Proceedings of the 26th International Conference *"Mixed Design of Integrated Circuits and Systems"*
June 27-29, 2019, Rzeszów, Poland

Design and Verification Environment for RISC-V Processor Cores

Adrian Oleksiak[1], Sebastian Cieślak[1], Krzysztof Marcinek[1,2], Witold A. Pleskacz[1]
[1]Warsaw University of Technology, Institute of Microelectronics and Optoelectronics, Warsaw, Poland
[2]ChipCraft Sp. z o.o., Lublin, Poland
email: A.Oleksiak@stud.elka.pw.edu.pl, S.Cieslak@stud.elka.pw.edu.pl,
K.Marcinek@{imio.pw.edu.pl,chipcraft-ic.com}, W.Pleskacz@imio.pw.edu.pl

***Abstract*—Processor verification is a very complex task. Even simple system has plenty of blocks which must work together in a determined way. Modern digital systems use advanced techniques to improve throughput and reduce power consumption. It enhances risk of error and consequently verification process requires a significant and continuous resources. This article presents the approach to create design and verification environment for RISC-V processor cores. The environment consists of behavioral golden reference model and online disassembler module, as well as a set of scripts for setting up software infrastructure. Golden reference model running in master-checker mode with core under design allows faulty behavior to be detected instantly while running direct tests or random verification technique.**

***Keywords*—RISC-V; verification; golden reference model; master-checker**

I. Introduction

Nowadays, the RISC-V [1] Instruction Set Architecture (ISA) becomes more and more popular. Open-source character and support from community show that this interest should continue to grow in the future. RISC-V is one of many different open architectures, but it has a few advantages, which distinguishes it from other. At first it is completely free, it uses Berkeley Software Distribution (BSD) open-source license. That's makes RISC-V more attractive for the commercial use. Another important thing is a software support. Community provides complete software stack – compiler toolchains, suitable Integrated Development Environment (IDE) and operating systems, which can be handled in embedded application. RISC-V has a widely documented hardware and software ecosystem. Base specification is now frozen, so ISA is very stable. Because of this, software written for RISC-V will run on all similar RISC-V compliant cores now and in the future. One of the major attributes of RISC-V ISA is extensibility, which allows constructing system fulfilling specific expectation. The RISC-V architecture is developed by non-profit RISC-V Foundation [2] with more than 100 members including Google, Nvidia or Qualcomm. Organization takes care of software and hardware development. This assistance and cooperation with scientific centers as well as industry may guarantee long support to this architecture [3], [4], [5].

Described arguments encouraged our team to do research on retargeting already available processor core to RISC-V architecture [6]. As RISC-V is heavily influenced by MIPS architecture [7], we decided to use CC100-C processor core delivered by ChipCraft Sp. z o.o. company [8]. The used core is based on legacy MIPS-II architecture and started before RISC-V was widely recognized, as a PhD research project on microprocessor architectures and instruction set extensions [9]. The developed IP component library was used also in dedicated SoCs for telehealth and telemedicine [10], [11], [12] as well as in single-chip GNSS receiver [13]. This work is focused on retargeting the design and verification environment, while the complementary work is carried out on retargeting the processor RTL model.

The main goal of this work was to prepare environment, helpful to processor core design and verification. Determined target was achieved by implementing two units: emulator, acting as golden reference model, and disassembler. Both are written in behavioral Verilog language to be used in the same environment as designed processor core. Emulator module is coupled with the RTL processor model's executed instructions flow. In case of any inconsistency, emulator instantly reports a particular error. Disassembler module shows instruction executed by the running core in a human understandable ASCI format. The entire environment was prepared to support all leading simulators, especially opensource compiler – Icarus Verilog [14] and wave viewer GTKWave [15]. The developed environment can be used to support design and verification of other RISC-V processor cores. The only current limitation is that emulator expects not more than one instruction to be retired per clock cycle by each processor core. In the future, the designed environment will be used to design low-power single stage RISC-V processor core which will be share to the open source community on GPL license.

