the command line.

There were small variations in the architecture's implementation for each individual block, but the general idea remains the same. For example, the TX Agent and RX Agent from Fig. 1 were here fused into a single agent for the ALU and the UART, since those blocks had few interfaces and their functionality were simple enough. In contrast, the MBC and the Bus had more than two agents because of their several interfaces. This type of modifications are supported inside the UVM standard, but one must remember that the less agents one has, the easier its environment construction but the harder its maintenance. The ALU and the UART are blocks with standard interfacing, not prone to change in further micro-architectures, and thus, agents may be considered stable as well. Meanwhile, the MBC and Bus are blocks that may change depending on features that might be added in further spins. If one separates the interfaces in features/devices and connect an individual agent to it, then one can only adjust the associated agent instead of modifying a more complex agent connected to all interfaces at once.

At the block-level, the reference model used was transaction-based [9]. That is, that at every transaction between the verification environment and the DUT, a checking is being made between the custom reference model's prediction and the actual results of the DUT. This reference model was implemented in SV inside the scoreboard, based on a written specification to avoid similarities with the RTL implementation as much as possible.

B. Chip-Level Verification

Chip-level verification means that the DUT is the complete RISC-V core. Here, a first consideration is that the core runs programs that cannot be totally randomized: for instance, memory addressing instructions have limited valid ranges, and certain registers have pre-defined functions. Second, the core has concurrent interfaces (UART, SPI and GPIO). Third,

the easiest way to debug a processor is looking into the core's register bank. However, none of these registers can be externally accessed, except indirectly via the UART or SPI ports. Finally, the moment when the checking with the reference model occurs must be selected carefully according to the micro-architecture; that is, that the time needed to execute instructions varies depending on the core structure (simple scalar multicycle, pipelined scalar, pipelined superscalar, etc.).

These restrictions imposed a Grey-Box approach as the chip-level verification philosophy, where the register bank is accessed via a backdoor. A "Golden Reference" methodology was selected, as recommended by [9], although it is possible to use a transaction-based reference model if the checking occurs at the end of each instruction cycle. Test program generation was handmade at this level. Figure 2 shows the custom architecture of the chip-level verification environment.

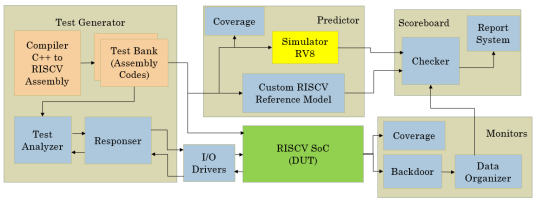

Figure 2. Custom architecture implemented for pre-silicon testing of a RISC-V core. The test generator calls a program created with the compiler and loads it inside the DUT. The responser communicates with the DUT if it is instructed in the program. Monitors are connected via backdoors to registers and specific control signals. Blue blocks and the DUT were written in SystemVerilog.

A program is loaded and executed in Siwa's RTL model. Information from the data and control-status registers (CSRs) information is stored in an array at the end of each instruction cycle. Simultaneously, a reference model predicts the correct result that ought to be stored in each register for every instruction, and stores it as well in another array. The final data arrays are compared after the testing program ends.

C. Generation of the Reference Model for Chip-Level Verification

Contrary to the block-level reference models, custom built using a specification document, a standardized reference model was used for chip-level verification. In order to build such custom reference model, the logic described in Fig. 3 is followed. The RISC-V simulator RV8 [10] is used in tandem with a custom reference model written in SV, validated itself against RV8 simulation results. The goal was to construct a model capable of predicting results stored into the register bank for each instruction. The custom reference model is used for Siwa's custom operations that do not follow the RISC-V standard; specifically: a smaller set of CSR and a restricted memory map (8kB).

