A RISC-V SystemC-TLM simulator

Màrius Montón
marius.monton@uab.cat
Departament de microelectrònica i sistemes electrònics
Universitat Autònoma de Barcelona
Barcelona, Spain

ABSTRACT

This work presents a SystemC-TLM based simulator for a RISC-V microcontroller. This simulator is focused on simplicity and easy expandable of a RISC-V. It is built around a full RISC-V instruction set simulator that supports full RISC-V ISA and extensions M, A, C, Zicsr and Zifencei.

The ISS is encapsulated in a TLM-2 wrapper that enables it to communicate with any other TLM-2 compatible module. The simulator also includes a very basic set of peripherals to enable a complete SoC simulator. The running code can be compiled with standard tools and using standard C libraries without modifications.

The simulator is able to correctly execute the riscv-compliance suite. The entire simulator is published as a docker image to ease its installation and use by developers. A porting of FreeRTOSv10.2.1 for the simulated SoC is also published.

CCS CONCEPTS

• **Computer systems organization → Embedded hardware**; **High-level language architectures**; • **Hardware → Simulation and emulation**.

KEYWORDS

RISC-V, SystemC, TLM-2.0, Simulation Infrastructure, ISS

ACM Reference Format:
Màrius Montón. 2020. A RISC-V SystemC-TLM simulator. In *Fourth Workshop on Computer Architecture Research with RISC-V (CARRV 2020).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Many simulators has been published since the release of first drafts of RISC-V ISA [8]. These simulators use different techniques and technologies to achieve different requirements: good performance, good visualization of the processor, architectural exploration, etc. Most of them conform to RISC-V ISA specifications; some of them use a previous infrastructure and adapt the ISS to follow the RISC-V ISA and re-uses some peripherals already simulated [3, 4, 7, 19]. Others are written from scratch and includes the ISS and a minimum set of peripherals [6, 11]. There are FPGA-based simulators to increase performance and simulation speed [9] as well as the precision of the simulation results.

The Spike simulator is most common simulator and it is used as reference model for RISC-V ISA [6]. Other simulators are intended for a graphical visualization for the entire execution of the instructions inside the CPU [15].

SystemC is a set of libraries for the C++ language to allow the description and simulation of hardware based systems by a event-driven simulation model. This libraries add time management, concurrency and hardware-like data types to C++ [1].

Transaction Level Modelling adds a layer to SystemC in order to model the interface between different modules in a lightweight way. This model technique uses transactions to abstract any kind of communication between modules, hiding or avoiding the details of the communication itself: a transaction is an access from a Master (called Initiator) to a Slave (called Target) to a memory address with a length and some attributes. The Slave will respond to the transaction within a time (that can be 0 for a basic modeling) and the writing or reading of the transaction. All other details of the transaction (bus access, signals change, etc.) are not modeled. In more detailed modeling, the different phases of a bus access can be specified. Currently, SystemC standard includes TLM modeling [1]. The modules can also interchange data using direct pointers to memory instead to transactions to increase simulation speed. This technique is named Direct Memory Interface (DMI).

TLM has boosted the interoperability between vendors models and the appearance of many IPs that are interchangeable and fully compatible among different systems and vendors. The fundamental idea of this work is to introduce all these features to a RISC-V simulator.

The source code of the entire project is open-source and published [14].

The presented simulator is intended for an easy use and simple to extend, with clear code and able to simulate an entire SoC, like any embedded microcontroller in the market. To keep the code simple, meta-programming has been avoided and C++ templates use is keep as low as possible.

The paper is structured in the following sections: Section II depicts the architecture of the entire simulator, Section III show software particularities and tool-chain modifications, Section IV shows simulation performance and compliance results. Section V concludes the paper.

2 SIMULATOR ARCHITECTURE

One of the main goals of this simulator was to be easily extensible and modifiable. To achieve this objective, the original design was very simple and clear, with the use of naive techniques and a source code designed for simplicity.