II. Preliminaries

There are two main verification techniques. Formal verification uses mathematical method to check functional correctness of design. RISC-V community has released the *riscv-formal* framework for formal verification of RISC-V processors [16]. Processor communicates with *riscv-formal* engine using the RISC-V Formal Interface (RVFI) [17]. Interface is an output-only trace port for signals generated by the processor core. Another, more classical approach, depends on functional verification. Simulation environment generates input stimuli for device-under-test (DUT) and checks weather the result are within the expected range. Additionally, environment should return coverage of tests in order to test all possible scenarios [18].

In the functional verification processes, most commonly, an input stimuli for processor is a program written in assembly or C language. Input code can be obtain in two ways – direct tests and random tests. Direct tests explicitly specify input signal. RISC-V foundation shares *riscv-tests* [19] and *riscv-compliance* [20] repositories. They consists of dedicated libraries which are focused on testing basic functionality of available instructions described in RISC-V specification. The second approach is to generate random tests to check DUT under hard to predict conditions, due to what, hard to include in direct tests. The *riscv-torture* [21] is a test generator for RISC-V ISA. It is written is *scala* language and generates RISC-V assembly. The generator originally uses *spike* [22] simulator as a golden reference model and compares it to processors output signature. The other worth mentioning tool is Csmith [23] generating random C programs based on C99 standard or Yarpgen [24] which produces correct runnable C/C++ program.

III. Verification Environment

The verification environment supports both direct and random verification technique. It uses master-checker relation, where processor is a master device and emulator serves as a golden reference model returning expected results (Fig. 1). The results are compared online one instruction after another instantly indicating faulty behavior. The DUT processor core feeds the emulator module with the following signals and data:

- system clock,
- ready for retirement signal,
- program counter,
- instruction code,
- ALU result,
- load-store unit (LSU) results – address and data,
- exception signal.

Emulator module consists of two main blocks. First block – *emu_proc* describes the processor DUT behavior. This block contains four further subblocks – *emu_inst* emulating program memory, *emu_ram* interfacing with RAM memory, *emu_spram* for tightly-coupled core memories and *emu_core*. The *emu_core* module is a simple high abstract level implementation of RISC-V architecture. The emulator is not a synthesizable module, so it can use abstract Verilog syntax

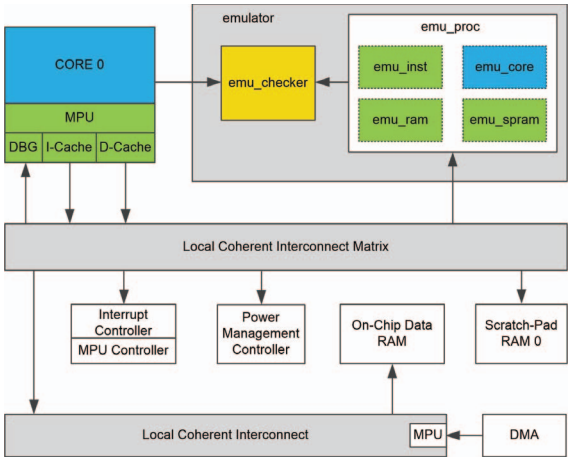

Fig. 1. Emulator connection with processor under test.

constructions, which simplifies its source code. Unlike the RTL core model, emulator calculates instructions results instantly. Lack of instruction pipeline queues, integrated buffers, stall and forwarding logic, branch prediction, etc., reduces the risk of errors in emulator model, however still allowing verification of these structures in the RTL design.

The current *emu_core* limitation is that it supports retirement of only one instruction each clock cycle per processor core. This limits the developed environment to single-issue processor cores or to simple superscalar processors with more than one execution ALU but with the maximum of one in-order retirement per clock cycle. On the other hand, the emulator unit can still act as the golden reference model and be used to calculate program signatures used in *riscv-torture* or Csmith.