D. Test Generation

The test generation was handled differently depending on the hierarchical level. All tests for block-level DUTs were randomly generated and constrained. The verification plan

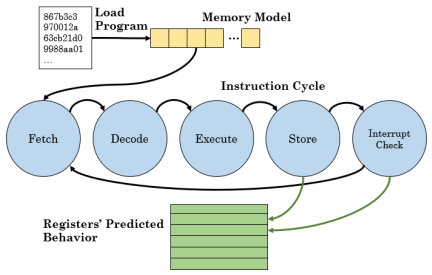

Figure 3. RISC-V reference model flow diagram. This model is called before loading the Flash memory. The predictor also helps to determine when a test should end, if the RTL does not reach this point, it means that something went wrong. Each of these steps conforms a typical instruction cycle.

previously designed specifies the necessary tests. For instance, a test for each arithmetic-logic functionality of ALU was implemented, for each type of transactions through the UART at different speeds and configurations. Read and write tests for the MBC and tests that emulate data flow traffic through the Bus were also created.

At chip-level, however, random instruction generation was more complicated. Since instructions must follow the ISA standard and the program needs a coherent intention to be able to run an application, the tests were completely oriented, hence, the random factor was taken out. Test selection comes from the verification plan. This meant tests were implemented for each instruction in the RISC-V 32I ISA [7], for each port and its intended functionality, for each type of core interrupt, and tests that increase coverage over the register bank and memory space were also needed. Each of these tests were written as individual programs to be loaded and run into the core. These tests were handmade and developed using a C Assembler compiler, from the Sifive Toolchain [11], [12]. Fig. 4 shows a diagram the tests order of execution, as they feed the verification environment.

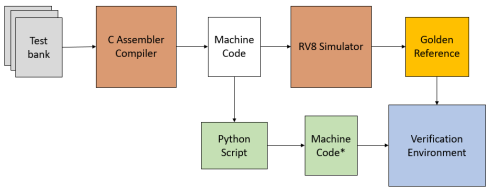

Figure 4. Diagram that depicts how tests are handled prior the beginning of the simulation. Machine code marked with a * has been edited with a Python script, in order to match the Flash memory model requirements.

The resulting text file with the program was modified afterwards via script to adjust it, so the SV model of a Flash memory could read it. Then, the core boots from that Flash memory connected to a SPI (Serial Peripheral Interface) port. Finally, the rest of the simulation and the checking occurs as explained on the previous sections.

E. Regression System

Regressions in functional verification are similar to test farms, where multiple tests are run one after another. Some are pseudo-random, and therefore, are linked to a seed. Others, are oriented tests running a particular important feature. The proposed flow integrated a regression platform, depicted in Fig. 5.

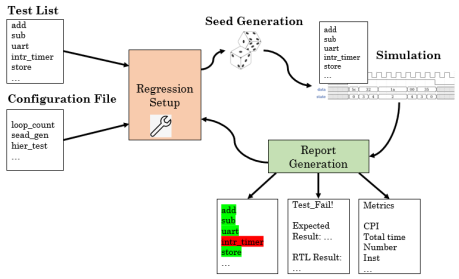

Figure 5. General structure of the regression system. Each test is run according to a list of tasks, with a configuration file defining specific aspects of the regression, such as loop quantity and type of coverage collected.

Basic shell scripting was used for coding the regression platform, following the general structure shown in Fig. 5. Seeds are generation using Python. For oriented tests at chip-level, the RISC-V compiler is invoked for test generation with RV8 providing the golden reference construction, prior to invoking VCS for simulation and coverage collection. A report is created or updated with the resulting status for each test. Once a batch of tests is finished, a new batch is prepared and executed using a different seed. The required number of iterations is specified by the user, according to the goal in terms of coverage.

V. Results

The proposed verification structure is fully functional and can be replicated to equivalent designs. More than 100 tests were implemented in total, some of them with multiple seeds, for at least 5 different DUTs with independent functional and structural coverage metrics.

An example of the regression console is given in Fig. 6. Several reports are generated by the regression, including information such as: test status with its respective seed, performance parameters such as execution time or Clock Per Cycle (CPI), and comparisons between the predicted and the RTL register bank for each individual test.