CARRV '20, May 30, 2020, Valencia, ES

ACM ISBN 978-1-4503-XXXX-X/18/06…$XX
https://doi.org/10.1145/nnnnnnn.nnnnnnn

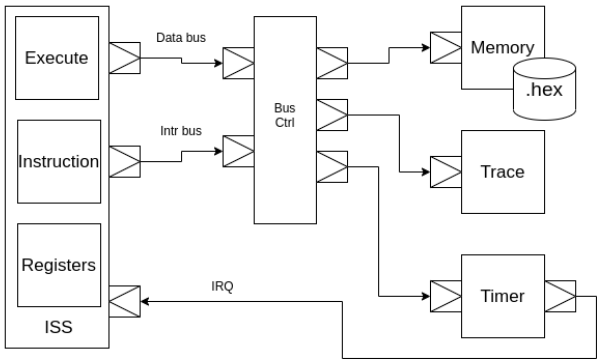

Figure 1: TLM Diagram of the entire simulator

The simulator architecture includes a ISS for RV32I ISA [20], a bus controller, the main memory and peripherals. Communication between these modules is done by TLM-2 sockets (see Figure 1).

2.1 CPU

The ISS simulates a single hardware thread (HART) and includes privileged instructions. It is divided in three modules: *Instruction*, *Execute* and *Registers*:

- *Instruction* Decodes instructions and checks for extensions. This module can access all fields of each instruction type (R, I, S, B, U and J type).
- *Execute* Executes instructions, accessing registers and memory and performing operations. This module also executes "MACZicsr_Zifencei" extensions [20].
- *Registers* Implements the register file for the entire CPU, including general-purpose registers (r0-r31), Program counter (pc) and all necessary entries in Control and Status Registers (CSR) registers.

This CPU is a minimal, fully functional model with a end-less loop fetching and executing instructions without pipeline, branch predictions or any other optimization technique. All instructions are executed in one single cycle, but it can be easy customized to per instruction cycle count.

The *Execute* module implements each instruction with a class method that receives the instruction register. These methods perform all necessary steps to execute the instruction. In case of a branch instruction, these methods are able to change the PC value. For Load/Store instructions, the methods are in charge to access the required memory address.

The CPU is designed following Harvard architecture, hence the ISS has separate TLM sockets to interface with external modules:

- Data bus: Simple initiator socket to access data memory.
- Instruction bus: Simple initiator socket to access instruction memory.
- IRQ line: Simple target socket to signal external IRQs.

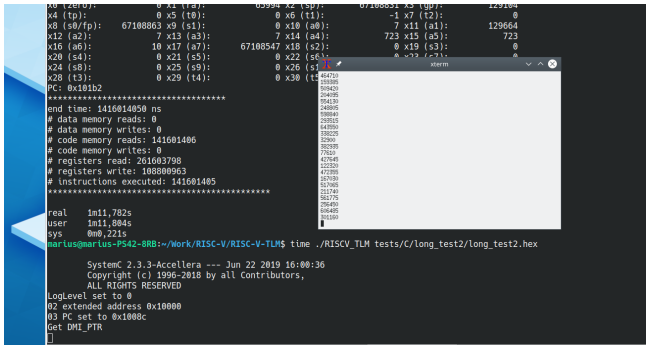

Figure 2: Simulator running with an xterm windows as terminal

2.2 Bus Controller

The simulator also includes a Bus controller in charge of the interconnection of all modules. The bus controller decodes the accesses address and does the communication to the proper module. In the actual status of the project, it contains two target sockets (instruction and data buses) and three initiator sockets: *Memory*, *Trace* and *Timer* modules, as described below.

2.3 Peripherals

The *Memory* module simulates a simple RAM memory, which is the main memory of the SoC, acting as instruction memory and data memory. This module can read a binary file in Intel HEX format obtained from the .elf file and load it to be the main program for the ISS. This module has a Simple target socket to be accessed that supports DMI to increase simulation speed.