The emulator has also limited support for Control and Status Registers (CSRs) [25]. It relies on the data provided from the DUT core. The exception is the *mtvec* register which has to be tracked precisely to calculate executed program flow. The exception detection is limited to the cases that can be directly evaluated from the program code. In case of unpredicted exceptions like bus errors, or interrupt exception, the emulator core follows the results provided by the processor code. Moreover, the emulator core is written in that way, it supports undefined states like fetching instruction from incorrect locations or loading data from uninitialized cells. As the result, the developed environment can be used for instruction-accurate failure detection even using randomly generated programs that can easily lead to exposure of unpredicted states.

The *emu_ram* block can work in two modes. In the first one it internally implements the entire operating memory space. As the results, besides the processor core, also data cache logic and data interconnect can be verified. When the simulated processor has more than one core, or simulation programs exploits DMA transfers, maintaining data coherency is extremely difficult. In such a case, the *emu_ram* can be configured to trust the data provided from the data interconnect verifying the processor core only.

The *emu_checker* block is responsible for comparing results returned by the processor core with values computed by the *emu_proc* module. There are few important values which must be checked. In case of *register-register* or *integer-register* instructions, module compares only data result. When processor core execute jump instruction, *emu_checker* verifies the program counter. The last group supported by the checker are load and store instructions. The RAM address and load/store data is compared with the golden reference model. When error occurs, checker pauses the simulation and displays message in the simulator console. The message contains information about program counter, instruction code, error source and expected values (Fig. 2).

```
EMU_ERROR:
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!! PC: 0x000003ec INST: 0x0ff00e93
!!!! EMULATOR 0: LOAD DATA ERROR
!!!! ADDRESS: 0xf0033000
!!!! RESULT: 0xf0033000; SHOULD BE: 0x00001880
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

Fig. 2. Detected processor core error example.

```
29665.0 ns core0: 0x000003c8 0x000022b7  lui     t0    , 0x00002             [0x00002000]
29675.0 ns core0: 0x000003cc 0x80028293  addi    t0    , t0    , 0x800       [0x00001800]
29685.0 ns core0: 0x000003d0 0x3002b073  csrrc   zero  , t0    , mstatus                  [0x00001800]
29695.0 ns core0: 0x000003d4 0x00000297  auipc   t0    , 0x00000             [0x000003d4]
29705.0 ns core0: 0x000003d8 0x01028293  addi    t0    , t0    , 0x010       [0x000003e4]
29735.0 ns core0: 0x000003dc 0x34129073  csrrw   zero  , t0    , mepc                     [0x000003e4]
29755.0 ns core0: 0x000003e0 0x30200073  mret
29765.0 ns core0: 0x000003e4 0x0ff00513  addi    a0    , zero  , 0x0ff       [0x000000ff]
29775.0 ns core0: 0x000003e8 0xc0001573  csrrw   a0    , zero  , cycle       [0xf0033000]
29785.0 ns core0: 0x000003ec 0x0ff00e93  addi    t4    , zero  , 0x0ff       [0x000000ff]
29795.0 ns core0: 0x000003f0 0x00b00193  addi    gp    , zero  , 0x00b       [0x0000000b]
29815.0 ns core0: 0x000003f4 0x03d51463  bne     a0    , t4    , 0x0028
29835.0 ns core0: 0x0000041c 0x0ff0000f  fence
29875.0 ns core0: 0x00000420 0x00018063  beq     gp    , zero  , 0x0000
29885.0 ns core0: 0x00000424 0x00119193  slli    gp    , gp    , 0x01        [0x00000016]
```

Fig. 3. Sample disassembler output.

Disassembler module was created in order to display instruction, which was executed by processor core. As a result the debug team can analyze executed program code. Disassembler, in the same way like checker, displays message in the simulator console. The disassembled instructions and their results are presented in a human readable ASCII format to support debugging process (Fig. 3).

Additionally the disassembler module incorporates dedicated counters for instruction occurrence count and gathering core statistics like instruction and data bus utilization or branch prediction performance (Fig. 4).

IV. Programming Environment

Processor tests are written as an assembly or C code, so they need to be compiled to machine format to be executed by the processor. For this purpose the Newlib cross-compiler from the riscv repository [26] was used. This compiler supports two architectures *rv32i* and *rv64i* plus standard extensions – atomics, multiplication and division, float and double. During the compilation process user can choose which extension is implemented in core. The compiler output is further processed by the developed dedicated software to translate Srec Motorola format to hexadecimal representation of assembler instructions, which can be use by simulation environment.