Figure 7 shows an example of coverage results, extracted from the Synopsys DVE coverage tool.

VI. Conclusions

A compact, affordable functional verification flow has been generated and used for the verification of a small RISC-V 32I based micro-controller, as a base case. The flow follows a functional verification strategy the includes the development of a test methodology and verification architecture, and is capable of carrying out hierarchical verification, reporting coverage metrics and performing intensive, randomized regressions.

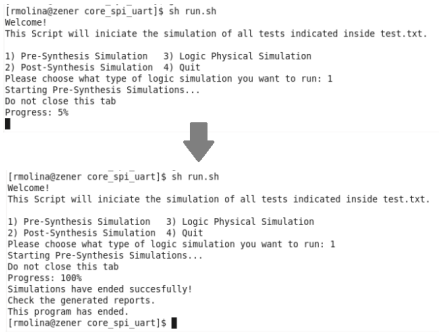

Figure 6. Screenshot of the regression console. Several reports are generated, and the execution is scheduled via Linux's Cron.

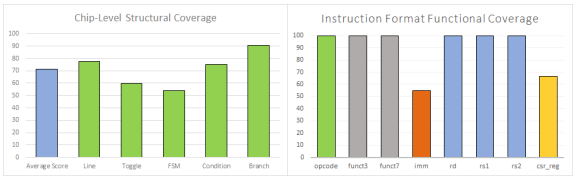

Figure 7. Accumulative coverage results obtained with the custom verification environment. The graph to the right shows the functional coverage of the RISC-V instruction formats [7]. The graph to the left depicts the average structural coverage obtained for chip level verification.

The flow can be extended to other digital designs and can incorporate alternative tools if required. Future works include adding the option for formal verification tests to the flow, and the possibility of generating random RISC-V coherent programs.

References

[1] *Universal Verification Methodology (UVM) 1.2 User's Guide*, Accellera, 2015.

[2] Doulos. (2019) Uvm knowhow. [Online]. Available: https://www.doulos.com/knowhow/sysverilog/uvm/

[3] R. Yang, L. Wu, J. Guo, and B. Liu, "The research and implement of an advanced function coverage based verification environment," in *2007 7th International Conference on ASIC*, Oct 2007, pp. 1253–1256.

[4] V. B and B. Bala Tripura Sundari, "Uvm based testbench architecture for coverage driven functional verification of spi protocol," in *2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, Sep. 2018, pp. 307–310.

[5] T. M. Pavithran and R. Bhakthavatchalu, "Uvm based testbench architecture for logic sub-system verification," in *2017 International Conference on Technological Advancements in Power and Energy (TAP Energy)*, Dec 2017, pp. 1–5.

[6] R. García, A. Chacón, R. Castro, A. Arnaud, M. Miguez, J. Gak, R. Molina, G. Madrigal, M. Oviedo, E. Solera, D. Salazar, D. Sánchez, M. Fonseca, J. Arrieta, and R. Rimolo, "Siwa: a RISC-V platform in a 0.18μm HV CMOS process for implantable medical devices," submitted.

[7] A. Waterman and K. Asanovic, *The RISC-V Instruction Set Manual Volume I: User-Level ISA*, SiFive Inc., CS Division, EECS Department, University of California, Berkeley, 5 2017, an optional note.

[8] "Ieee standard for the functional verification language e," *IEEE Std 1647-2016 (Revision of IEEE Std 1647-2011)*, pp. 1–558, Jan 2017.

[9] B. Wile, J. Goss, and W. Roesner, *Comprehensive Functional Verification: The Complete Industry Cycle (Systems on Silicon)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[10] Risc-v simulator for x86-64. [Online]. Available: https://rv8.io/

[11] Risc-v gnu toolchain. [Online]. Available: https://github.com/sifive/riscv-gnu-toolchain

[12] Risc-v elf to hex converter. [Online]. Available: https://github.com/sifive/elf2hex