The simulated Soc includes a very basic *Timer* module. This module includes two 64 bits register mapped to 4 addresses. On of this registers (*mtime*) keeps current simulated time in nanosecodns resolution. The second register (*mtimecmp*) is intended to program a future IRQ. The module triggers an IRQ using its Simple initiator socket.

The *Trace* module is a very simple tracing device, that outputs through a xterm window the characters received. This module is intended as a basic mimic of the ITM module of Cortex-M CPUs [10]. Figure 2 shows the simulator running with an xterm windows as output console.

Two other modules are included in the simulator: *Performance* and *Log*. The *Performance* module take statistics of the simulation, like instructions executed, registers accessed, memory accesses, etc. It dumps this information when the simulation ends. The other module allows the simulator to create a log file with different levels of information.

At maximum level of logging, each instruction executed is logged into the file with its name, address, time and register values or addresses accessed. The log file at maximum debug level shows information about the current time, PC value and the instruction executed. It also prints the values of the registers used. Figure 3 shows a real executed log file.

The log file at maximum debug level shows information about the current time, PC value and the instruction executed. It also prints

```
time 0 s: Using file: tests/C/long_test1/long_test1.hex
time 0 s: PC: 0x1008c. time 0 s: AUIPC x3 <- 0xf000 + PC (0x1f08c)
time 10 ns: PC: 0x10090. time 10 ns: ADDI: x3 + 1988 -> x3(0x1f850)
time 20 ns: PC: 0x10094. time 20 ns: ADDI: x3 + 460 -> x10(0x1fa1c)
time 30 ns: PC: 0x10098. time 30 ns: ADDI: x3 + 552 -> x12(0x1fa78)
time 40 ns: PC: 0x1009c. time 40 ns: C.SUB: x12 - x10 -> x12
```

Figure 3: Log file view

the values of the registers used. Figure 3 shows a real executed log file.

3 SOFTWARE IMPLEMENTATION AND TOOLCHAIN

The entire simulator is designed to work on pure bare-metal simulation. There is not direct communication between the simulator and the host machine, meaning for instance that *printf* implementation outputs directly to a host computer console. This is intended to do a simulation as similar to a real Hardware as possible, because the same exact code and the compiled binary that runs in the simulator will run in the real SoC.

For this reason the instructions *EBREAK* and *ECALL* are implemented in that way: *EBREAK* stops the simulation and dump some statistics. In a real system, has no sense to call *EBREAK* instruction and depending of the implementation can trigger a system reset or a *NOP*. The *ECALL* instruction raises an exception, dump statistics of the simulation and continues the execution for the same reason.

To supply the lack of semi-hosting options, the *Trace* module can be used to print out some information. With the use of proper helper functions, it is possible to use *printf()* -like functions. In this case, the *_write* function must be written to send the received data to *Trace* module as follows:

```
int _write(int file, const char *ptr, int len)
{
  int x;

  for (x = 0; x < len; x++) {
    TRACE =  *ptr++;
  }

  return (len);
}
```

The initial value for the Program Counter register (PC) is obtained from the HEX binary file and set before starting the simulation. The stack pointer register (SP) is set to last memory address.

This flexibility and the compatibility accomplished enables the use of the standard GCC cross compiler with little options:

```
-march=rv32imac -mabi=ilp32 --specs=nosys.specs
```

The options specifies the architecture and ABI (Application Binary Interface) and specifies the bare-metal option for newlib standard C library.

This allows complete use of C library on the application code, including math library, stdio and string libraries.

3.1 Docker version

A docker version of the simulator is provided [12]. It can be used to ease the installation and use of the simulator to avoid user to compile and gather all necessary libraries.

This image has been used in conjunction with another docker image that contains a riscv-toolchain. It can be used to ease the installation and use of the simulator, and specifically, to avoid the user to compile and gather all necessary libraries.

The simulator image is published and available in docker hub [13].

3.2 FreeRTOS

A porting of FreeRTOS version 10.2.1 were written for the simulated SoC [2]. The simulator is able to run this complex project without any error. The FreeRTOS test project includes 3 tasks that communicate and synchronize using one common queue. The two producer tasks use FreeRTOS' delay functions to suspend for a amount of time. Only one of the tasks prints out debug information.