```
Statistics Core 0:
    Execution time:
      Cycles:               2314
      Time:                 23140.9 ns
    Executed instructions:
      Count:                182
        32-bit:             100.00%
        16-bit:             0.00%
    Pipeline stall cycles:
      Count:                158
      Time:                 6.83%
    I-Cache stall cycles:
      Count:                2059
      Time:                 88.98%
    D-Cache stall cycles:
      Count:                38
      Time:                 1.64%
    Utilization:
      IPC:                  0.08
      Cycles/inst:          12.71
      Time:                 7.87%
    Branches:
      Count:                42
      Missed:               13
      Hit rate:             69.05%
```

Fig. 4. Sample processor core statistics.

TABLE I.
PRELIMINARY RISCV-TESTS RESULTS

RISC-V test	emulator	RTL model
JAL	OK	ERROR
DIV	OK	ERROR
REM	OK	ERROR
CSR	OK	ERROR
FENCE_I	OK	ERROR
ILLEGAL	ERROR	OK
MA_FETCH	OK	ERROR
SHAMT	ERROR	OK
MA_ADDR	OK	ERROR
BREAKPOINT	ERROR	OK

V. Results and Conclusions

After preliminary emulator and RTL code retargeting was done, we started to perform first tests. For this purpose we used *riscv-test* and *riscv-compliance* libraries to generate reliable processors batch file. This libraries helped us discover few errors in user-level instruction and much more errors in machine mode instruction (TABLE I). What was not surprising, after preliminary implementations, errors occurred both in emulator and RTL processor models. Most often in one of them at a time. User-level errors include wrong behavior of jump instruction in case of bad misaligned address. Other errors pertain to CSR instructions and bad exceptions operation. All of the encountered errors were corrected to the date.

The major advantage of the presented environment is the instruction accurate identification of the error location. Each of RISC-V test tools inform about overall faulty behavior based on generated signature without precise error identification. This allowed us to find bugs easily using RISC-V test tools and many third-party programs which do not have self-testing mechanism. The developed environment has also some limitations. As mentioned earlier, when processor uses more than one core or DMA is active, emulator assumes that data form RAM and data interconnect is correct. RAM emulator can be used only when only one core is running and DMA is disabled. Operation related with control and status register are not independent, emulator cannot check their content. In case of

instruction which use CSR, emulator uses value provided by the processor. The similarly situation is with some unpredicted exceptions like bus errors, or interrupt exception.

After debugging process, the developed environment with the RTL processor model were subjected to perform more advanced C programs like Dhrystone [27] and Coremark [28] benchmarks. Both programs were run in master-checker mode and both terminated normally returning expected results. No errors from the checker module means, that both emulator and RTL model executed each instruction providing the same result.

In conclusion, we have succeeded in creating environment which helped us to retarget exiting processor core to RISC-V architecture. The developed environment can be also used to design and verify new implementations of RISC-V architecture processors. In the near future we plan to run *riscv-torture* and to add support for formal verification using *riscv-formal* framework. Random test generator will increase diversity of processor programs, whereas *riscv-formal* will provide mathematical prove of RISC-V ISA compliance.

REFERENCES

[1] A. Waterman and K. Asanovic, "The RISC-V Instruction Set Manual; Volume I: User-Level ISA," SiFive Inc. and CS Division, EECS Department, University of California, Berkeley, 2017.

[2] RISC-V Foundation. [Online]. Available: https://riscv.org.

[3] K. Patsidis, D. Konstantinou, C. Nicopoulos, G. Dimitrakopoulos, "A low-cost synthesizable RISC-V dual-issue processor core leveraging the compressed Instruction Set Extension," Microprocessors and Microsystems, vol. 61, pp.1-10, Sept. 2018.

[4] V. Herdt, D. Große, H. M. Le, R. Drechsler, "Extensible and Configurable RISC-V based Virtual Prototype," Forum on Specification & Design Languages, pp 5-16, Sept. 2018.

[5] Why RISC-V. [Online]. Available: https://riscv.org/why-risc-v/.

[6] S. Cieslak, A. Oleksiak, K. Marcinek, W. A. Pleskacz, "Retargeting the MIPS-II CPU Core to the RISC-V Architecture," Int. Conf. Mixed Design of Integrated Circuits & Systems, June 2019. (forthcoming)

[7] MIPS Architectures. [Online]. Available:. https://www.mips.com.

[8] ChipCraft Sp. z o.o.. [Online]. Available: http://chipcraft-ic.com/.

[9] K. Marcinek, W. A. Pleskacz, "AGATE-towards designing a low-power chip multithreading processor for mobile software defined radio systems," IEEE Int. Symp. Des. Diagn. Electron. Circuits Syst.,pp. 26-29, Apr. 2012.

[10] K. Marcinek, M. Plasota, A. Wielgus, W. A. Pleskacz, "Implementation of the ADELITE Microcontroller for Biomedical Applications", IEEE Int. Symp. Des. Diagn. Electron. Circuits Syst.,pp. 271-274, Apr. 2015.

[11] K. Siwiec, K. Marcinek, T. Borejko, A. Jarosz, J. Kopański, E. Kurjata-Pfitzner, P. Narczyk, M. Plasota, A. Wielgus, W. A. Pleskacz, "A CMOS system-on-chip for physiological parameters acquisition, processing and monitoring", Int. Conf. Mixed Design of Integrated Circuits & Systems, pp. 37-42, June 2015.

[12] K. Siwiec, K. Marcinek, P. Boguszewicz, T. Borejko, A. Halauko, A. Jarosz, J. Kopański, E. Kurjata-Pfitzner, P. Narczyk, M. Plasota, A. Wielgus, W. A. Pleskacz, "BioSoC: Highly integrated System-on-Chip for health monitoring", IEEE Int. Symp. Des. Diagn. Electron. Circuits Syst.,pp. 1-6, Apr. 2016.

[13] NaviSoC project. [Online]. Available: http://navisoc.com/.

[14] Icarus Verilog. [Online]. Available: http://iverilog.icarus.com.

[15] GTKWave. [Online]. Available: http://gtkwave.sourceforge.net.

[16] C. Wolf, RISC-V Formal Verification Framework. [Online]. Available: https://github.com/cliffordwolf/riscv-formal.

[17] S. Hoover, Á. Hadnagy, "Formally Verifying WARP-V, an Open-Source TL-Verilog RISC-V Core Generator", arXiv.or, Nov. 2018.

[18] L. Gon, O. Diessel, "Functional Verification of Dynamically Reconfigurable FPGA based Systems" Springer, Heidelberg (2015).

[19] Riscv-tests. [Online]. Available: https://github.com/riscv/riscv-tests.

[20] Riscv-compliance. [Online]. Available: https://github.com/riscv/riscv-compliance.

[21] Riscv-torture. [Online]. Available: https://github.com/ucb-bar/riscv-torture.

[22] Spike RISC-V ISA Simulator. [Online]. Available: https://github.com/riscv/riscv-isa-sim.

[23] Csmith. [Online]. Available: https://embed.cs.utah.edu/csmith/.

[24] Yarpgen. [Online]. Available: https://github.com/intel/yarpgen.

[25] "The RISC-V Instruction Set Manual; Volume II: Privileged Architecture," SiFive Inc. and CS Division, EECS Department, University of California, Berkeley, 2017.

[26] RISC-V Compiler Toolchain. [Online]. Available: https://github.com/riscv/riscv-gnu-toolchain.

[27] "Dhrystone Benchmark - History, Analysis, "Scores" and Recommendations", Alan R. Weiss, EEMBC Certification Laboratories. [Online]. Available: http://ebenchmarks.com/.

[28] CoreMark. [Online]. Available: https://www.eembc.org/coremark/.