4 TEST AND RESULTS

Different test were done to ensure the compatibility of the simulator. Also some performance results are presented from the same tests.

The compiler for RISC-V code is the RISC-V GCC version is 8.3.0 build with ABI configured to *ilp32* and architecture set to *rv32i*.

4.1 Tests Compliance

The simulator implements RISC-V RV32IMACZicsr_Zifencei V2.1 instruction set [20, 21] and it passes all tests in risc-test and riscv-compliance suites [16, 17]. The riscv-compliance tests have a coverage of 97.23% for RV32I, 89.95% for RV32IM and 59.68% for RV32IMC. These percentage means the number of all possible instructions and registers combinations are tested.

A more complex program, the *dhrystone* benchmark test is passed with correct results as well.

The project code has been statically checked with *coverity* by Synopsis. The analysis results in only 1 minor error found in TLM-2 library code but any error in the simulator code itself [18]. Also, code quality is checked with *Codacy* tool [5]. This tool checks for code quality, security, unused code, etc. The outcome of this tool is a A score, with only 10 minor warnings about code style.

In the next section is discussed the performance of this simulator.

4.2 Performance

A set of four program are written to test the performance of the simulator. Of these tests, test 1 checks memory transfer between two memory locations; test 2 and 3 perform arithmetic operations in three variables, one prints out the results and the other one is not using the console; the last test uses string manipulation functions from stdlib C library (printf, sprintf, strcpy).

All test do a end-less loop of some mathematical operations and prints out the result using *Trace* module. Each test is executed 3 times for different execution time (from 10 to aprox. 60 seconds execution time). The Figure 4 shows average of these 3 runs.

Its performance varies mainly with the level of the logging system due to huge I/O traffic in the log file. With lowest level o logging, the performance of the simulator is about 8 million of simulated

Table 1: Performance result. Values in instructions/second

Test	Native	Docker
Test1	8.252.929	3.854.110
Test2	6.298.774	3.291.465
Test3	8.921.763	3.754.295
Test4	12.899.367	4.375.651
Dhrystone	10.700.733	3.796.328

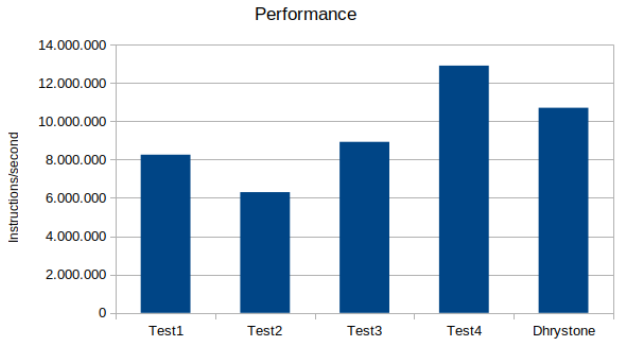

Figure 4: Execution results for all tests

instructions per second (see Table 1 and Figure 4) in a Intel Core i7-8550U CPU @ 1.88 GHz with 16 GB of memory. As a reference, in the same computer the Spike simulator performance is about 170 million of simulated instructions per second.

The low performance of Test2 can be due to the intensive use of the *Trace* module and the overhead it implies.

For the *Dhrystone* benchmark, it is executed with good results and the performance is about 7200 Dhrystones/second. It has been tested with 10.000, 250.000 and 500.000 loops of the Dhrystone test.

4.3 Docker version

The same tests has been run with the docker version of the simulator. The results are summarized in Table 1 and depicted in Figure 5.

In case of docker version, the performance has a penalty from 47% to up to 69% depending on the test. The performance of this version is depicted in Figure 5.

5 CONCLUSIONS

This paper introduces a new RISC-V simulator. It has been designed from scratch to simulate an entire SoC with simplicity on focus. It has been designed in SystemC and TLM-2 as language and modeling schema.

It has been presented the main architecture of the simulator, the software configuration and tools required. Followed by a brief discussion about the simulation performance and the conformance to the specifications.

The use of standards is important in any aspects of the engineering effort. In the case of system-level simulators, the existence of the TLM-2 and SystemC standards should be encourage and used by vendors and researchers to increase the interoperability and

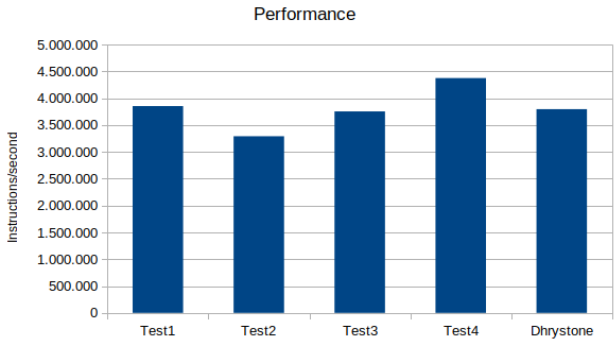

Figure 5: Execution results for all tests with the Docker version

re-usability of the components. This simple simulator is a first step towards this achievement.

REFERENCES

[1] 2012. IEEE Standard for Standard SystemC Language Reference Manual. *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)* (2012), 1–638.

[2] Inc Amazon Web Services. 2020. *FreeRTOS HomePage.* https://www.freertos.org/

[3] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Anaheim, CA) *(ATEC '05)*. USENIX Association, USA, 41.

[4] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[5] Codacy. 2020. *Codacy - RISC-V-TLM Dashboard.* https://app.codacy.com/manual/mariusmm/RISC-V-TLM/dashboard

[6] RISC-V foundation. 2020. *RISC-V Spike.* https://github.com/riscv/riscv-tools

[7] Imperas. 2020. *A Complete, Fully Functional, Configurable RISC-V Simulator.* https://github.com/riscv/riscv-ovpsim

[8] RISC-V International. 2020. *RISC-V Software Ecosystem Overview - Simulators.* https://riscv.org/software-status/#simulators

[9] Jonathan Bachrach Scott Beamer et al Krste Asanović, Rimas Avižienis. 2016. *The Rocket Chip Generator.* Technical Report Technical Report UCB/EECS-2016-17. EECS Department, University of California, Berkeley. https://github.com/chipsalliance/rocket-chip

[10] ARM Limited. 2020. *Cortex™-M3 Technical Reference Manual - ITM.* http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0337e/BABCCDFD.html

[11] Neethu Bal Mallya, Cecilia Gonzalez-Alvarez, and Trevor E Carlson. 2018. Flexible timing simulation of RISC-V processors with sniper. *Simulation* 4 (2018), 1.

[12] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.* 2014, 239, Article 2 (March 2014), 1 pages.

[13] Màrius Montón. 2020. *mariusmm/riscv-tlm.* https://hub.docker.com/repository/docker/mariusmm/riscv-tlm

[14] Màrius Montón. 2020. *RISC-V-TLM Simulator.* https://github.com/mariusmm/RISC-V-TLM

[15] Morten Petersen. 2020. *Ripes.* https://github.com/mortbopet/Ripes

[16] RISCV.org. 2020. *RISC-V Compliance Task Group.* https://github.com/riscv/riscv-compliance/

[17] RISCV.org. 2020. *RISC-V Unit Tests.* https://github.com/riscv/riscv-tests

[18] Synopsys. 2020. *Coverity - RISC-V-TLM.* https://scan.coverity.com/projects/mariusmm-risc-v-tlm

[19] Tuan Ta, Lin Cheng, and Christopher Batten. 2018. Simulating multi-core RISC-V systems in gem5. In *Workshop on Computer Architecture Research with RISC-V.*

[20] Andrew Waterman and Krste Asanovi. 2019. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA.* Technical Report Version 20191213. RISC-V Foundation.

[21] Andrew Waterman and Krste Asanovi. 2019. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture.* Technical Report Version 20190608-Priv-MSU-Ratified. RISC-V Foundation.